Fault-Tolerant Consistency Management in Model-Driven Engineering

Nils Weidmann

Faculty of Computer Science, Electrical Engineering and Mathematics

Paderborn University

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Paderborn, December 20, 2021

There was never a sound beside the wood but one,
And that was my long scythe whispering to the ground.
What was it it whispered? I knew not well myself;
Perhaps it was something about the heat of the sun,
Something, perhaps, about the lack of sound—
And that was why it whispered and did not speak.
It was no dream of the gift of idle hours,
Or easy gold at the hand of fay or elf:
Anything more than the truth would have seemed too weak
To the earnest love that laid the swale in rows,
Not without feeble-pointed spikes of flowers
(Pale orchises), and scared a bright green snake.
The fact is the sweetest dream that labor knows.
My long scythe whispered and left the hay to make.

Robert Frost: Mowing (1913)

Acknowledgements

This thesis would not have been possible without the support of many others, to whom I want to express my gratitude at this point. As there are quite a lot of people who supported me during the last four years, I can mention just a few of them explicitly, and further appreciate the help of others as members of one or more groups.

First of all, I want to thank Anthony Anjorin and Gregor Engels for being my supervisors in the first and second half of my time as a PhD student, respectively. Tony, from the first semester of my Master studies on, you have shared your enthusiasm for MDE research with me. Whenever I needed the help of an experienced researcher to discuss new ideas, transform drafts into papers, or overcome technical challenges, you spent far more time and energy on it than one could ever expect.

Gregor, I can imagine how difficult it is to supervise ten or more PhD students with diverse topics at the same time, but it is probably even more challenging to take over someone after one and a half years without pushing the topic into another direction. From you, I learned how to communicate results in a way that they can be understood by a broader audience. Thank you!

Furthermore, I am grateful for the external review of my thesis by Andy Schürr. From you, Andy, I received valuable feedback for early ideas, papers, and especially for the thesis itself. I further like to thank Stefan Sauer and Thomas Vogel for being members of my examination board.

For giving me the opportunity to work in two research projects with different partner companies, I want to thank Stefan Sauer, as well as Bernd and Lisa Kleinjohann. I got valuable insights into the business processes of such companies, established interesting contacts, and got a view beyond the horizon of my PhD thesis.

Also, I would like to thank all my present and former colleagues at the SICP, C-Lab, and the AG Engels at Paderborn University, and the Real-Time Systems Lab at TU Darmstadt. I felt comfortable and welcome every single day, became part of a productive team, and enjoyed spending my time with you both during and after work. Many colleagues became friends, and I am sure that this will outlast our time at university.

Numerous students who wrote their Bachelor and Master theses under our supervision, and/or participated in the project group "VICToRy", contributed a lot to my research as well. Without your commitment, it would not have been possible to develop the mature tool support that exists today. The same holds for all SHK and WHB students, who substantially helped us to make a success of different research projects.

Science is about working in a team, so special thanks go to all co-authors with whom I worked on different research papers. Similarly, I am also grateful for all – positive and negative – feedback we received from numerous anonymous reviewers: With the help of your suggestions and impressions, it was possible to improve the quality of publications.

Last but not least, I want to thank my friends and my family, especially my parents Gabriele and Reinhard and my sister Lara, for their unconditional support in these times. As more than half of my PhD studies took place during the COVID-19 pandemic, content-related problems often receded to the background. Thank you for reminding me that life happens before, after, and also while doing research!

Abstract

Models play an important role in nowadays' software engineering processes, providing stakeholders with a suitable level of abstraction for specifying software systems. While models are often used for design and documentation purposes, Model-Driven Engineering (MDE) places models in the centre of the development process, such that source code and test cases can be generated directly from the specification.

In practice, software development of this kind involves multiple models which often have a semantic overlap, and which are usually created by multiple (teams of) domain experts concurrently. To develop software systems of high quality, it is necessary to establish mechanisms to maintain and restore consistency between such models, which are commonly denoted as consistency management operations. While such operations can be specified separately in principle, Bidirectional Transformation (BX) approaches pursue the idea of formally describing a consistency relation between two models, and subsequently deriving all consistency management operations from this relation.

While MDE established itself as a subfield of software engineering, a frequently named argument against the use of MDE techniques in practice is its lack of flexibility, though, especially in later phases of the development process. Current consistency management tools can only operate on consistent, i. e., fault-free input models. This forces users to remove all faults from the input models, before being able to continue working with the respective modelling tools.

To address these issues, consistency management is considered as an optimisation problem in the scope of this thesis. Instead of enforcing perfect consistency, a solution that is consistent to the largest possible extent is determined in case of faulty input models. A hybrid framework is proposed that synergetically combines Triple Graph Grammars (TGGs) – a declarative, rule-based BX approach – and optimisation techniques, including Integer Linear Programming (ILP) and different meta-heuristics. The hybrid framework supports various consistency management tasks, including model transformations, consistency checking, and the synchronisation of (concurrent) updates. Advanced language features such as graph constraints and attribute conditions increase the expressive power of the consistency management operations. Based on this approach, users can be equipped with powerful and flexible consistency management tools, such that the applicability of MDE techniques in practice is improved.

The conceptual solution is entirely implemented as part of eMoflon, an MDE tool suite for various model management tasks. To support the users' understanding for different consistency management processes, the MDE debugger VICToRy was developed as an add-on component for eMoflon. The applicability of the fault-tolerant framework in practice is demonstrated via two industrial case studies, including a BX between a formal and a semi-formal language for railway systems engineering, and a model-driven solution for optimal test scheduling, i. e., allocating human resources and testing tasks.

Zusammenfassung

Modelle spielen in heutigen Softwareentwicklungsprozessen eine wichtige Rolle, indem sie Stakeholdern ein angemessenes Abstraktionsniveau zur Spezifizierung von Softwaresystemen bieten. Während Modelle oft zu Entwurfs- und Dokumentationszwecken verwendet werden, stellt Modellgetriebene Softwareentwicklung (MDE) Modelle ins Zentrum des Entwicklungsprozesses, sodass Quellcode und Testfälle direkt aus der Spezifikation generiert werden können.

In der Praxis umfasst Softwareentwicklung dieser Art mehrere Modelle, welche oft semantische Überschneidungen aufweisen, und normalerweise von mehreren Domänenexperten(-teams) parallel erstellt werden. Um hochqualitative Softwaresysteme zu entwickeln, ist es notwendig, Mechanismen zur Konsistenzerhaltung und -wiederherstellung zwischen solchen Modellen einzusetzen, welche üblicherweise als Konsistenzmanagementoperationen bezeichnet werden. Während solche Operationen prinzipiell unabhängig voneinander spezifiziert werden können, verfolgen Ansätze der Bidirektionalen Transformation (BX) die Idee, eine Konsistenzrelation zwischen zwei Modellen formal zu beschreiben, und alle Konsistenzmanagementoperationen anschließend aus dieser Relation abzuleiten.

Während MDE sich als Teilbereich des Software Engineering etabliert hat, ist ein häufig genanntes Argument gegen den Einsatz von MDE-Techniken in der Praxis jedoch deren fehlende Flexibilität, besonders in späteren Phasen des Entwicklungsprozesses. Aktuelle Konsistenzmanagementwerkzeuge können nur konsistente, d.h. fehlerfreie Eingabemodelle verarbeiten. Das zwingt Anwender, zunächst alle Fehler aus den Eingabemodellen zu entfernen, bevor sie in die Lage versetzt werden, mit den jeweiligen Modellierungswerkzeugen weiterzuarbeiten.

Um diese Themen zu adressieren, wird im Rahmen dieser Arbeit Konsistenzmanagement als ein Optimierungsproblem betrachtet. Anstatt perfekte Konsistenz zu erzwingen, wird im Fall von fehlerhaften Eingabemodellen eine Lösung bestimmt, die so weit wie möglich konsistent ist. Ein hybrides Rahmenwerk wird vorgestellt, welches Triple-Graph-Grammatiken (TGGs) - einen deklarativen, regelbasierten BX-Ansatz - mit Optimierungstechniken wie Ganzzahliger Linearer Optimierung (ILP) und verschiedenen Meta-Heuristiken synergetisch kombiniert. Das hybride Rahmenwerk unterstützt verschiedene Konsistenzmanagementaufgaben, wie Modelltransformationen, Konsistenzchecks, und die Synchronisierung von (parallelen) Änderungen. Weitergehende Sprachfeatures wie Graph-Constraints und Attributbedingungen erhöhen die Ausdrucksmächtigkeit der Konsistenzmanagementoperationen. Basierend auf diesem Ansatz können Nutzer mit mächtigen und flexiblen Konsistenzmanagementwerkzeugen ausgestattet werden, sodass die Anwendbarkeit von MDE-Techniken in der Praxis verbessert wird.

Die konzeptionelle Lösung ist vollständig als Teil von eMoflon implementiert, einer MDE-Tool-Suite für verschiedene Modellierungsaufgaben. Um das Verständnis verschiedener Konsistenzmanagementprozesse auf Anwenderseite weiter zu fördern, wurde der MDE-Debugger VICToRy als eine Zusatzkomponente für eMoflon entwickelt. Die praktische Anwendbarkeit des fehlertoleranten Rahmenwerks wird mithilfe zweier industrieller Fallstudien verdeutlicht, einschließlich einer BX zwischen einer formalen und einer semi-formalen Sprache im Bereich Bahnsystemtechnik, sowie einer modellgetriebenen Lösung zur optimalen Testplanung, welche personelle Ressourcen und Testaufgaben einander zuweist.

Contents

List of Figures

List of Tables

List of Definitions

List of Abbreviations

AADL	Architecture Analysis and Design Language
AC	Application Condition
API	Application Programming Interface
ATL	Atlas Transformation Language
BX	Bidirectional Transformation
CPS	Cyber-Physical System
DSE	Design Space Exploration
DSL	Domain-Specific Language
EA	Enterprise Architect
EMF	Eclipse Modeling Framework
eMSL	eMoflon Specification Language
fUML	foundational UML
GA	Genetic Algorithm
GPL	General Purpose Language
GT	Graph Transformation
IDE	Integrated Development Environment
ILP	Integer Linear Programming
JDK	Java Development Kit
JRE	Java Runtime Environment
JTL	Janus Transformation Language
JVM	Java Virtual Machine
LCDP	Low-Code Development Platform
MBSE	Model-Based Software Engineering
MDE	Model-Driven Engineering
NAC	Negative Application Condition
NSGA-II	Non-Dominated Sorting Genetic Algorithm II

OCL Object Constraint Language

OMG Object Management Group

PAC Positive Application Condition

PL Programming Languages

PN Petri Net

QVT-O Query/View/Transformation-Operational

QVT-R Query/View/Transformation-Relations

SA Simulated Annealing

SAT Boolean Satisfiability Problem

SBSE Search-based Software Engineering

SE Software Engineering

SLR Systematic Literature Review

SMT Satisfiability Modulo Theories

SysML Systems Modeling Language

TGG Triple Graph Grammar

UI User Interface

UML Unified Modeling Language

XMI XML Metadata Interchange

Part I

Foundations and Related Work

1 Introduction and Motivation

Software plays an ubiquitous role for nowadays society, with even increasing importance through advancing digitalisation. One reason why software could pervade many areas of life is the emergence of software engineering methods as a reaction to the software crisis [Dij72]. The need for defined and standardised procedures for software development became apparent to reduce error rates and thereby life-cycle costs, making the use of software profitable in industry. During the last 50 years, software development became an engineering discipline with different branches and paradigms, such as structured programming, formal methods, and agile development [Bro18].

Model-Driven Engineering (MDE), another sub-domain of software engineering, places *models* in the centre of the development process. Models are used to provide the stakeholders with a suitable level of abstraction [Béz05], which is crucial for building complex software systems of high quality. The use of models enables *domain experts* to specify, validate and maintain software systems, as no advanced programming skills are required to create these models. Thereby, communication problems that lead to misunderstood requirements are avoided by design to some extent. In the beginning of the 21st century, MDE established itself as a mature software engineering paradigm and was successfully applied to industrial use cases [HRW11].

As a special form of Model-Based Software Engineering (MBSE), in which models are used to support a common understanding of all involved stakeholders and for documentation purposes, MDE treats models as primary artefacts throughout the entire software development process. This involves tasks such as requirements specification, system design, code generation and model-based testing. Well-known examples for modelling languages are the Unified Modeling Language (UML)[1] as a general-purpose modelling language for software development, and the Systems Modeling Language (SysML)[2], which both restricts and leverages the expressiveness of UML for systems engineering.

When developing software systems of realistic size, multiple models are usually used. They describe the system from different perspectives, are often created and maintained by different (teams of) domain experts and have a semantic overlap. To build software systems that meet high quality standards, it is therefore essential to maintain *consistency* between semantically interrelated models throughout the development process to preserve the shared information. This task is commonly denoted as model management or *consistency management*. In the context of MDE, intra-model consistency denotes a property that states whether a model is consistent in isolation, whereas inter-model consistency refers to semantically interrelated models.

Maintaining consistency between multiple models recently gained importance in several subfields of computer science. The development of intelligent modelling environments, e.g., involves different sorts of models, which interact with each other in a shared ecosystem [MCK+20]. Descriptive models are used to describe the real world on a higher abstraction level, whereas prescriptive models are used to define how a software system should operate (which is in line with the role of models in MDE). Finally, in the emerging field of artificial intelligence, predictive models are used to derive knowledge from data sources.

[1] https://www.omg.org/spec/UML/2.5.1/PDF
[2] https://www.omg.org/spec/SysML/1.6/PDF

Consistency management can be broken down into several subtasks: *Model transformation* describes the process of transforming a model into another model that is expressed in a different language, which means that the new model is generated from scratch, whereby certain rules or constraints must be respected to reach a consistent state. When applying *model synchronisation*, in contrast, we assume that two models already exist and have been consistent with each other at some point of time. One model has been modified since then, i.e., model elements have been created, deleted, or attribute values have changed. These updates have to be *propagated* to the other model to restore consistency. In settings where multiple teams work on their models *in parallel*, concurrent updates on all involved models can occur, such that update propagation in various directions is necessary, also denoted as *concurrent (model) synchronisation*.

A technique of special interest is Bidirectional Transformation (BX), in which the synchronisation of two models is performed based on the same specification for all involved tasks [ASCG+16]. By defining the consistency relation once, the previously mentioned tasks can be automatically derived, such that it is not necessary to implement, e.g., forward and backward transformation separately. This reduces the implementation effort for the tool developer and supports the understandability and trustworthiness of the tool, as only one specification needs to be understood by the integration expert. In order to characterise the involved stakeholders in more detail and motivate the role of fault-tolerance in this scenario, a concrete example from the railway systems engineering domain is used in the upcoming Sect. 1.1 and 1.2, before the structure and main contributions of this thesis are introduced in Sect. 1.3, and an overview of published research that builds a basis for this thesis is provided in Sect. 1.4.

1.1 Motivation and Problem Statement

Maintaining consistency between semantically interrelated models has been an intensively studied research topic for several years and has attracted interest in industry as well [GHN10, BPD+14, AYL+18]. To motivate the aspect of fault-tolerance in such processes, we consider a collaborative systems engineering scenario of the railway domain, involving the semi-formal language SysML and the formal language Event-B [AH07, Hoa13]. SysML as a de-facto standard for modelling mechatronic systems is often used in safety-critical contexts, as it is well-known among systems engineers. As software faults in safety-critical systems can cause serious damage, the developed components need to be verified and validated to guarantee that they work as expected. A formal proof of correctness cannot be given for SysML models, such that the system specifications need to be transformed into models of a formal language (Event-B in the concrete use case), and enriched with safety constraints that cannot be expressed in SysML. The formal semantics of Event-B makes it possible to verify safety-related properties, but is only understood by experts of this field and therefore not adequate for modelling (software) systems. Consequently, there is a need for transforming SysML models into Event-B code and, as a reaction to the verification process, propagating changes back to the SysML models to keep all software engineering artefacts up to date.

The automated process for systems engineering with SysML and Event-B is depicted in Fig. 1.1 in an informal notation inspired by Stevens [Ste17]. First, the SysML engineer creates an initial version of the system model (i). Subsequently, an integration expert transforms the SysML model to Event-B code (ii). An Event-B expert can now add safety constraints (iii.1) to validate the model to guarantee formal properties. According to the

validation results, the Event-B model needs to be updated (iii.2). In order to also keep the SysML model up to date, the changes have to be propagated back by the integration expert (iv).

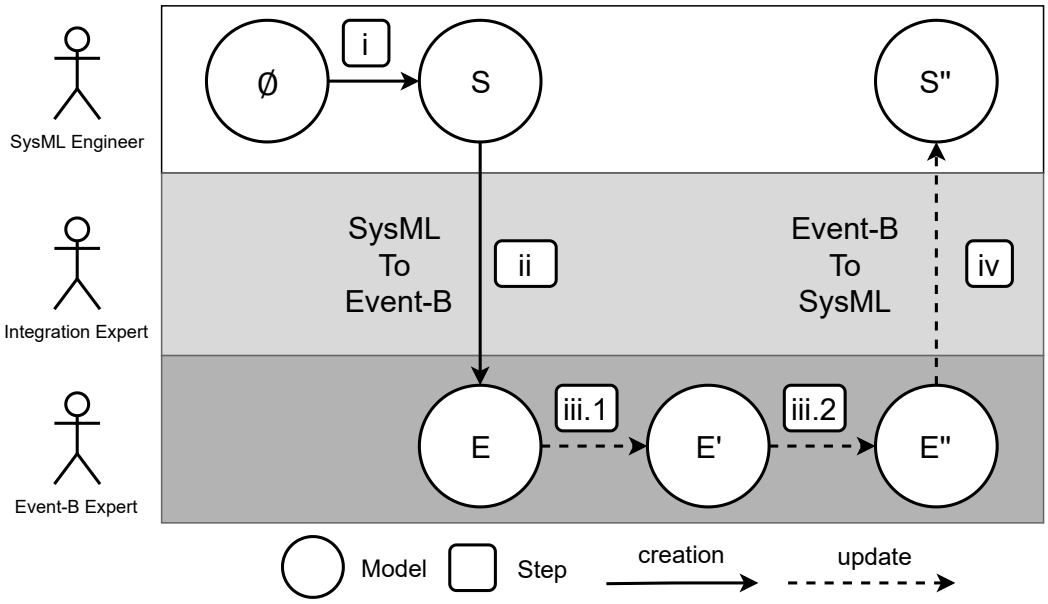


Figure 1.1: Swim-lane diagram: Model transformations in railway systems engineering

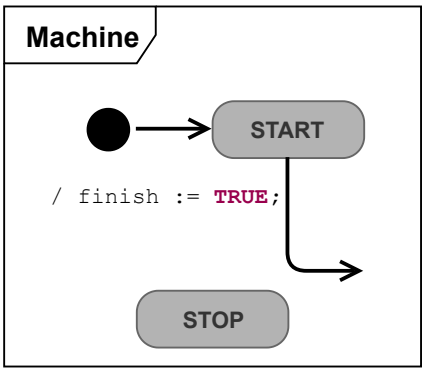
The transformation and synchronisation steps can be considered as the critical part of the development process, as the results of the formal verification might be invalidated by errors in the transformation process. As human work tends to be error-prone, it seems to be adequate to use a *transformation engine* which – as soon as its specification is proven to be correct – meets the requirements for reliable model and consistency management. All actions of the transformation engine are based on a *consistency relation* between the two languages at hand: Model pairs which are contained in this relation are considered to be consistent with each other and can be the result of a successful transformation, synchronisation or consistency check. The development of such a transformation engine, however, requires a solid formal basis and stable tools which abstract from the concrete use case to consistency management problems in general.

In principle, the described scenario of synchronising SysML models and Event-B code can be automated using existing model transformation approaches. However, these concepts and tools enforce *perfect consistency* each time the transformation engine is used, i.e., the input models must be consistent in themselves (*intra-model consistency* or *well-formedness*) and it must always be possible to create a transformation result that is consistent with the input (*inter-model consistency*). Strict consistency at any point of time is almost impossible to achieve in practical applications and is often not even a desirable goal, which makes concepts and tool support for *fault-tolerant* consistency management inevitable. In her seminal paper on fault-tolerance in MDE [Ste14], Stevens points out that realistic application scenarios always involve inconsistent aspects which makes the suitable handling of inconsistent system states necessary as well. In the following, her three arguments are applied on the concrete use case of a BX between SysML and Event-B.

Partial consistency relation

As models of heterogeneous languages must be handled, there might be well-formed SysML models, which cannot be consistently transformed into Event-B code, as the set of transformation rules does not cover the specific model instance. Likewise, there might be correct Event-B models which do not have a consistent counterpart in SysML: In Fig. 1.2b, an Event-B machine is shown that can be considered as well-formed. Its semantically equivalent SysML state machine (Fig. 1.2a), however, involves a dangling outgoing edge from the START state, which violates the syntax of the SysML language. Therefore, a non-tolerant transformation engine would reject the input in its current form.

For the user, it might be more helpful to perform a partial transformation instead that produces the depicted SysML model, and point out that there is still an open issue that needs to be fixed (i. e., the dangling edge). The fault could be subsequently removed by the user, letting the dangling edge point to the STOP state. This update could then be propagated to Event-B code, such that the variable STOP (representing the respective state) is set to TRUE in the THEN block.



(a) SysML model

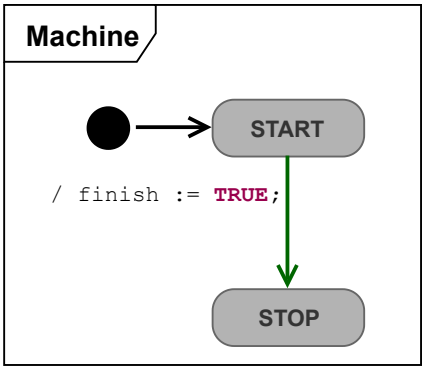
```
MACHINE Machine
VARIABLES START STOP finish
INVARIANTS ...
EVENTS
  INITIALISATION ≙
    ...
  COMPLETION ≙
    WHEN
      isin_START : START = TRUE
    THEN
      leave_START : START := FALSE;
      act1 : finish := TRUE;
    END
END
```

(b) Event-B code

Figure 1.2: Malformed SysML model and semantically equivalent Event-B code

Infeasible search

Getting results within an acceptable amount of time is an important non-functional requirement that also holds for MDE tools. When working with large models, it might be infeasible to determine a consistent result with acceptable time effort, even if one exists. Instead, it could be better to return the best solution found so far after some pre-defined time, and again refer to the problems that were left open. It is hardly possible to provide a small comprehensible example for this situation, but let us take a look at Fig. 1.3, in which the faults of Fig. 1.2 were removed in different ways in the SysML model and Event-B code. The SysML engineer connected the states START and STOP with a directed edge (highlighted with green colour in Fig. 1.3a), whereas both the transition and the STOP state were removed from the Event-B code (crossed out in Fig. 1.3b). Clearly, it is not possible to find a synchronised solution that takes both changes into account. Instead, according to Stevens, it is advisable to compute a solution that balances both interests and point out those changes that could not be considered.



(a) SysML model

```
MACHINE Machine
VARIABLES START ~~STOP~~ finish
INVARIANTS ...
EVENTS
  INITIALISATION ≙
    ...
  COMPLETION ≙
    WHEN
      isin_START : START = TRUE
    THEN
      ~~leave_START : START := FALSE~~;
      act1 : finish := TRUE;
    END
END
```

(b) Event-B code

Figure 1.3: Concurrent changes on both models result in a conflict

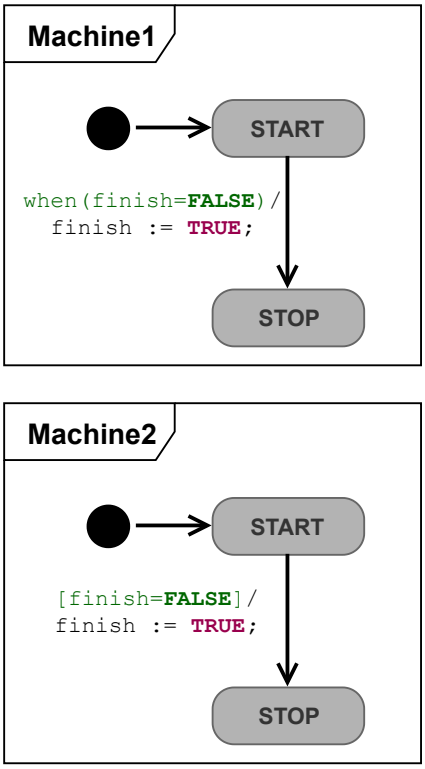
Multiple solutions

A transformation or synchronisation step might also end up in several possible solutions. Let us assume that the experts for SysML and Event-B agreed on the solution of 1.3a, i.e., there should be a transition between the states `START` and `END`. In the Event-B code, a further guard named `incomplete` is added that ensures that the transition can only be executed if the variable `finish` is still `FALSE` (cf. Fig. 1.4b). This guard can be either translated into a trigger (Machine1) or a guard (Machine2) in SysML, as shown in Fig. 1.4a. Conventional solution approaches would either return one result at random, or return all possible solutions and let the user pick one, which is not feasible for large solution sets. Instead, Stevens suggests to return a solution that incorporates all uncontroversial changes (which are not present in this example due to its minimality) and let the user include controversial modifications (the new trigger or guard), possibly at a later stage.

Research statement

It is important to note that eventually restoring consistency is still the ultimate goal: In order to create reliable and trustworthy software systems, all faults, i. e., all intra- and inter-model inconsistencies, need to be removed from the system models in their final versions. To make sure that all involved models are free of faults, consistency must be (re-)established. This is essential for the considered case of safety-critical systems, but also for software applications in general. Accepting inconsistent solutions in between should, first and foremost, make the development process more flexible and comprehensible, removing unnecessary barriers for the development team. Consequently, the development process can be sped up while achieving the same level of software quality. To be able to check and restore consistency completely, additional information should be provided to the stakeholders along with the (intermediate) inconsistent solution that help them to adapt the models in a suitable way, such that consistency can be eventually achieved. As these arguments can be transferred from this illustrating example to a broader context, there is a noteworthy need for research on fault-tolerant consistency management.

Inconsistent states in software systems have already been a research topic for several years. Balzer names tolerating inconsistencies an important challenge [Bal91] and suggests identifying and flagging inconsistencies for the user, as it is more helpful than forbidding



(a) SysML model

```
MACHINE Machine
VARIABLES START STOP finish
INVARIANTS ...
EVENTS
  INITIALISATION ≙
    ...
  COMPLETION ≙
    WHEN
      isin_START : START = TRUE
      incomplete: finish = FALSE
    THEN
      leave_START : START := FALSE;
      enter_STOP : STOP := TRUE;
      action1 : finish := TRUE;
    END
END
```

(b) Event-B code

Figure 1.4: Multiple solutions for a backward synchronisation

them completely. Similar to Stevens, Bertossi et al. point out that inconsistencies can represent more than an inferior or imperfect state of a system. They can be considered as a formal representation for misunderstanding, conflicts and problems that occur when multiple engineers work on the same software system [BHS05]. Fault-tolerance plays an important role in other fields of computer science as well. Decker intensively studied the role of fault-tolerance for database repair and integrity checks of databases [DM11,Dec17]. Cheney et al. [CGMS15] state that aiming at a consistent solution in each step tends to be a too inflexible strategy. Instead, they state that a small change in the source model should only cause a small change in the target model, possibly at the price of not restoring perfect consistency immediately, which they denote as the principle of least surprise.

Although ongoing research explores how to deal with inconsistency while still guaranteeing formal properties, there is no formal and practical concept that provides full support for consistency management in MDE in a fault-tolerant manner. That means that it is unclear how fault-tolerance can be temporarily permitted to enable flexible consistency and model management, while inconsistencies are detected and tracked at the same time to eventually restore consistency. This is problematic, as it is not possible yet to build software systems that allow concurrent engineering in a flexible and therefore resource-efficient way while guaranteeing high-quality results by suitable consistency management. In the scope of this thesis, a solution concept shall be developed that sufficiently addresses this open issue. To demonstrate its practical applicability, the conceptual solution should be supported by software tools which are stable and powerful enough to be used for industrial use cases. This research statement will be elaborated by identifying the respective stakeholders and requirements (Sect. 1.2). Afterwards, an overview of the solution proposed in this thesis is presented in Sect. 1.3.

1.2 Stakeholders and Requirements

Putting the example use case of Fig. 1.1 into a broader context, several *stakeholders* can be identified who differ in their requirements for a fault-tolerant consistency management system. In the following, the roles of all stakeholders in the overall process will be briefly discussed, including a set of *requirements* that are posed to the fault-tolerant consistency management system by each stakeholder.

- The SysML engineer and Event-B specialist are examples for **domain experts**, who must be able to model their problem domain with suitable tools. For them, it is important that the modelling process is flexible and that necessary changes due to model synchronisation are comprehensible.
- The **integration expert** is a mediator between the different domain experts: He or she is in charge of transforming models entirely (such as the initial transformation from SysML to Event-B), propagating changes from one model to another (which can, e.g., result from a formal verification of the Event-B model) and checking consistency to spot faults which were either undetected or put aside.
- The **tool developer** equips the integration expert with a suitable software tool for their work. He or she is able to define transformation rules and discuss them with the integration expert. The roles integration expert and tool developer can be taken up by the same person.
- The **meta-tool developer** is responsible for developing generic tool support, that can be adapted to the concrete problem domain by the tool developer. He or she must implement different consistency management operations on a formal basis, and offer a User Interface (UI) to specify rules and constraints.

Based on the stakeholders' expectations, the following requirements for a fault-tolerant consistency management framework with suitable tool support can be derived:

R1 In several application domains, including safety-critical systems, process certification plays an important role (cf. Fig. 1.1). Therefore, the language for expressing the consistency relation should be formal, such that, e.g., safety properties can be verified.

R2 To enable the domain experts to continue working when intra- or inter-model consistency is violated, the consistency management framework is required to be fault-tolerant.

R3 Multiple consistency management operations, such as forward and backward transformation or consistency checks, should be supported by the framework.

R4 The language for specifying the consistency relation (cf. R1) should be expressive enough to precisely map constructs of the involved modelling languages to each other while respecting additional constraints of both domains.

R5 In case of concurrent modifications on multiple models, a consistent state needs to be restored, probably requiring a compromise that balances conflicting changes.

R6 As it is too inefficient and error-prone to develop a consistency management tool for a particular use case from scratch, generic tool support is required that the tool developer can adapt and configure for the concrete application scenario.

R7 Based on a specified consistency relation (e.g., in form of transformation rules), consistency management tasks (cf. R3, R5) should be executable in a fully automated way.

R8 As the completion of these tasks should be as fast as possible, the integration expert should be equipped with an efficient tool that also scales well for larger model sizes.

R9 Especially when relating new languages to each other, or after the consistency relation was changed (e.g., the underlying rule-base was modified), it is possible that errors occur in the specification that are hard to detect without further support. Therefore, besides running the fully automated background mode (R7), an interactive mode for consistency management tasks should be offered to explore the specification and remove faults from it.

R10 The framework and tool support should be independent of the context of use and applicable to different domains. This should be underpinned by real-world case studies from industry.

Some requirements are only important for specific stakeholders, whereas other requirements concern all persons who are involved in the development process. Table 1.1 shows the relation between requirements and stakeholders, whereby a × indicates that a requirement is relevant for the respective stakeholder.

Req.	Description	Chapter(s)	Domain Expert	Integration Expert	Tool Developer	Meta-Tool Developer
R1	Formal Basis	3-8				×
R2	Fault-Tolerance	2,5-8	×	×		
R3	Multiple operations	5		×	×	×
R4	Expressiveness	4,6	×	×	×	×
R5	Concurrent Synchronisation	7,8		×		
R6	Generic Tool Support	2,9,10			×	
R7	Full Automation	5-8		×		
R8	Efficiency, Scalability	5-12		×		
R9	Interaction	10		×	×	
R10	Applicability	2,11,12			×	

Table 1.1: Mapping of requirements to stakeholders

The upcoming Sect. 1.3 is devoted to how this thesis contributes to a formal and practical framework that fulfils the stated requirements.

1.3 Solution Overview and Contribution

In order to address the requirements for a fault-tolerant consistency management system, both formal concepts and tool support for these concepts are developed throughout this thesis. It is composed of 13 chapters as depicted in Fig. 1.5. Following the general introduction and motivation for fault-tolerant consistency management in this chapter, a systematic literature review presents the state of the art in detail (Chap. 2). As an underlying formalism to address the identified research gap, Triple Graph Grammars (TGGs) [Sch94] are chosen as a well-known and established approach to rule-based model transformations in MDE. Together with a brief introduction to the two languages of the example transformation, the foundations of TGGs are introduced in Chap. 3, followed by an overview of the most important language features and their expressive power (Chap. 4).

Based on the first four chapters, the conceptual solution can be subsequently introduced. A fault-tolerant approach to model transformation and consistency checking based on TGGs and Integer Linear Programming (ILP) is introduced in Chap. 5, and extended towards domain constraint handling in Chap. 6. Domain constraints are an important feature for specifying intra-model consistency, and are therefore necessary to be considered in practical applications. The fault-tolerant framework is further extended towards concurrent model synchronisation in Chap. 7, before ILP is replaced by different meta-heuristics in Chap. 8 aiming at runtime performance improvements.

The entire conceptual solution is implemented within different components of the eMoflon tool suite[3], which are briefly presented in Chap. 9. To support user interaction for the different consistency management operations, the MDE debugger "VICToRy" (**V**isual **I**nteractive **C**onsistency Management in **To**lerant **R**ule-based Sy**s**tems) was developed (Chap. 10) and connected to eMoflon. Equipped with this tool support, it was possible to apply TGG-based consistency management to industrial case studies (Chap. 11 and 12). Finally, Chap. 13 sums up the results, and suggests directions for further research.

In the following, the main contributions of all chapters are described more concretely. Furthermore, the contributions are mapped to the requirements they mainly address (cp. Sect. 1.2). The requirement R8 (efficiency and scalability) concerns all chapters and is therefore not explicitly listed.

Systematic Literature Review on Fault-Tolerance in MDE (Chapter 2)

To determine the necessity of creating fault-tolerant frameworks, it is inevitable to study the state of the art, i.e., existing approaches to fault-tolerant consistency management. We conducted a *systematic literature review* following the guidelines presented by Kitchenham [Kit04] to get an overview of the related work. As the concept of *uncertainty* in MDE is closely related to fault-tolerance, one goal was to draw a line between these two terms with help of the literature review. The result of the review process was threefold:

- Classifications for consistency, fault-tolerance and uncertainty, wrapped up in one feature model each (R2)
- Use cases, tools and application domains for demonstrating fault-tolerance, uncertainty and flexibility (R6, R10)

[3] https://emoflon.org/

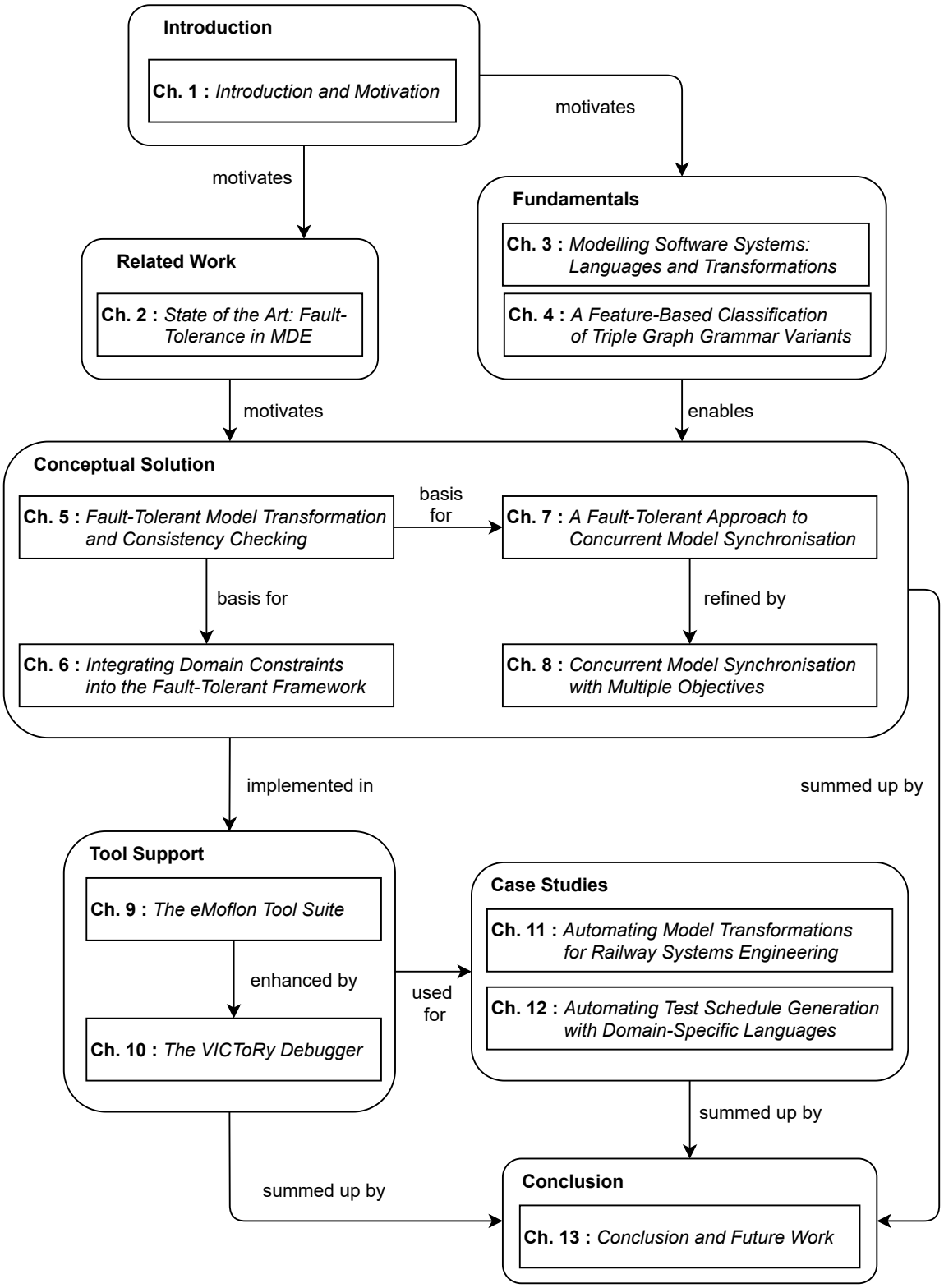


Figure 1.5: Thesis structure

- Benefits and drawbacks of involving fault-tolerance, uncertainty and flexibility in MDE concepts (R2)

Background: Introduction to TGGs and their Language Features (Chapter 3 & 4)

As a formal basis for a fault-tolerant consistency management framework, we chose TGGs, a declarative and rule-based approach to BX. The unique selling proposition of TGGs is that operational rules for multiple consistency management operations can be derived from the same declarative specification. The original TGG definition was enhanced with additional language features in recent years [ALS15] to increase expressiveness. It is necessary to consider these features for the developed framework in order to enable the integration expert to handle, e.g., domain constraints appropriately. Chapter 3 provides background information on the foundations of TGGs, before Chap. 4 introduces language features that are used in Chap. 6 and 11. The main contributions are as follows:

- An introduction to TGGs as a BX language (R1)
- A common definition of *expressiveness* for TGGs (R4)
- Formal definitions and examples for basic TGG rules, multi-amalgamation, attribute conditions and application conditions (R1)
- A comparison of TGG language features with respect to expressiveness (R4)

Formal Framework for Fault-Tolerant Consistency Management (Chapter 5)

Tolerating inconsistencies during the consistency management process makes it necessary to consider consistency as a *continuous measure* instead of a binary decision: Instead of simply reporting errors to the user in case of inconsistent inputs, it is desirable to return a solution which is *as consistent as possible*. We convert operations such as (unidirectional) model transformations and consistency checks into optimisation problems to explore the search space of possible solutions first and then choose a solution that is consistent to the largest possible extent by means of ILP solving. The approach is an extension of the consistency checking algorithm proposed by Leblebici et al. [LAS17, Leb18], whereby the following new contributions are added:

- Generalised definitions for input models, output models, and operational rules for forward and backward transformation, and for two forms of consistency checks (R1, R3)
- A generalised construction approach for the objective function and the linear constraints for the previously mentioned operations (R7)
- A brief discussion of the optimisation result in case of faulty inputs, the maximal consistent subtriple (R2)

Fault-Tolerant Handling of Domain Constraints (Chapter 6)

In Chap. 4, we have shown that application conditions - as an indirect method to ensure that domain constraints are respected - increase expressiveness and should therefore be integrated into the formal framework. The ILP-based approach gives us, however, the opportunity to specify domain constraints *directly* in form of graph constraints that must hold for the output models by leveraging the ILP with further linear constraints. In particular, this chapter contributes to the formal framework as follows:

- A uniform way of encoding matches for graph constraints as additional ILP constraints for the fault-tolerant framework (R2, R4, R7)
- A formal proof of correctness and completeness for all consistency management operations in the presence of negative and implication constraints as an extension of the proof for consistency checking without graph constraints [Leb18] (R1)

Synchronisation of Concurrent Model Updates (Chapters 7 & 8)

In recent years, concurrent model synchronisation, i. e., the propagation of concurrent updates to the respective other model, gained special interest [OPN20]. The operation is especially challenging to formalise and implement, as updates can be *conflicting*, such that it is not possible to consider all of them to form a synchronised solution. Therefore, the result is not unique but depends on the conflict resolution strategy at hand. Further aiming at fault-tolerance makes the problem even harder, but gives us the opportunity to build upon the formal framework of Chap. 5 and 6. Opposed to the previously considered "stateless" operations (also denoted as *batch transformations*), concurrent model synchronisation requires information about changes made since the last synchronisation point, denoted as *delta* structures, besides the input models. Elements can be created, deleted, or remain unchanged, which makes it necessary to balance multiple optimisation goals. Furthermore, *rule variants* are specified to reasonably cope with user edits of different shape. As it seems promising to sacrifice optimality for performance improvements, single- and multi-objective meta-heuristics are integrated into the formal framework as an alternative to the exact ILP method. The main contributions of these chapters are:

- A definition of delta structures as a distinguished subset of the input models (R1)
- A specification of rule variants that allow us to reasonably handle arbitrary user edits (R1, R2)
- A parametrised objective function that makes it possible to balance multiple optimisation goals according to user preferences (R5, R7)
- The integration of the meta-heuristics Simulated Annealing (SA), Genetic Algorithm (GA) for single-objective optimisation and Non-Dominated Sorting Genetic Algorithm II (NSGA-II) for multi-objective optimisation into the formal framework (R5)

Tool Support: eMoflon Tool Suite and VICToRy Debugger (Chapters 9 & 10)

In order to transfer formal concepts into practice, stable and mature tool support is inevitable. Therefore, the entire approach is implemented as part of eMoflon, an Eclipse-based tool suite for MDE that builds upon algebraic graph transformations and TGGs. It consists of the main components IBeX (Sect. 9.3 & 9.4) and Neo (Sect. 9.5), whereby the former is based on the Eclipse Modeling Framework (EMF) and stores models in form of Ecore-compliant XML Metadata Interchange (XMI) files, whereas the latter uses the graph database Neo4j for this purpose. To further support user involvement in the consistency management process, the VICToRy debugger (Chap. 10) allows us to step-wise execute rule applications to both understand the background process and detect errors in either the TGG specification or the input models. The main contributions are as follows:

- An overview of the front-end of the eMoflon tool suite components (R6) and the VICToRy debugger (R9)

- UML component and activity diagrams that describe the software architecture and overall work-flow of IBeX, Neo and VICToRy (R6, R9)

Use Cases for Fault-Tolerant Consistency Management (Chapters 11 & 12)

Finally, the applicability of the developed concepts in practice is demonstrated with help of two industrial case studies. The first case study at DB Netz AG deals with a BX between defined subsets of the semi-formal language SysML and the formal language Event-B, which was already used to introduce and motivate the topic of this thesis. eMoflon was used to perform TGG-based forward transformations and backward synchronisations, while the other operations seem to be adequate for future use as well. At dSPACE GmbH, a software and hardware developer for mechatronic control systems, the manual process of allocating software testers to work packages was automated by creating (meta-)models for human resources on the one hand and for testing tasks on the other hand, and establishing a consistency relation between these two models. For the scope of a product release test of 6-8 weeks, an optimal allocation of software testers and testing tasks was computed using eMoflon. Although only a few features of the fault-tolerant framework are necessary to solve the particular use cases, they serve as a general motivation to use BX concepts and tools for real-world applications. The following contributions are presented:

- Different examples for using consistency management operations in the context of resource allocation and verification and validation of safety-critical systems (R10)
- Realistic examples for modelling consistency relations with TGGs (R10), making use of several TGG language features (cf. Chap. 4)
- Exemplary ways of integrating eMoflon with existing tools at partner companies (R10)

1.4 Publication Overview

This thesis contributes novel concepts, tools and case studies of fault-tolerant consistency management in MDE. Research results on different sub-topics have already been published in form of conference, workshop and journal articles before. Table 1.2 provides an overview of all publications which this thesis is based on, their publication venue, and to which chapter they mainly contribute. An overview of further co-authored publications, which are related to consistency management in MDE but are not explicitly discussed in the scope of this thesis, is given in Tab. 1.3.

Furthermore, several Bachelor and Master theses contributed additional content, especially in the areas of tool support and case studies. In the thesis of Kannan [Kan20], a first version of the systematic literature review was presented, Verma [Ver20] took part in the integration of graph constraints into the formal framework. Regarding tool support, Robrecht [Rob18] described major parts of eMoflon::IBeX-GT in his thesis, Jose [Jos21] and Srivastava [Sri21] extended the VICToRy debugger towards breakpoints and a prototype for concurrent synchronisation, respectively. The case studies for test schedule generation and the transformation between SysML and Event-B are based on the theses of Salunkhe [Sal20] and Oppermann [Opp18]. Finally, numerous Bachelor and Master theses at Paderborn University have contributed to the development of eMoflon::IBeX and eMoflon::Neo between 2016 and 2020.

Publication	Venue	Topic	Ch.
[Wei18]	MODELS 2018	Motivation for fault-tolerance in MDE	1
[WKA21]	CoRR 2021	Systematic literature review	2
[WOR19]	SLE 2019	Language features of triple graph grammars	4
[WALS19]	SLE 2019	Model transformation and consistency checking	5
[WA20]	FASE 2020	Domain constraints for consistency checks	6
[WA21b]	FAoC 2021	Domain constraints for model transformations	6
[WFA20]	SLE 2020	Concurrent synchronisation with exact methods	7
[WE21]	GECCO 2021	Concurrent synchronisation with meta-heuristics	8
[WARV19]	ICGT 2019	Graph transformations with eMoflon::IBeX-GT	9
[WAF+19]	BX 2019	Model transformations with eMoflon::IBeX-TGG	9
[WA21a]	BX 2021	Model transformations with eMoflon::Neo	9
[WAC20]	GCM 2020	Debugging model transformations with VICToRy	10
[WSA+21]	JOT 2021	Case study: Railway systems engineering	11
[AWO+20]	MODELS 2020	Case study: Test schedule generation	12

Table 1.2: Overview of underlying publications

Publication	Venue	Topic
[WASK19]	ICGT 2019	Foundations of pattern invocation networks
[JWY+20]	ICSMM 2020	Model-driven test case migration
[WS20]	WSRE 2020	BX applications in industrial contexts
[BYWE21]	HCSE 2021	User interface adaptations with TGGs
[YGWE21]	MODELS 2021	Collaborative software modelling

Table 1.3: Overview of other related publications

2 State of the Art: Fault-Tolerance in MDE

Although the challenge of tolerating inconsistencies has been discussed from different viewpoints within and outside the modelling community, there exists no structured overview of existing and current work in this regard. We provide such an overview to detect the unsolved problems of tolerating inconsistencies in MDE and thereby motivating the TGG-based approach which is proposed in the remainder of this thesis. We follow the standard process of a Systematic Literature Review (SLR) to point out what fault-tolerance means, how it relates to uncertainty, which use cases for fault-tolerant software systems have already been discussed, and which benefits and drawbacks tolerating inconsistencies entails.

The scope of this SLR is twofold: First, existing approaches to implementing fault-tolerance in MDE contexts should be identified and compared to the requirements listed in Sect. 1.2. Second, the motivation for studying the topic of fault-tolerant consistency management in MDE shall be put on a broader basis by gathering exemplary use cases and collecting arguments for and against fault-tolerant concepts. While several sources that were found in the course of the SLR address the second point, we did not find a holistic approach for adding fault-tolerance to multiple consistency management tasks. A substantial amount of related work has indeed been presented for single aspects of the required framework, which will be discussed in the respective sections of Chap. 5 – 10.

This chapter is structured as follows: The conducted SLR is motivated in Sect. 2.1, including a list of research questions that are answered in the course of this chapter, before Sect. 2.2 provides an overview of similar studies that apply the SLR procedure on MDE topics. Section 2.3 describes the survey procedure and briefly sketches how the SLR was supported by MDE tooling. In order to address the research questions, classifications for the key terms consistency, fault-tolerance, and uncertainty are provided in Sect. 2.4. A collection of exemplary use cases that are used to demonstrate these concepts is briefly presented in Sect. 2.5, before a summary of benefits and drawbacks is presented in Sect. 2.6. While the results are described verbally and with help of feature diagrams at this point, a detailed tabular overview is available online[1]. Meta-data of the collected papers are analysed and interpreted in Sect. 2.7, before the proposed solution approach of this thesis is introduced in Sect. 2.8. Finally, the results are summarised and discussed in Sect. 2.9 to motivate further research on fault-tolerance in MDE based on the findings of the SLR.

2.1 Motivation

Up until now, research on consistency management in MDE has primarily focussed on preserving or restoring perfect consistency between the involved models. As stated in Sect. 1.1, there is an apparent need for temporarily admitting inconsistencies in the software development process, even though it is uncontroversial that consistency shall be fully restored at the end of the process. The existence of a partial consistency relation, the non-trivial choice between multiple consistent solutions and the time effort to find a perfectly consistent solution were named by Stevens [Ste14] as situations in which MDE tools could reasonably return inconsistent results. Closely related to fault-tolerance, but

[1] https://drive.google.com/file/d/1uSuOn3hX5BHpLhw3jaH2ZpVffgTxHOov

yet significantly different, is the concept of modelling with uncertainty [FSC12a]. It addresses a common problem in using modelling techniques in practice: At early stages of the development process, requirements are often too vague to model the system precisely, which is why a range of possible solutions is encoded into a single uncertain model to postpone the final design decisions. Modelling uncertainty is also a suitable technique to model information that will likely change during the development process. As approaches proposing support for fault-tolerance and uncertainty tackle similar problems, i. e., aim at increasing the practicability of MDE techniques, it is difficult to draw the line between such approaches, which shall be addressed in the scope of this SLR.

Although a substantial amount of work with respect to fault-tolerance has already been done in software engineering research and related domains [Bal91, Egy06, Ste14, GdL18], there does not yet exist a structured overview of existing approaches. Such an overview facilitates further research - both in the scope of this thesis and in subsequent work - for several reasons: First of all, it helps to collect and aggregate achieved results, such as common definitions for fault-tolerance, or benefits and drawbacks which several authors already agreed on. Second, the SLR could help to discover examples from other subfields of computer science that can be easily transferred to MDE. In this way, different communities could learn from each other, and cross-discipline collaboration could be strengthened addressing the same problem area. In the database community, for example, there has been long-term research on maintaining and restoring consistency, and performing operations in the presence of errors [DM11, Dec17]. Consequently, to avoid reinventing the wheel, the SLR will consider research from related fields, as long as key terms have the same meaning and transferring results to MDE is possible.

To achieve these goals, first, an overview of existing definitions and classifications of the terms *consistency*, *fault-tolerance* and *uncertainty* shall be provided. Second, to underpin the applicability of these concepts, formal and practical frameworks should be identified that support handling of fault-tolerance, uncertainty, or flexibility (which subsumes the first two terms) in software modelling. A range of realistic use cases and examples is essential, which can be used in future approaches to facilitate comparison to existing work. Third, a critical discussion of taking fault-tolerance, uncertainty, or flexibility into account when modelling software systems should also be included to identify opportunities and risks that are inherent to fault-tolerant approaches. In particular, we aim at answering the following research questions:

RQ1 Scope and Classification: How is (in)consistency defined? How do fault-tolerance and uncertainty differ? What makes an approach or tool fault-tolerant? Which different dimensions of fault-tolerance are there and how can they be classified?

RQ2 Use Cases and Application Domains: In which application domains are fault-tolerance, uncertainty and flexibility discussed? Which use cases and tools are there to demonstrate ideas and results?

RQ3 Benefits and Challenges: What are the benefits of fault-tolerance, uncertainty and flexibility? What are open questions and challenges?

As a second contribution, we propose a framework that supported us while conducting this SLR and which can be reused for creating future literature reviews in computer science. In consensus with previous findings [Göt18], we noticed that SLRs in general are conducted with little or no tool support (or at least lack a respective description), although they involve numerous steps that could be automated, leading to unnecessary manual effort. Likewise, it also requires a substantial amount of work to reproduce the

results of an SLR, leading to opacity of findings due to time restrictions. The gathering and reproduction of results should thus be eased to better utilise human resources for tasks that require advanced knowledge of the problem domain. We propose a tool chain for partly automating the review process, which involves an adapter for querying the DBLP research database[2], a transformation of the results to the eMoflon Specification Language (eMSL). eMSL is the uniform specification language of the model management tool eMoflon::Neo, which will be described in more detail in Sect. 9.5. The collected data is exported to the graph database Neo4j[3], which can be queried to analyse the results.

2.2 Related Literature Reviews and Mapping Studies

Studies that provide a structured overview of existing work on a particular topic are often conducted as SLRs [Kit04] or as mapping studies [PFMM08]. SLRs are a secondary study that identifies, analyses and interprets all available information related to one or more research questions. SLRs follow a predefined review protocol, such that the process of retrieving results is transparent and the introduced bias is minimised. Mapping studies categorise existing work, often leading to a visual mapping of categories that supports the understanding of what is already addressed in a specific domain.

Several studies of either type have been conducted in the MDE domain and related fields. Modelling languages were investigated with a focus on the SysML language [WMC+20], the Query/View/Transformation-Operational (QVT-O) standard [GSS14,GSS16], and the application of modelling in Industry 4.0 [WBCW20,WCB17]. Further MDE-related work investigates literature on models at runtime [SZ16], software testing process models [VDT18], articles that appeared in the Journal of Software and Systems Modelling [GR16], and quality in MDE [GAM16]. In the requirements engineering domain, studies on software tooling for requirement elicitation [IKJ19] and software testing in the context of agile software development [CdM19] have been presented. For software product lines, existing work on the automated analysis of feature models [GBT+19], variability management [GWT+14], and tool support [BGR+17] has been already investigated. Context modelling [KHJS14] and environment modelling [ST15] are further topics of existing SLRs. In contrast to this chapter's SLR, these studies focus on special modelling languages or particular application domains.

The notion of consistency in modelling languages has also been a topic of multiple SLRs and mapping studies. Awadid and Nurcan [AN19, AN16] composed an overview of consistency requirements of business process models by proposing a framework for the categorization of approaches and a road-map for future research on consistency requirements elicitation and management. The work of Muram et al. [MTZ17] takes consistency checking of software behavioural models into account. Seven main categories for consistency checking in this domain were identified, and suggestions for future research in this direction were proposed. Hoisl and Sobernig [HS15] conducted a literature review on consistency rules for UML-based language models, discussing frequently-named defects of such models and demanding more tool support for enforcing consistency rules in this setting. All of these studies focus on a sub-domain of MDE and do not take fault-tolerance or uncertainty into account.

Only two studies on existing work relating fault-tolerance to software engineering problems could be found. Nascimento et al. [NRB+14] analysed literature on the design of a fault-tolerant Service-Oriented Architectures (SOAs) using design diversity, deriving

[2] https://dblp.uni-trier.de/
[3] https://neo4j.com/

guidelines for a fault-tolerant SOA design and proposing a taxonomy for useful techniques in this respect. A mapping study for fault-tolerant Internet of Things (IoT) applications [MM19] identifies key factors for fault-tolerant systems, including the use of microservices and the distribution of IoT components. Both studies are neither directly related to MDE nor address the problem of maintaining consistency.

In a study combining an SLR, semi-structured interviews, and an empirical evaluation, Marinho et al. [MSdM18, MdddM15] propose and evaluate techniques to distinguish risks and uncertainties to reduce the latter in software projects. Salih et al. [SOY17] provide an overview of existing work on uncertainty involved in requirements engineering via a categorisation of relevant sources, while several questions are left open. Measurement uncertainty was studied by da Silva Hack et al. [dtC12], resulting in a classification of approaches and a list of methods for calculating uncertainty. However, these treatments focus solely on uncertainty, whereas fault-tolerance and software modelling are not considered.

As previously mentioned, SLRs in computer science often lack adequate tool support; this issue has been identified and discussed by existing work. Götz proposes a tool for processing the findings of SLRs [Göt18], which enables the user to assign the relevant papers to formed categories, such that diagrams can be generated that visualize the characteristic values for one or two categories. The tool supports SLRs in a later phase, though, as the list of relevant sources is required as input data. The SLR-Tool by Fernández-Sáez et al. [FSBR10] supports the process of conducting SLRs in different phases. Relevant metadata can be stored for each source, a classification scheme can be created, and diagrams for result visualisation can be exported. In contrast to our tool chain, the sources have to be imported manually first, and tool-support for the snowballing step is not provided.

In total, to the best of our knowledge, there does not exist a survey that provides an overview of existing work on fault-tolerant consistency management in MDE. Also, existing tool-support for SLRs eases the maintenance of relevant sources, but does not aim at reducing the manual effort in initial phases of conducting SLRs.

2.3 Survey Procedure

This section briefly presents the methodology we followed to conduct the SLR. As all results should be reproducible and easily accessible for researchers of the modelling community, we therefore followed the guidelines proposed by Kitchenham et al. [KBB+09] for literature reviews in the software engineering domain. The review was conducted from October 2019 to September 2020 and considers sources published until June 2020. We used DBLP as a research database due to its large amount of listed publications, its focus on computer science, and its well-described API for automated queries.[4] Following the proposed guidelines, an initial and a final set of sources was determined by applying search phrases, and criteria for inclusion, exclusion, and quality of the gathered sources, described in the following.

To form an **initial set of sources**, we defined six search strings inspired by the research questions and the domain MDE, of which at least two must appear in a title. Each of these strings has a wildcard ($\star$) as suffix to take nouns, verbs and adjectives into account. We therefore decided to query the database with all pairs formed from the search strings `model*`, `consisten*`, `inconsisten*`, `uncertain*`, `tolera*` and `flexib*`. As a combined inclusion and quality criterion, we further require the respective sources to be published at a conference listed by the CORE ranking[5] and assigned to the research

[4] https://dblp.uni-trier.de/faq/13501473.html
[5] http://portal.core.edu.au/conf-ranks/

field 0803 (Software Engineering). Due to this requirement, the search is focussed on the software engineering domain and peer-reviewed publications, while journal and workshop papers are initially excluded. Due to the publishing behaviour in the computer science domain, we expect late-breaking research results to be published at conferences, whereas journal articles usually extend previously published results of conference papers in more depth. Furthermore, the list of journals at CORE[6] was outdated when the review was conducted, making it difficult to apply equal criteria to journal and conference papers in the initial search step. Workshop papers often present work in progress and initial ideas to be published at conferences afterwards. To detect relevant papers which do not fulfil all criteria of the initial search, we applied snowballing at a subsequent step. In this manner, we retrieved 268 sources, which we denote in the following as *core* papers.

To compile a **final set of sources**, we distributed the core papers between three researchers and assessed their relevance based on the abstract. In case it remained unclear if the respective paper should be considered, introduction and conclusion were read as well. As suitable examples demonstrating the use of fault-tolerant system behaviour are essential to answer RQ2, the remaining parts of the papers were skimmed for such examples. The assessment was based on inclusion, exclusion, and quality criteria. A paper was included in the further review process if it (1) presents an MDE or Programming Languages (PL) approach related to (in)consistency management, or (2) if it contains an example or application related to fault-tolerance or uncertainty. We excluded a paper if any of the search terms has a different meaning than the one implied by the research questions, e.g. if model refers to the physical behaviour of a Cyber-Physical System (CPS). Papers written in other languages than English were also excluded, while this criterion never had to be applied, probably due to the choice of search strings. To ensure a high quality of the selected sources, prefaces and extended abstracts were excluded. In total, 114 *relevant core papers* were identified and added to the final set of sources.

In a second iteration, we applied **snowballing** to consider papers that were not detected in our first iteration but might be nonetheless relevant to answer the research questions. The corpus of this SLR was extended by all sources which are cited by at least one of the core papers, resulting in 3201 additional papers. To keep the number of paper for the second assessment phase manageable, we added a further inclusion criterion for these additional papers: a minimum citation count by relevant core papers, as papers cited more frequently are more likely to be relevant. As newer sources naturally have a lower citation count, the number of required citations was set in relation to the publication year. As papers published at Software Engineering (SE) venues should be preferred, the minimum citation count per year was set to 0.2 for SE papers and to 0.3 for all other papers. As a result, 53 papers from SE venues and 41 papers from other venues were evaluated according to the same inclusion, exclusion, and quality criteria as the core papers. After the second assessment phase, 23 papers from SE venues and 20 other sources were added, increasing the final set of sources to 157 papers. An overview of our assessment and selection process is depicted in Tab. 2.1. For each property, a check (✓) means that it is fulfilled by the respective category of papers, whereas a cross (✗) means the opposite. If the property is irrelevant, this is indicated by a hyphen (-). The rightmost two columns contain the number of papers per category identified as (not) relevant.

Although the survey procedure is well-defined and takes multiple objective measures into account in order to make results transparent and reproducible, several **threats to validity** have to be mentioned as well. The naming conventions for conferences in the DBLP database and in the CORE ranking differ slightly. To overcome this problem,

[6] http://portal.core.edu.au/jnl-ranks/

≥ 2 keywords in title?	published at an SE venue?	min. annual citation count	initially read?	relevant	not relevant
✓	✓	-	✓	114	154
✗	✓	≥ 0.2	✓	23	30
✗	✓	< 0.2	✗	0	1093
-	✗	≥ 0.3	✓	20	21
-	✗	< 0.3	✗	0	2014

Table 2.1: Categorization of papers for the review

we matched the respective conferences if their acronym is the same, or if one name is a substring of the other. This method works well for venues listed in the latest version of the CORE ranking but, as conferences are renamed over time, older venues might not be identified as SE venues. While prefaces and extended abstracts were excluded, no distinction was made between different paper categories, such as full, short or tool paper. Even though late-breaking results are usually published as conference papers in the computer science domain, we might have missed results only published in journal papers, as those were not considered in the first assessment phase. Besides DBLP, the use of other research databases - such as Google scholar - could have helped to gather more sources for the SLR and to therefore minimise the risk of missing important work. Finally, although inclusion, exclusion and quality criteria were discussed between the involved researchers in detail before the review was conducted, only one researcher per paper evaluated its relevance, which might lead to biased results as the assessment of relevance depends on a single person.

In Fig. 2.1, the metamodel for the representation of results is depicted. A `Paper` is written by `Authors` and appears at a `Venue`. For each `Paper`, the title and the year of publication are extracted. Boolean values indicate whether the paper is a core paper, and whether it was identified as relevant for the SLR in the initial reading phase. A `cites` relation defines which `Paper` references which other `Papers`. For the `Author`, only the name is stored, whereas the `Venue` is additionally flagged as being an SE conference listed at DBLP or not. In Fig. 2.2, the architecture for transferring the bibliography data records to a graph database is depicted. The DBLP research database provides an interface to query records that match a specified search string containing logical connectors and wild cards. Additionally, we exported the list of venues assigned to the research field 0803 (SE) as a Comma-Separated Values (CSV) file. For each possible pair of the six keywords, the database was queried and its venue was compared to the list of SE venues. In this way, the core papers for this SLR are identified and saved as models typed over the metamodel of Fig. 2.1. After examining each of the core papers regarding its relevance for the research questions, the respective attribute was manually added in the bibliography model. In the last step, the files were exported from eMoflon::Neo to Neo4j (cf. Sect. 2.1), such that queries on the bibliography model can be used to analyse the collected data.

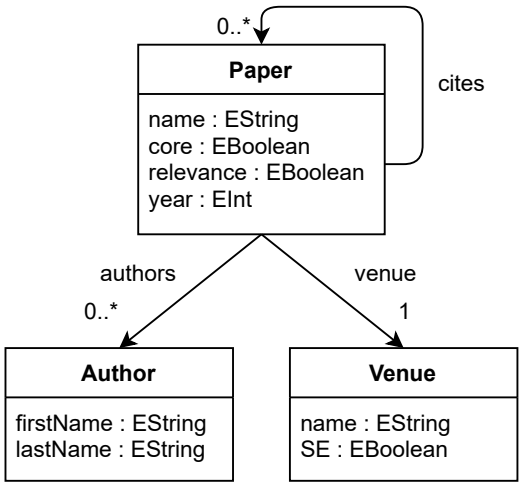


Figure 2.1: SLR metamodel

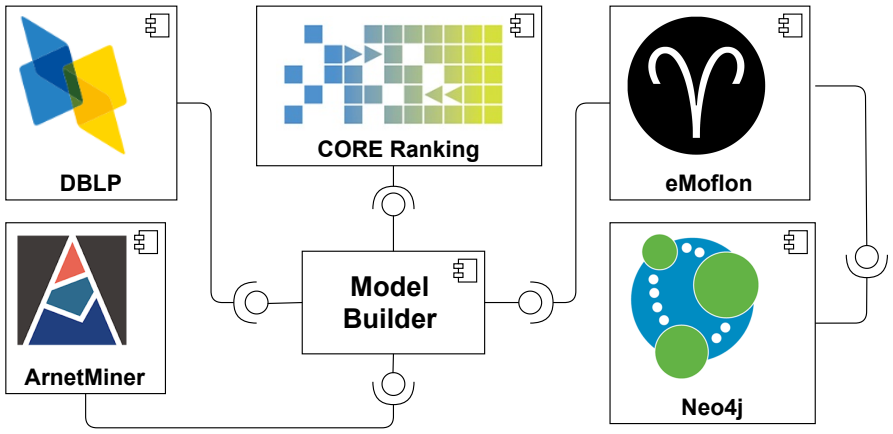


Figure 2.2: Component diagram: Tool chain

An important argument for the use of a database representation for the SLR was the snowballing step, which is – depending on the number of relevant papers – a time-consuming and error-prone task. To integrate citation information into the database, all papers referenced by the core papers were added via the database snapshot *DBLP-Citation-network V12*[7] which was created with the tool ArnetMiner [TZY+08]. Having extended the bibliography model with all papers cited from the initial set of sources, it is possible to collect all papers that fulfil the condition for being added within the snowballing step with a single database query.

2.4 Scope and Classification

In order to answer the first research question, definitions for the key terms of this SLR were gathered and aggregated while analysing the relevant sources. Besides a brief textual summary, feature models for consistency (Fig. 2.3), fault-tolerance (Fig. 2.4) and uncertainty (Fig. 2.5) provide an overview of the different dimensions involved. In accordance with the textual description, they represent and classify the existing literature related to the three aforementioned terms.

Consistency

In general, consistency can be understood as a relation over sets of models, which can be specified in different ways [Ste14, SNEC08, HEC+14, Ste18b, Ste17, Ste18a, CRE+10]. Most frequently, consistency is specified by a provided set of constraints [KKE18, VGH+12, RE12a, SZ04, SNL+07, LTZ12, Dec11, MSD06, BMMM08, EDG+11, Egy07a, KKD+17, ELF08, Red11, LAS17, KGV05, DMV+17, GdL18, JKT16, SZ06, LTZ13, Bal91, REDE14, Egy11, NEF03, ZCM+16, Egy06, NER01, XHZ+09, NCEF01, BZJ19, BSV20, KKE19, GHHS15, HHR+11, RE13], which can be formulated in different ways. Often, the Object Constraint Language (OCL) is chosen for defining consistency, but also graph constraints and logical constraints are commonly used, such as formulae of propositional or first-order logic, or Satisfiability Modulo Theories (SMT). Independent of the language in use, a model (or a proposed solution) is typically viewed as being consistent if it satisfies all constraints.

Consistency can also be defined constructively via a given model transformation T, such that two models A and B are consistent if and only if $A = T(B)$ [WGP09, MC13].

[7] https://www.aminer.org/citation

Furthermore, multiple model transformations (e.g., syntactic changes) are often considered to be consistent if they implement the same underlying transformation (e.g., semantic change) [KEK+15]. This is especially relevant in the context of co-evolution, where multiple interrelated transformations are concurrently conducted. There are also constructive definitions which define methodological consistency over the sequence of operations required to construct a model. As long as certain steps are followed in the construction or "design" process, the consistency of the resulting model(s) can be guaranteed [BMMM08, RE12b]. For example, a name must be assigned immediately after an element is created.

Besides these general consistency specifications, some definitions are also tailored to a specific application area. When modelling CPSs, a model is said to be consistent with the real world – or any other system for which this can be checked – if the model makes statements or reaches conclusions that are actually true [KCT+15]. For goal-oriented modelling, "plan consistency" means that the achievement of sub-goals implies the achievement of their parent goal [Fri18]. In the application domain of software product line engineering, a feature model is consistent if at least one valid configuration exists [BM14]. Many other application domains are conceivable as well.

The scope of consistency involves both the intra- and inter-model case. For inter-model consistency, multiple models are consistent if they are not in conflict regarding their overlapping parts, i. e., the same information contained in multiple models [CST12, EEEP08, NGTS10, FGdL12]. Another important definition deals with the relationship between a model and its metamodel [PBBT09, RKPP09, MKKJ10, TBSA11, KR07, GdL18, HTJ92, HS17, Hil16, SKE+14, DKEM16, SB16, SMSJ03, BZJ19, BSV20, KKE19, GHHS15, CK13, RE13, HHR+11]. This notion can be handled on two levels: (i) structural consistency includes multiplicities, composition constraints, as well as the types of model elements, and (ii) static semantics expressed, e. g., via OCL constraints. Finally, a set of constraints is often denoted as consistent if there exists at least one solution (e. g., a variable assignment) that satisfies all constraints [WXH+10, WC09, GKH09, HTJ92, SMSJ03].

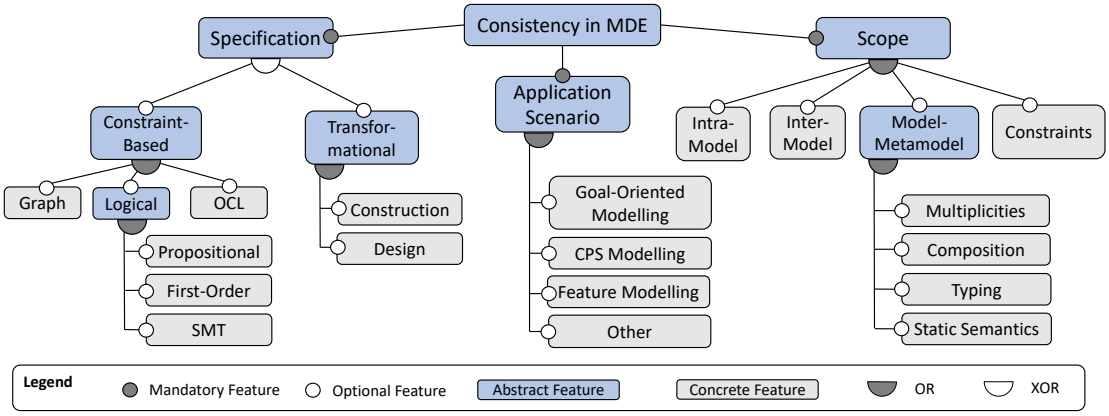


Figure 2.3: Consistency in MDE

Fault-Tolerance

Based on the collected definitions of consistency, the term fault-tolerance can be specified more concretely. Building on the constraint-based definition of consistency, tolerance can be implemented by weakening the requirement of constraint satisfaction [WXH+10, PBBT09, BM14, PST+97, RE12b, LAS17, DMV+17, VPM90]. Solutions that satisfy more

important constraints are then "better", i.e., more consistent than other solutions that might satisfy more but less important constraints. Fault-tolerance here is, therefore, basically a ranking or weighting of constraints and a score for solutions based on how many weighted constraints are satisfied. This prioritisation and sorting process is often referred to as "relaxation"; the constraints are sometimes denoted as "soft constraints" as their violation no longer directly implies exclusion of the respective solution. By defining different classes of inconsistencies and measuring consistency as vectors over these dimensions, one can obtain a more fine-grained view of the extent to which consistency is achieved or improved with respect to each class [YV00, KPP08a]. A similar approach divides the constraints or requirements into primary and non-primary, whereby non-primary constraints can be ignored in a fault-tolerant scenario [HDH10, Red11]. The idea is to filter inconsistencies that are "irrelevant", e.g., concerning white space or time stamps, layout, etc.

Another important group of strategies for implementing fault-tolerance involve temporal aspects. Most approaches assume that inconsistencies can be tolerated up to some point in time when consistency is restored, such that fixes are delayed up to this point [HDH10, YV00]. For distributed systems, a variable threshold for inconsistencies is defined by a temporal window such that more inconsistencies are accepted at the beginning and fewer towards the end. These approaches aim at letting a system "stabilise" before demanding a high level of consistency.

In contrast to temporal strategies, a further group takes a spatial approach to implementing fault-tolerance, i. e., guarantees that consistency improves with a limited extent: Case-based restoration guarantees that every part of the model that was consistent before is still consistent afterwards [Ste14, DM08, Dec11, Ste17, Egy06, Bal91]. For efficiency reasons, only a subset of "relevant" cases can be determined, i. e., a scope of influence is computed for changes, and then checked as for case-based restorers. Measure-based restorers guarantee that a chosen measure of consistency is not reduced by the restoration process.

A frequently named property of fault-tolerant systems is that strategies are implemented to detect and fix inconsistencies either automatically or by involving the user [HM05, ELF08, GdL18, BG07, EC07, Egy07b, NCEF01, Egy11, VPM90]. Even if a system is brought into an inconsistent state, it can transition back to a consistent state by applying fix strategies. Consequently, an inconsistent state is temporarily acceptable, making it unnecessary to check if edits are consistency-preserving, or to propagate changes to other models immediately. However, it is often important to keep track of inconsistent model parts as this can speed up the consistency restoration later. In case of user involvement, this can also help to avoid overwhelming users with too many design decisions.

Overall, fault-tolerant concepts can serve different purposes: They can help easing the work-flow for modelling tasks by not enforcing an immediate resolution of inconsistencies. Additionally, they can function as a mechanism to detect unresolved conflicts in the real world, or to rethink prematurely made design decisions [NER01]. Finally, tolerating and highlighting inconsistencies can be used to indicate misunderstandings or a potential disagreement of the involved developers [XHZ+09].

Uncertainty

To help make a distinction between fault-tolerance and uncertainty, we provide an overview of the most common notions of uncertainty in the following.

Modelling with uncertainty often involves encoding a range of possible values or alternatives into a single attribute value or part of a model [GHYZ11, BEP+17, BCC+16, FBDD+15, EPR15, FS13, BG07, FSC12a, PK19, HT99, MVD08]. Additionally, the set of

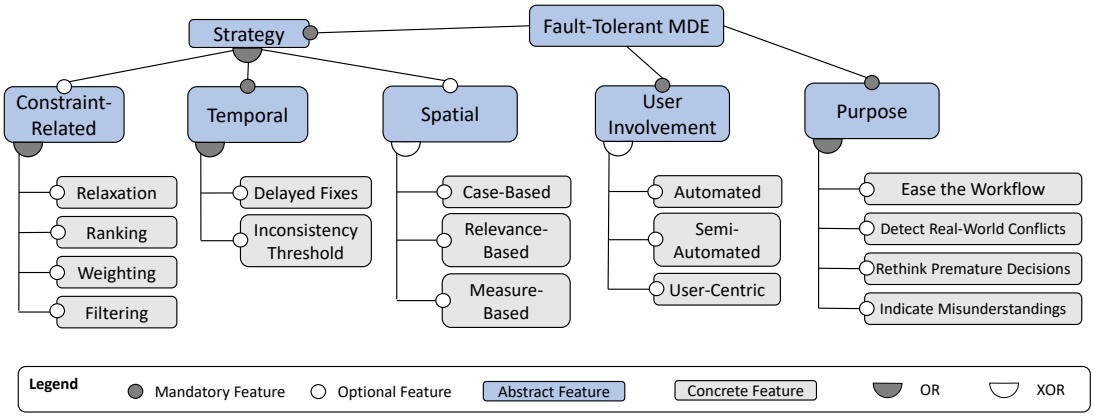


Figure 2.4: Fault-tolerant MDE

valid combinations of these parts must also be defined, which can possibly increase or reduce the range of valid alternatives. As a result, the designer is provided with a compact but expressive representation of all solution candidates. Furthermore, modelling with uncertainty can mean a probabilistic extension of a normal model made by adding probabilities to every assumed value, which are mostly attribute values [KCT+15, VMO16, CSBW09, MWV16, TM14]. These probabilities are often referred to as confidence values.

Uncertainty can be used in different phases of the modelling process, and there are multiple strategies to eventually resolve uncertainty [FS10, FS13, CBGS18, HT99, BLC18, FSSC13, Gar10, CGSB17, IFED09]. The lack of information about the content of models is denoted as design-time uncertainty, which makes it impossible to select among alternative design decisions. This uncertainty can be captured in partial models consisting of a "base model" enriched with annotations that express the set of alternatives [SCH12, SFC12, FSC12b]. By refining the partial model, uncertainties can be resolved during the design phase [SCG12]. The residual uncertainty is denoted as run-time uncertainty and is resolved by the user via a selection out of all remaining alternatives.

Different sources of uncertainty can be distinguished with respect to multiple dimensions [CGSB17, GC08, GPST13, RCBS12, SPV20, EM10, ZAY+19, ZAYN17]. The source of uncertainty can either be the system itself or its execution environment. System uncertainty includes uncertainty about input parameters, structural and algorithmic uncertainty due to approximations, or experimental uncertainty caused by variable measured values. Environmental uncertainty can originate from incomplete information about the behaviour of external components, which are provided by third-party organisations, or input data provided by sensors or wireless networks. Furthermore, the root cause of uncertainty can either be the lack of knowledge about one of the aforementioned factors or some non-determinism within the system. In general, as uncertainty forces the developer to make decisions based on assumptions, one is not able to guarantee the optimality of those decisions, involving various trade-offs.

In summary, modelling with uncertainty denotes a way of efficiently encoding multiple alternatives into a single representation, from which at least one valid, i. e., consistent configuration should be derivable. Fault-tolerance, in contrast, means being able to perform operations on models in the presence of inconsistencies, while the ultimate goal is still to eventually reach a consistent state. Both concepts aim at facilitating the work-flow of system designers and developers by aligning the principles of MDE to practical requirements. Likewise, both fault-tolerance and uncertainty involve a combination of automated and user-centric resolution strategies to finally obtain an unambiguous and consistent model.

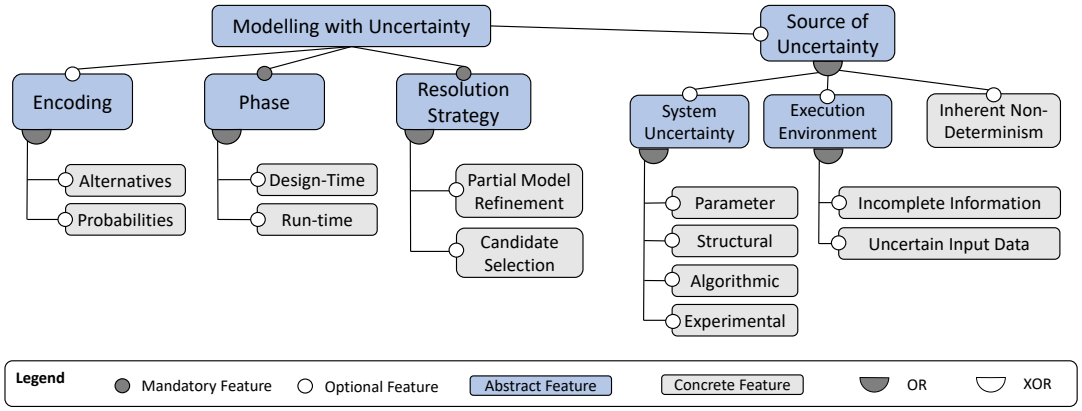


Figure 2.5: Modelling with uncertainty

2.5 Use Cases and Application Domains

The key terms *consistency*, *fault-tolerance* and *uncertainty* were classified in the previous section, addressing research question RQ1, and preparing a foundation for further analyses. This section gives an overview of existing use cases related to fault-tolerance and uncertainty in different application fields. The notion of consistency is essential for understanding the proposed approaches, but dedicated use cases and a discussion of (dis-) advantages of consistency management is out of scope for this thesis. Instead, the study of existing literature has shown that *flexibility* is frequently used as a general term subsuming fault-tolerance and uncertainty, such that we devote the third subsection to flexibility concepts in MDE.

Research Domain	#	Research Domain	#
Aspect-oriented modelling	3	Process Modelling	4
(Meta-)model Co-Evolution	4	Product Line Engineering	7
Cyber-Physical Systems	7	Requirements Engineering	9
Databases	5	Service-Oriented Computing	3
Distributed Systems	4	Smart & Adaptive Systems	10
Language Engineering	4	Software Architecture	4
Mobile & Cloud Computing	3	Software Engineering (Other)	5
Model-Based Testing	7	Software Verification	4
Model-Driven Engineering	74	**TOTAL**	**157**

Table 2.2: Number of relevant papers per research domain

In Tab. 2.2, the number of relevant sources per research domain is depicted. Most of the sources are related to MDE and similar fields, such as co-evolution, model-based testing, process modelling, and aspect-oriented modelling. Especially uncertainty appears to play an important role for requirements engineering and adaptive systems. The relatively large number of papers concerning other sub-domains of software engineering such as language engineering, product line engineering, software architecture and service-oriented computing underpins the importance of fault-tolerance and uncertainty for the entire field of research. Cyber-physical and distributed systems, as well as mobile computing, can be identified as relevant application domains due to the substantial impact of environmental conditions involved. Several papers concern multiple research domains, such that a prioritisation was necessary in these cases: When MDE concepts were applied to a concrete

use case, the paper was allocated to the application domain. For papers which can be matched to different software engineering domains, the main focus was taken as decisive factor.

From the set of relevant sources, 36 exemplary use cases could be extracted, which can be used to illustrate approaches to fault-tolerance, uncertainty, flexibility, or consistency management in general. 23 use cases focus on a conceptual approach, 6 are used for tool demonstrations, and 7 cover both purposes equally. In the following, use cases for fault-tolerance, uncertainty, and flexibility are briefly sketched; a complete list including use cases for consistency management and further classifications can be found online[8].

Fault-Tolerance

A frequently used example for fault-tolerance is a simplified video-on-demand system modelled with UML diagrams [EDG+11, Egy07a, KKD+17, ELF08, RE12b, Egy07b, Egy11, Egy06, XHZ+09, KKE19]. The system consists of a streamer retrieving and decoding the content, and a display showing the video and receiving user input. Each component is modelled with a state chart diagram, in addition to a common class and sequence diagram for both components. Design rules describe the semantic interrelations between state charts, class diagram, and sequence diagram, e. g., that a class method name be equal to the corresponding message name in the sequence diagram, or that a message sequence match the behaviour in the state chart. As a software tool cannot decide on its own whether the effects of automated fixes lead to a satisfactory solution, a fault-tolerant treatment is suggested, such that the decision about controversial changes is left for the user.

Tolerating inconsistencies has been a long-term research topic for databases, which is illustrated by an example dealing with a project management tool storing information about the utilisation of employees for projects, as well as how many hours per week they should work [Bal91]. Constraints ensure that the sum of hours an employee works in all projects is equal to their regular working hours. The time an employee is needed for a project is maintained by project managers, whereas the regular working hours can only be changed by the business office. Reducing the amount of working hours for an employee makes it necessary to also reduce their time budget for one or more projects. It is, therefore, not possible to independently change project plans or working hours without temporarily introducing inconsistencies.

In the requirements engineering domain, requirements can be described with model fragments that conform to a core requirements metamodel [PBBT09]. The complete specification for a library management system, in which books must be registered such that customers can borrow them, can be created by fusing all fragments into one model. As this procedure can possibly lead to inconsistencies and the loss of metamodel compliance, it is necessary to temporarily relax the metamodel regarding abstract classes, multiplicities, and containment relations. Metamodel conformance is later restored by fixing the remaining inconsistencies.

The DOPLER tool suite [VGH+12] is used in software product line engineering for sales support systems for product configuration. Based on Eclipse, the tool is able to manage consistency between a variability model, a calculation model, and document templates. All models can be edited in parallel via suitable editor windows. Inconsistencies are tolerated in a way that their immediate resolution is not enforced, but occurring problems are listed in the Eclipse error viewer. For this specific use case, the tool suite can be regarded as an example for a fault-tolerant software modelling tool.

[8] https://drive.google.com/file/d/1uSuOn3hX5BHpLhw3jaH2ZpVffgTxHOov

Adaptive systems have to cope with the impact of environmental conditions; this is also reflected in their software models. For flood warning systems, it is important to predict floods as early as possible to reduce damage [GC08]. In a distributed system of sensors, water depth is calculated with pressure sensors, while flow speed is determined with camera sensors. The sensor nodes transmit the information to a gateway node, which forwards the predictions to an off-site server. The system needs to be fault-tolerant because signals can get lost, nodes can get disconnected, etc. Uncertainty is also involved regarding the execution environment and an appropriate trade-off between functional behaviour (e. g., prediction accuracy) and non-functional characteristics (e. g., energy efficiency) for changing environmental conditions.

While some exemplary use cases for fault-tolerance in MDE and a tool suite for fault-tolerant software product line engineering have been proposed in existing work, the examples and the tool support are specific for a particular application domain. Holistic concepts for handling fault-tolerance in software modelling could not be found in the course of the SLR.

Uncertainty

Another use case for systems that dynamically adapt to uncertain environmental conditions is a smart phone app for shop reviews, which provides users with information about lower prices for a product [GPST13]. The product's bar code is scanned with the camera to identify the product, and the user's position is determined to make suggestions for nearby shops, while an online search in web shops is performed simultaneously. However, the quality of the photo, the positioning system and the availability of mobile data represent sources of uncertainty that have to be taken into account when modelling the system.

Uncertainty in model-based testing is demonstrated by testing UML specifications for a video conference system [JLC+18]. The models store information about the number of connected participants and the video quality. Changing environmental conditions, such as packet loss in the network, or joining and leaving participants, are the primary sources of uncertainty.

The use of type systems can be substantially influenced by the uncertainty of measured values. In an illustrative case study, a toy car drives along a straight track, which is partitioned into multiple sections. Within this set-up, the car's velocity and acceleration on each of the sections [MWV16] should be computed. The system model involves uncertainty regarding the length of the sections (at design-time) and the time measurements (at run-time). Besides these absolute values, also relative values, such as the velocity and acceleration of the car, are uncertain.

Several small-scale but useful examples for modelling with uncertainty originate from MDE research itself. In an e-commerce application for selling books, data about books, authors, comments on the books, and details on books and authors is shown to the user [BEP+17]. A user interaction model specifies how a user can navigate between the respective views with help of the Interaction Flow Modeling Language (IFML). Due to a combinatorial explosion, it is challenging to evaluate all possible alternative flows with usability tests. Integrating uncertainty in IFML models, however, can help to specify a compact encoding of all these alternatives.

Code refactorings can be expressed in software models in terms of transformation rules. This becomes especially challenging for models incorporating uncertainty [FSC12b]. To explain transformation semantics on uncertain models, it is assumed that a modeller might not be sure whether an attribute should be added to the subclass or the superclass of an inheritance hierarchy. Furthermore, the model is to be refactored by adding get- and

set-methods to both classes, which leads to an exponential growth of possible results, demonstrating the need for a compact encoding of uncertain values.

In a fictional automotive design project, modelling with uncertainty is motivated for UML class diagrams. The three involved classes represent controllers for engine, body, and security of the car [SCG12]. Similarly, a perception system for autonomous driving is used to demonstrate uncertainty occurring during object detection and position determination in a scene [SPV20]. In a partial model (involving design uncertainty), each controller's attributes are modelled as attribute sets, which can be refined to discrete attributes by partial model refinement. Besides attributes, this refinement step can affect the existence of an inheritance relation (e. g., between the classes car and vehicle) or the knowledge if car and vehicle are actually the same class [SFC12].

Smart home systems provide solutions for intrusion detection with sensors, which are however exposed to uncertainty on several levels. Both imprecise measurements on sensors, the network infrastructure which connects the sensors and the interactions between physical units can be sources of uncertainty, leading to false positives and negatives when triggering alarm signals [CR20, ZAY+19].

The design of an automatic reasoning engine for logical expressions is taken as a use case for design uncertainty [FSSC13]. When the reasoning engine reaches an error state, a solver exception should be thrown, whose concrete implementation has some points of uncertainty. The exception may be an inner class of the solver, or an attribute could possibly provide more information about the error type. A similar use case for uncertainty resulting from incomplete requirements is presented via an UML state chart for a bank ATM. Depending on the required level of strictness, the ATM can either be restarted or set to be out of service in case of errors [EPR14].

Finally, a framework for model-based testing under uncertain conditions was presented in recent work [ZAY+19] to cope with the inherent uncertainty of CPS components. Connecting it to a test ready model evolution framework [ZAYN17], it is possible to generate further test cases for CPSs from evolved models.

In summary, various use cases and tool support for modelling with uncertainty can be identified, which are tailored to a specific application domain, though.

Flexibility

Some of the considered use cases illustrate the use of flexibility in software modelling, which can be seen as a generalisation of the two concepts fault-tolerance and uncertainty.

Motivating examples for flexibility with respect to metamodel conformance are proposed in co-evolution scenarios. When keeping class diagrams and relational databases consistent [KEK+15], refactorings on the metamodel make it necessary to adapt the transformation definition and the models, which should comply to this modified metamodel. As common examples for refactorings, deleting or moving attributes to other classes, introducing inheritance relations, or renaming references are listed.

In a similar setting, a family register is to be kept consistent with a persons register [SB16], such that, for example, the first and last name of a family member should be consistent with the full name of the corresponding person. When the metamodel is adapted, e. g., by adding a nickname attribute to the family member or by fusing first and last name, it is useful to relax the conformance relation by (temporarily) deactivating type or cardinality checks.

Especially when working with EMF, co-evolving metamodels introduce problems and additional effort for the persons involved, which is illustrated by a small case study modelling the network infrastructure of an office, including all shared gadgets such as scanners,

photocopiers and fax machines [RKPP09]. As the EMF editor always enforces strict metamodel conformance, it is not possible to work with models conforming to older versions of the metamodel. A common workaround is the trial-and-error strategy of loading a model to get an error message from EMF, and then attempting to fix this error directly in the XMI document, obviously a tedious and error-prone task.

To model flexibility in software processes, the Eclipse Process Framework Composer (EPFC) was extended to ease the collaboration of the involved persons [MVD08]. The process engineers propose a flexible work-flow, which can be adapted by other participants in a second step.

2.6 Benefits and Challenges

To investigate the third research question, arguments were collected that support or question the use of fault-tolerance or uncertainty. As benefits and drawbacks differ, the two concepts were analysed separately. More general arguments, which deal with more flexibility in software engineering, concern both concepts and are discussed afterwards.

Fault-Tolerance

A range of benefits resulting from tolerating inconsistencies to some extent was identified during the SLR.

In some application scenarios, such as distributed software systems, fault-tolerance is required to achieve availability and partition-tolerance [HDH10,HM05,Dec11,Bal91]. Similarly, being able to handle inconsistencies is essential due to the modularity of applications and data sources in modern software systems; faults can easily occur when composing building blocks in a new way, even if each module is implemented correctly [FYCL09, DM08]. A frequently mentioned point is that temporarily tolerating inconsistencies can ease the work-flow for system designers and testers as a fault-tolerant Integrated Development Environment (IDE) does not enforce the restoration of consistency before further modelling steps can be performed [VGH+12,PK04,PBBT09,GZJ16,GdL18,JKT16,Egy11, SKE+14].

Usually, atomic changes such as graph edits can lead to inconsistent states in between, which should be tolerated at least until the entire edit sequence is completed [KKE18, Ste14, Dec11, YV00, GZJ16, VGE+10, RE12b, VPM90, GdL18, dSOdR+03, Egy11, NER01, KKE19]. From a practical point of view, improving consistency might be more helpful than enforcing it strictly. Often, the restoration process requires multiple changes that can each be regarded as an improving step towards consistency, while only the last one is able to finally restore consistency [Ste14, Dec11]. Furthermore, a detected fault often reveals another problem, which might be the root cause of multiple other defects. Therefore, information about inconsistencies is often more helpful than an automated fix that achieves consistency [ELF08, Egy07b, KKE19].

In fault-tolerant systems, the number of automated changes can be decreased, which can improve the tool's trustworthiness for the designer [Ste14]. Even if automated changes are the preferred way of resolving conflicts, their application often relies on a central authority to define a policy for restoration steps. Especially when more than two models are involved, it is in general not possible to declare one of the models as the authoritative one, or prefer a certain type of changes over others due to, e.g., transitive consequences [Fri18, Ste18b]. It follows that, whenever a design decision is ambiguous, only the uncontroversial steps can be fully automated - leading to a possibly inconsistent state - before the user must participate in resolving the remaining inconsistencies [Ste14, EDG+11, XHZ+09].

Another set of arguments refers to weaknesses of fixing-procedures. To maintain an acceptable level of efficiency, many approaches apply local fixes to restore consistency. This means that mechanisms do not have a global view on the modelled system, and local fixes can have side-effects that are not monitored by the tool [EDG+11, Egy07a, HHR+11]. Consequently, these fixes might introduce new inconsistencies, which are often hard to detect, and which have to be fixed at a later point [Dec11, Egy07a, ELF08, KR07, Egy11, KKE19, HHR+11]. As multiple stakeholders are involved in the modelling of complex systems, their requirements can be contradictory. Without being able to tolerate these defects for a while, the modelling process gets stuck at this point and requires an instant resolution [DC16, Egy06]. Last but not least, the consistency check itself can be erroneous, such that the respective tool finds false positives. While it is undisputed that the fault has to be removed, fault-tolerant behaviour could again ease the continuation of the modelling process [Red11].

Despite this long list of advantages, many authors argue against involving fault-tolerance in system design. It is questionable how long and to which extent inconsistencies should be tolerated, because a large number of factors have an influence on the value of fault-tolerance in a specific use case [HDH10, PBBT09, BMMM08, Egy07a, KKD+17, ECK06]. Certainly, one should not lose track of the goal of eventually restoring consistency. Assuming that it is always possible to fix inconsistencies at a later point, this might still involve additional effort and thus a higher cost [Egy07b, Egy06, RE12a, SCH12, HTJ92, KKE19].

On the one hand, when consistency should eventually be restored, the developer might be confronted with so many faults at once that they are overwhelmed. On the other hand, faults might be caused by changes that occurred so long ago, that design decisions have to be completely revisited [Egy06]. As developers are usually willing to fix faults as soon as they are noticed, they will probably do the same when they detect inconsistencies that could actually be tolerated by the system. This means that such a tool's potential for supporting fault-tolerance will probably be ignored by its users [KPP08a]. Also, performing operations (e. g., model transformations) on inconsistent models likely introduces further faults [KKE19]. Finally, many tools for model management are based on formal methods, which are not yet compatible with fault-tolerant concepts [DM08].

Uncertainty

An important argument for modelling with uncertainty is that it represents real-world scenarios more accurately. The input data and the behaviour of CPSs and adaptive systems is imprecise, e. g., their sensors and actuators provide the system with imprecise data, and this should be reflected in a model of the system [KCT+15, MWV16, GC08, VMO16, CBGS18, PK19, BLC18, AY15, CGSB17, EM10]. Similarly, information might not be available in distributed systems, or not accessible due to security restrictions or authentication problems [GHYZ11].

In software and requirements engineering processes, uncertainty plays an important role, especially in early phases. When designing complex and widely heterogeneous systems, multiple stakeholders are involved, whose understanding of the final result may be incomplete [HWF13, FS10, FBDD+15, FS13, CBGS18, FSC12a, RCBS12, JBEM10, HT99, FSSC13, DC17, Gar10, CR20, EPR14]. Additionally, this incomplete information makes it necessary to continuously adapt the development process and the requirements [CSBW09, MSD06, SCH12, CBGS18, RCBS12, Gar10, FSC12b, SCG12]. Likewise, removing uncertainty too early can force the designer to commit to premature decisions that can increase cost and efforts to remove resulting faults later [FSSC13, EPR14], while ignoring uncertainty completely decreases the overall quality of the software [IFED09, EM10].

An obvious alternative to model uncertainty is to list each alternative value explicitly, but this can quickly become infeasible. Uncertainty is an elegant way to encode alternatives and non-determinism, while still keeping models manageable [BEP+17, BCC+16, EPR15, RE13]. Indeed, uncertain values are probably easier to maintain than a large set of alternatives [BCC+16, GPST13]. In case some design choices are more likely to be applied than others, uncertainty is also useful to express these varying probabilities quantitatively [ORS09]. From a practical point of view, uncertainty is often indirectly added to models via informal annotations in case its direct expression is not supported. Therefore, enabling the designer to model uncertainty in the given formal notation improves the verification and validation of such models [SCG12].

Handling uncertainty, however, can also increase development cost as the encoded set of alternatives - which takes all possible combinations of values into account - can be much larger than the set of possible options in practice [CSBW09, HT99, IFED09]. Following the same argument, the model space grows exponentially with the degree of uncertainty, which can cause performance problems for larger model sizes [ORS09, FSC12a, EM10]. Even though uncertainty is an appropriate way of expressing probabilities, it can be difficult to realistically quantify uncertainty measures, as empirical tests for these measures are often missing or cannot be conducted at all [ORS09, EM10]. Finally, operations on models are usually designed for single models, whereas uncertain models encode a whole set, restricting the applicability of standard tooling [FSC12a].

Flexibility

It is possible that multiple consistent solutions exist, which deviate in their quality. As it is hard to specify which solution should be taken, the system should provide the flexibility to let the user make this final decision [WXH+10]. To keep the complexity of a system manageable, software is usually developed with an idealised environment in mind. However, the system is also expected to react robustly to environment changes and unforeseen circumstances at runtime [FHA17, WGP09, HT99, HFS+15]. In application domains such as product line engineering, "hard constraints" can reduce the potential of the modelling language and therefore restrict the scope of action for the designer [BM14, Aqu09]. In the area of model-metamodel co-evolution, some flexibility is necessary for a modelling tool to be appropriate for practical use. The temporary loss of metamodel conformance should not lead to a situation where the model cannot be modified or even loaded in the respective editor [RKPP09, HS17, Hil16, AGF15, HKB16, ZCM+16, SB16]. Finally, the result of a model transformation is often not unique, requiring a flexible encoding [CK13, EPR14].

A few arguments can also be found that question the benefits of flexibility. As tools are typically not built by the intended users, the developer might have a different understanding of flexibility, such that the user may ignore or even disregard any support for it [HWF13]. The more flexibility is added to a system, the more complexity is involved as well, which can end up in a misinterpretation of functionality or a loss of overview while developing and maintaining the tool [SZ04, FGdL12, FS13] Finally, in case of faults and other inconsistencies, software developers are currently used to instant feedback from IDEs for General Purpose Languages (GPLs), and will probably expect similar behaviour from modelling tools. Transitioning to fault-tolerant tooling will therefore require a certain retraining of users to ensure acceptance, and it is still unclear how challenging this will be in practice [Egy06].

In summary, various arguments for adding fault-tolerance, uncertainty or flexibility to software models have been identified in the scope of this SLR, further motivating the development of a fault-tolerant consistency management framework.

2.7 Result Analysis

This section provides an overview of aggregated meta-data of the considered sources, before directions for future research are sketched, which can be motivated by this SLR.

The distribution of all sources in the database, i. e., all core papers and all sources cited by them, is depicted in Fig. 2.6, whereby ten sources published before 1975 are not captured in the diagram. In total, 268 core papers, 1146 other papers at SE venues and 2055 other sources were found in the initial search step. The median (mean) publication year is 2010.5 (2009.78) for core papers, 2007 (2005.87) for other papers at SE venues and 2006 (2004.35) for the remaining sources. It follows that filtering the additional sources was necessary to keep the amount of work manageable, and also that being published at a venue listed as research area 0803 (software engineering) is a useful indicator for increased relevance; at least one third of the additional sources was published at such a venue. The differences between core papers and other sources with respect to the average publication year can be explained by the applied snowballing technique, by which only sources released prior to the initial source can be found. Furthermore, the set of sources published at non-SE venues include standard references in form of books and journal articles, which is probably the reason why the papers from SE venues are slightly newer on average.

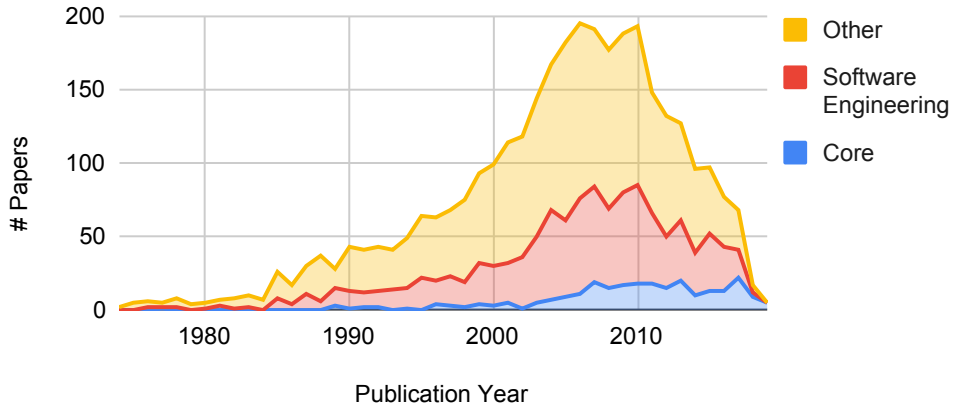


Figure 2.6: Number of sources per year

When taking only those sources into account that were later identified as being relevant for answering the research questions, the majority of sources originates from the set of core papers (cf. Fig. 2.7). Besides 114 of the core papers, 23 papers published at SE venues and 20 other sources were classified as relevant. Compared to the full corpus, the relevant papers are newer on average: The median (mean) publication year is 2011 (2011.05) for core papers, 2012 (2011.08) for publications at SE venues, and 2012 (2011.29) for the remaining relevant sources. An explanation can be that the search terms are used in a different meaning more frequently in older sources, according to our experience. Furthermore, as the research field MDE became popular along with the emergence of the UML in the late 1990s, sources published before can only be relevant for our purposes if they describe transferable concepts or examples from other domains.

Table 2.3 gives an overview of the number of relevant papers per venue, listing those venues with at least four relevant papers. As this SLR deals with a subtopic of MDE, finding the MODELS conference at the top of the ranking is a result one would expect. Five papers published at co-located events were relevant for this SLR as well. ICSE and ASE as two top-ranked SE conferences in the list underpin the topic's relevance for a

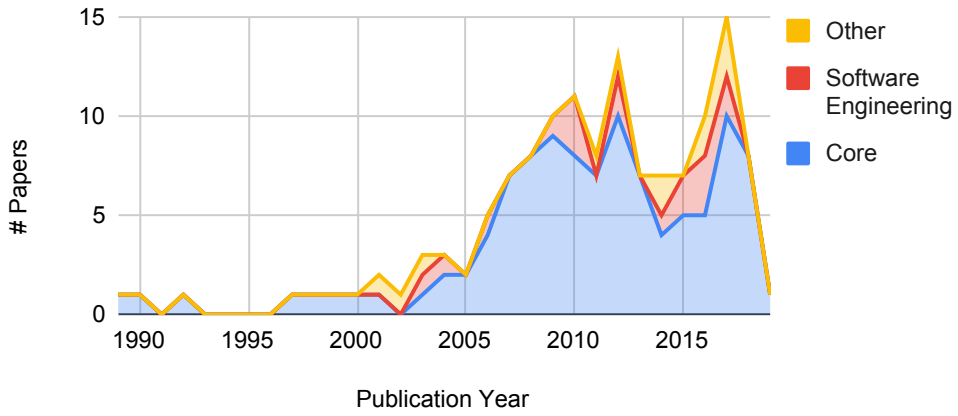


Figure 2.7: Number of relevant sources per year

broader audience. The appearance of important conferences for more specialised research fields, such as requirements engineering (RE), software language design (SLE) and software testing (ICST), shows that fault-tolerance concerns the entire software development process. The remaining well-known SE venues COMPSAC, SEKE, APSEC and FASE complete the list of venues with 4 or more relevant papers. Overall, the list of venues makes us confident that a topic of noticeable importance is successfully targeted by the conducted SLR.

Rank	Acronym	Venue	#
1	MODELS	Model Driven Engineering: Languages and Systems	20
2	ICSE	International Conference on Software Engineering	15
3	COMPSAC	Computer Software and Applications Conference	10
4	ASE	Automated Software Engineering	8
5	SEKE	Software Engineering and Knowledge Engineering	7
6	FASE	Fundamental Approaches to Software Engineering	6
7	APSEC	Asia-Pacific Software Engineering Conference	5
7	RE	Intern. Conference on Requirements Engineering	5
7	-	MODELS Satellite Events	5
10	SLE	Software Language Engineering	4
10	ICST	International Conference on Software Testing	4

Table 2.3: Top 10 conferences by number of relevant papers

2.8 Solution Approach

Although many thorough definitions, useful examples and convincing arguments for fault-tolerance in MDE could be extracted as a result of this SLR, a need for further research became apparent simultaneously. As fault-tolerance was mostly defined intuitively, an extended formal framework (e. g., for temporarily softening domain constraints) would be helpful to prove properties of fault-tolerant systems. This includes quantitative measures for (in)consistency, as well as quality criteria for the intermediate and final solutions to assess the utility of proposed approaches.

Up to now, it remains also unclear when exactly consistency shall be restored, up to which point faults can be tolerated, and how to deal with conflicting changes that were

made in the meantime. Consistency restoration is a non-trivial problem because removing faults from one model can introduce other faults in different places of the other model. Further studies could show to which extent user interaction is required to resolve such conflicts, and which restoration actions can be performed automatically. Several authors pointed out that some faults are too serious to be tolerated, but still strategies are needed to assess the severity of errors in a model, though.

While model transformations in presence of uncertainty are already investigated, fault-tolerant consistency management is still a widely unsolved issue in the context of MDE. Although fault-tolerance and uncertainty could be differentiated in spite of their common goal of making model-based software development processes more flexible and applicable, it would be useful to specify which of the two concepts is more helpful in which particular scenarios.

Finally, the review has shown that there are indeed realistic use cases for applying fault-tolerant concepts in MDE. However, we could not identify an example that convincingly demonstrates how model transformations and other consistency management tasks can be performed in the presence of faults.

The identified need for further research motivates the development of a fault-tolerant consistency management framework, which will be presented in detail in the remainder of this thesis. A brief overview of the solution shall be presented at this point already. Based on the feature models of Sect. 2.4, the proposed solution is classified in terms of existing research on fault-tolerant consistency management. The underlying notion of consistency, which the approach presented in this thesis is based on, is depicted in Fig. 2.8. Features which are used in the approach are highlighted with a bold border, whereas unused features are greyed out.

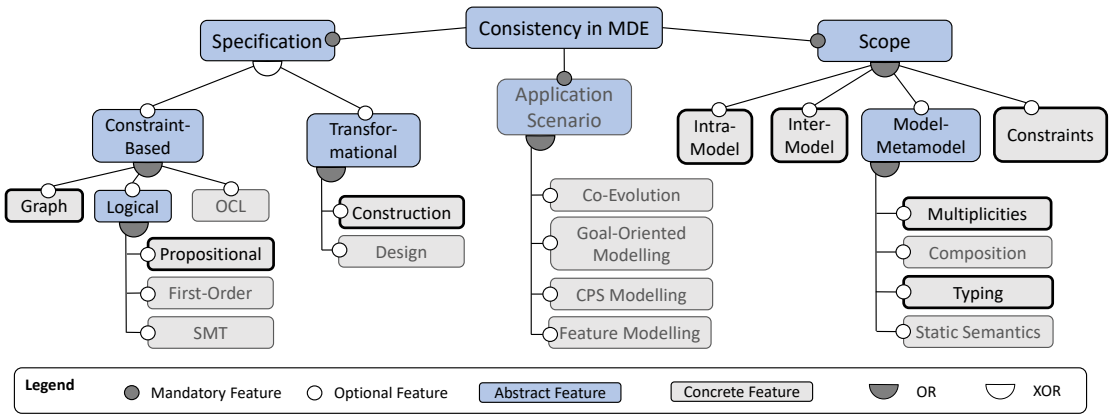


Figure 2.8: Features of the proposed solution with respect to consistency

First, the specification of consistency plays an import role. As one can see in the left branch of Fig. 2.8, our consistency specification consists of multiple building blocks. Constraints can be specified in form of graph patterns, which consistent models have to conform to. Furthermore, TGGs as a formalism for defining BX are the transformational part of the specification. Declarative rules describe how to construct consistent triples, and thereby forming the consistency relation. Technically, rules, constraints, and their interdependencies are encoded as a optimisation problem, i. e., an ILP. As the used variables are all binary, we are able to specify linear constraints for the ILP which are equivalent to formulae in propositional logic.

Second, the scope of our consistency definition is relatively wide: Both intra- and inter-model consistency are covered, meaning that the approach can detect faults both in a single model and in the interplay of multiple models. Additionally, model consistency is dependent on the relation between models and metamodels. From the range of aspects such a model-metamodel relation can have, the solution approach will take multiplicities and typing into account. The consistency definition can be further enriched with constraints that cannot be directly derived from the metamodel.

The notion of consistency enables us to sketch how fault-tolerance is implemented in our approach. An overview of which features of fault-tolerant MDE are considered is provided in Fig. 2.9. A first important aspect is the relaxation of constraints: The requirement of getting fully consistent input models is dropped, such that the consistency management engine must be able to compute a solution for inconsistent inputs as well. The idea is to show the remaining faults to the user, but not to enforce an immediate fix. The user should be aware of existing faults, but should be enabled to continue working with a tentative solution and fully restore consistency at a later point. From this perspective, the approach can be considered as semi-automated: The engine can solve a consistency management task without further user interaction, but the user can (re-)start the process multiple times with different model versions, taking the feedback from previous runs into account. In total, the fault-tolerant system behaviour should ease the work-flow and increase the acceptance of TGG-based consistency management in practice.

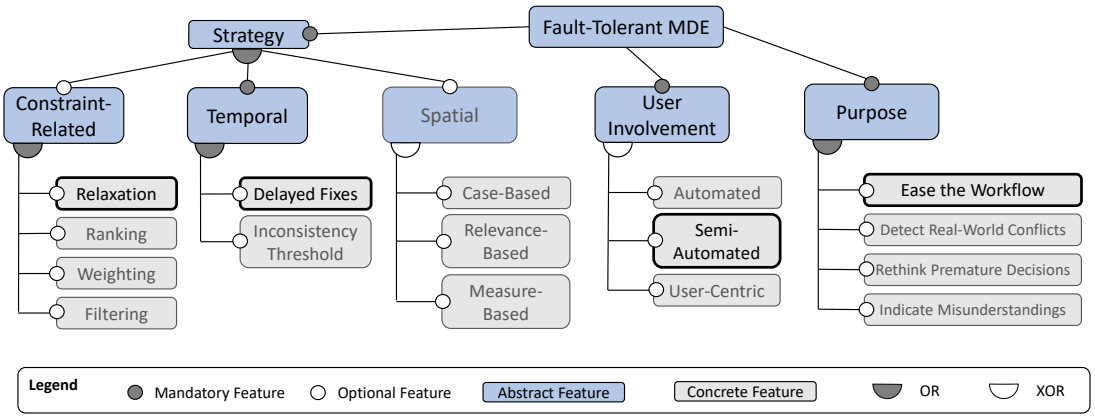


Figure 2.9: Features of the proposed solution with respect to fault-tolerance

2.9 Summary and Discussion

We presented the results of an SLR on fault-tolerance in MDE, which took 157 relevant sources into account. The key terms consistency, fault-tolerance, and uncertainty were defined and represented in feature models, such that salient differences and commonalities between fault-tolerance and uncertainty could be pointed out. Typical use cases for fault-tolerant and uncertain modelling were sketched, and benefits and challenges of the respective concepts were discussed. To ease the reproducibility of our results and to support future SLRs in computer science, we proposed a model-driven tool chain based on open-source components under active development.

The scope of the SLR can be reasonably extended in future studies: Although many relevant journal articles and workshop papers were identified by the snowballing step, these venues could already be included in the initial search step as well. Since the CORE2020

journal ranking[9] was recently made available, we plan to extend the literature review towards journal papers following the search strategies presented in this chapter. To the same end, other research databases could be considered as well.

Motivated by the identified need for further research, a TGG-based framework for fault-tolerant consistency management is proposed in the remainder of this thesis. To demonstrate the different concepts, the introductory example of maintaining consistency between the semi-formal language SysML and the formal language Event-B is continued. With this example, we were able to demonstrate Stevens' arguments for tolerating inconsistencies [Ste14] in the introduction (cf. Chap. 1). Furthermore, the consistency relation between these two languages is of practical relevance for systems engineering problems: In Chap. 11, it will be shown how a TGG-based model transformation tool chain can be applied for system verification and validation in the railway domain. The next chapter will introduce the languages of the running example and the TGG formalism to form a basis for presenting the fault-tolerant consistency management framework in the subsequent chapters.

[9] http://portal.core.edu.au/jnl-ranks/

3 Modelling Software Systems: Languages and Transformations

In this chapter, the basic language constructs of SysML and Event-B are introduced, which have been used to provide a first intuition for the problem area of this thesis in Chap. 1. The example originates from a real-world case study, in which these two languages are used to engineer safety-critical systems at DB Netz AG, an infrastructure manager of the German railway system (cf. Chap. 11). The following chapters will use a BX between these two languages to demonstrate the proposed concepts for fault-tolerant consistency management, because from our point of view, the examples presented in Sect. 2.5 are less suitable in the context of different consistency management operations.

SysML can be considered as a de-facto standard in the systems engineering domain. Similar to the UML, SysML offers structural and behavioural diagrams, whereby we will restrict ourselves to a language subset that is sufficient to describe basic SysML state machines. Event-B, in contrast, is a formal language for verifying and validating safety-critical properties of a system via invariants and theorems. Also for Event-B, only the basic language constructs of behavioural state machines are considered.

In order to formally describe the consistency relation between the two languages, TGGs are chosen to be used in the course of this thesis. TGGs are a well-known rule-based approach to bidirectional transformations in MDE, and are used in various application domains such as synchronising textual [KKS07] or visual [GdL06] software languages. TGGs were also used in industrial projects, e. g., to synchronise AUTOSAR and SysML [GHN10], or to automatically translate satellite procedures [HGN+13]. To represent the consistency relation, a third *correspondence* model is introduced to map semantically related elements of the two other models to each other. The main advantage of TGGs is that different operations like model transformation, model synchronisation and consistency checking can be performed using the same underlying specification.

The remainder of this chapter is structured as follows: The two languages of the running example of this thesis, SysML and Event-B, are introduced in Sect. 3.1 and 3.2, before a brief overview of the fundamentals of algebraic graph transformation and TGGs is given in Sect. 3.3. Section 3.4 summarises the main results and motivates the introduction of further TGG language features in Chap. 4.

3.1 SysML: A Semi-Formal Language

SysML was developed from UML as a general purpose language for MBSE [HP19]. It can be regarded as both an extension and a restriction of UML including nine diagrams subdivided into structural (block definition diagram, internal block diagram, package diagram), behavioural (use case, activity, sequence, and state machine diagram), requirement and parametric diagrams. SysML is a primarily visual modelling language and aims to be easily understood by system engineers. It has gained popularity in different fields such as aerospace, defence, and medical industries [HP19]. While the SysML specification provides nine diagrams in total, we restrict ourselves to the transformation of *state machine diagrams* in the scope of this thesis. In the following, we introduce the relevant concepts

of SysML by an extended version of the motivating example for multiple synchronisation solutions (cf. Sect. 1.1).

The example state machine is shown in Fig. 3.1. As a basis, the second option depicted in Fig. 1.4a is chosen, which implements the condition that the variable `finish` should have the value `FALSE` before leaving the start state as a guard. Furthermore, a trigger is added to the transition that requires the variable `exec` to be `TRUE`, which might be necessary if the state machine waits for an execution command from outside the system. In the following, the relevant language constructs of SysML will be presented in brief.

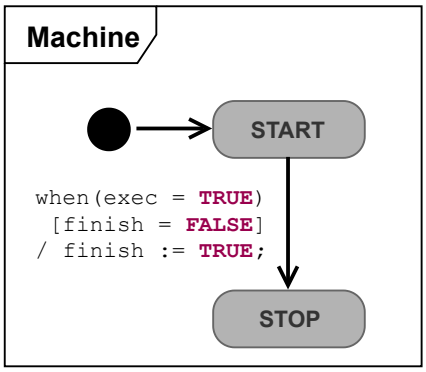


Figure 3.1: SysML state machine

State Machine and Region

In SysML, there are two types of state machines: *Behavioural state machines* describe the behaviour of subsystems, whereas *protocol state machines* define valid interaction sequences, denoted as *protocols*. For defining the transformation to Event-B, the focus is set on behavioural state machines.[1] The state machine of Fig. 3.1 is an example for a behavioural state machine, identified by its name on the top left. A state machine can consist of multiple regions, while only one region is necessary for this example.

State

A state models a situation in the execution of a state machine, during which some invariant condition holds. States are also considered the fundamental building blocks of the state machine. In each active region, at most one state can be assumed at the same time. States can be subdivided into simple, composite, and sub-machine states; only simple states are relevant for the scope of this thesis. Simple states do not have any sub-states, regions or internal transitions. Simple states can either be atomic or final states. While atomic states have no special meaning, final states, when active, indicate the completion of their parent region, i. e., the admissibility of a given input sequence. Figure 3.1 depicts two simple states (`START` and `STOP`), which are both atomic.

Pseudo-State

The difference between a state and a pseudo-state is that the latter cannot be assumed by the state machine. Pseudo-states are typically used to connect multiple transitions into more complex paths. For example, a fork pseudo-state with a single incoming and multiple outgoing transitions can be regarded as a compound transition that leads to a set of orthogonal target states. Pseudo-states can be classified as initial states, junctions, choices, forks and joins, entry and exit points or history states. In the scope of this example, we restrict ourselves to initial states, whereby an extension towards other pseudo-state types is possible. In a region, there can be at most one initial state present. The initial state has only outgoing but no incoming transitions. The outgoing transitions of the initial state cannot trigger any event and have no guards. An initial state is depicted as a small solid filled circle (cf. Fig. 3.1).

[1] To distinguish state machines from the eponymous diagram type of the UML, they are also referred to as SysML state machines.

Transition, Event, Effect, Trigger, and Guard

A *transition* in a state machine is a directed association between a source state and a target state, and can be expressed in the following form:

```
transition ::= [trigger][guard]['/'effect]
```

In Fig. 3.1, two transitions are depicted as arrows between the three states. An optional *trigger* can be used to specify an event that induces a state transition. An *event* is a notable occurrence at a point in time that causes a reaction of the state machine. These reactions lead to an execution step of the modelled behaviour. For example, a signal event may trigger a transition of a state machine. In Fig. 3.1, for instance, `exec` triggers the transition from `START` to `STOP`. An optional guard specifies additional constraints as a boolean expression. The transition from `START` to `STOP` in Fig. 3.1 can only be fired if the variable `finish` has the value `FALSE`. An optional *effect* is an action to be executed when its transition fires. While effects are sufficient for our current considerations, further action types such as *entry*, *exit*, and *do* actions can be added as an extension of the transformation. Considering the running example of Fig. 3.1, the variable `finish` is set to `TRUE` as an effect of the transition from `START` to `STOP`.

Port

Ports are interfaces via which external entities can connect to and interact with the specified system. In the running example, the state machine receives the instruction whether to execute the transition from `START` to `STOP` via the port `exec`. The port `finish`, in contrast, is rather used as an output signal to indicated whether the `STOP` state is already reached. Ports are often used to trigger events. In state machine diagrams, they lack a distinguished visual symbol and are simply represented as textual variables in triggers, guards or actions.

3.2 Event-B: A Formal Language

Even though SysML is appropriate to describe the behaviour of the state machine, it is not possible to verify safety-related properties, due to its lack of a formal semantics. In the context of the industrial case study, this is important to verify and validate the constructed system models, which motivates us to establish a BX between SysML and the formal language Event-B. An overview of the relevant syntactic constructs of the Event-B language is provided in the following.

Event-B is a formal method for system level modelling and analysis.[2] Its key features are the use of set theory as a modelling notation, defining a system at different levels of abstraction via refinements, and the verification of formal properties via theorems and invariants [AH07]. In the following, the language constructs which are relevant for the transformation of SysML state machines are briefly presented.

Machine, Variables, Events

In Event-B, a machine defines the behavioural properties of the model. Each machine is composed of variables (v), invariants ($I(v)$), and a collection of transitions denoted as events (cf. the schematic example[3] depicted in Fig. 3.2). There are further optional

[2] http://www.event-b.org/
[3] Adapted from http://deploy-eprints.ecs.soton.ac.uk/11/3/notation-1.5.pdf

building blocks annotated with a * in Fig. 3.2: A machine can refine another machine and be embedded into one or more contexts. Refined machines have an additional variant block containing an expression that is unique for the machine. Theorems $R(v)$ are additional properties that must be derivable from the set of invariants $I(v)$. For each theorem, a proof obligation is generated to prove that the theorem is derivable from the invariants, i. e., $I(v) \vdash R(v)$ [Hoa13]. As the mandatory components are sufficient for describing the running example, we will now describe them in the remainder of this section.

```
MACHINE
  <machine_identifier>
REFINES *
  <machine_identifier>
SEES *
  <context_identifier>
  ...
VARIABLES
  <variable_identifier>
  ...
INVARIANTS
  <label> : <predicate>
  ...
THEOREMS *
  <label> : <predicate>
  ...
```

```
EVENTS
  <label> ≙
    STATUS
      <status>
    WHEN
      <label> : <guard>
      ...
    THEN
      <label> : <action>
      ...
    END
      ...
VARIANT *
  <variant>
END
```

Figure 3.2: Event-B: Schema for machines

The set of variables v defines the current state of an Event-B machine, and is used in several other language constructs. For the running example, four boolean variables are required (cf. Fig. 3.3): For the current state of the state machine (`START`, `STOP`) and for the signals to communicate with the system's environment (`exec`, `finish`), one boolean variable per signal is required. The variables for the states indicate whether the machine currently assumes the respective state or not.

```
MACHINE
  Machine
VARIABLES
  START
  STOP
  exec
  finish
END
```

Figure 3.3: Event-B: Machine with variables

Invariants

Invariants $I(v)$ define constraints which have to hold at any time, i. e., in each possible state of the machine. They can involve one or more variables forming an expression in first order logic. Each invariant has a label, followed by a colon and an expression. In Fig. 3.4, the invariants `typeof_START`, `typeof_STOP`, `typeof_exec`, and `typeof_finish` define the data types of the respective variables, which is `BOOL` in this case. In contrast to SysML, it is possible to *type* variables, which makes it possible to restrict the set of legal

values that these variables can assume. This is important to, e. g., ensure that each state machine assumes exactly one state at a specific point of time.

```
INVARIANTS
  TYPEOF_START : START ∈ BOOL
  TYPEOF_STOP : STOP ∈ BOOL
  TYPEOF_exec : exec ∈ BOOL
  TYPEOF_finish : finish ∈ BOOL
```

Figure 3.4: Event-B: Invariants

Events, Guards, Actions

As a machine specifies the dynamic behaviour of an Event-B model, events are essential to trigger changes from one machine state to another. Events involve a set of variables v and parameters x for these variables. An event e occurs in some state, if there exists some value for its parameter x such that the *guard* $G(x, v)$ holds in that state. In Fig. 3.5, there is one event that describes that if the machine passes from the state `START` to `STOP`, the variable `finish` is set to `TRUE`. To fire this transition, the machine must be in the state `START` (`isin_START`) and the variable `finish` must be `FALSE` (`guard1`). Furthermore, an execution command must be given from outside to pass to the other state (`trigger1`). These three conditions represent Event-B guards of the event. Actions describe how state variable values change when an event occurs. Similar to a guard, an action $Q(x, v)$ involves parameters x and variables v. As soon as the event occurs, the specified values are assigned to the respective variables. Each assignment is identified by a label (cf. Fig. 3.5). In case of the `COMPLETION` event, the variable `finish` is set to `TRUE`, and the variables `START` and `STOP` flip their values. All guards are contained in a block following the keyword `WHEN`, the actions are surrounded by the keywords `THEN` and `END`.

```
EVENTS
  ...
  COMPLETION ≙
  STATUS
    ordinary
  WHEN
    isin_START : START = TRUE
    trigger1 : exec = TRUE
    guard1 : finish = FALSE
  THEN
    leave_START : START := FALSE;
    enter_STOP : STOP := TRUE;
    act1 : finish := TRUE;
  END
```

Figure 3.5: Event-B: Completion event

There are also events that do not have a guard, i. e., only consist of an action block. In our example, the initialisation event brings the state machine into the state `START` in the beginning, as depicted in Fig. 3.6. Syntactically, the `WHEN` block is omitted, and the keyword `THEN` is replaced by `BEGIN`. As the name already suggests, this event initialises all involved variables with their start values. Only the variable for the current state (`START`) is set to `TRUE`, all other variables receive `FALSE` as initial value.

```
EVENTS
  INITIALISATION ≙
  STATUS
    ordinary
  BEGIN
    init_START : START := TRUE;
    init_STOP : STOP := FALSE;
    init_exec : exec := FALSE;
    init_finish : finish := FALSE;
  END
  ...
```

Figure 3.6: Event-B: Initialisation event

In total, it is possible to define global constraints in Event-B that must hold at any point of time using invariants or theorems (which have not been presented as they are not part of the transformation). For verifying the system behaviour, appropriate constraints can be specified and checked in Event-B, which is not possible for SysML state machines. In the next section, we will see how a BX between these two languages can be established with help of TGGs.

3.3 Bidirectional Model Transformations with TGGs

In the scope of this thesis, we define inter-model consistency by means of TGGs. TGGs are a declarative rule-based approach to bidirectional model transformations, which is frequently applied in MDE contexts [CFH+09]. A source and a target model (in which these are interchangeable) are represented as typed attributed graphs. The consistency relation is expressed as a correspondence graph, which links elements of the source graph to the target graph.

Based on the original specification by Schürr [Sch94], the formalism was continuously enhanced with further language features (cf. Chap. 4) to meet the requirements of being applied to practical use cases [ALS15]. After identifying semantically interrelated constructs in both languages, the TGG formalism can be used to maintain consistency between SysML and Event-B models in a semantics-preserving manner.

In this section, the fundamentals of TGGs are formally defined and illustrated with concrete examples in the context of the previously introduced running example. The TGG formalism builds upon graph transformation, a means of specifying the behaviour of systems based on graph structures. While applications in other research fields exist (cf. [HO19] for a recent example in the biology domain), graph transformations are used in computer science to define the dynamic semantics of software systems. In the scope of this thesis, the *algebraic* approach to graph transformation is used, which uses category theory to define graphs and specify operations on them. Basic definitions are taken from Ehrig et al. [EEPT06] in an adapted version, which we also refer to for further background information about the category of graphs.

To begin with, graphs as the central data structure of the TGG formalism are defined in Def. 3.1. Graphs are used to formally represent models of the software system. In our formal framework, they can be connected by *graph morphisms*, i. e., structure-preserving maps of one graph to another. These mappings can, e. g., be used to describe transitions between system states.

Definition 3.1 (Graph (Morphism))**.**
A ***graph*** $G = (V, E, src, trg)$ *consists of a set* V *of nodes (vertices), a set* E *of edges, and two functions* $src, trg : E \to V$ *that assign each edge a source and target node, respectively. Given graphs* $G = (V, E, src, trg)$, $G' = (V', E', src', trg')$, *a* ***graph morphism*** $f : G \to G'$ *consists of two functions* $f_V : V \to V'$ *and* $f_E : E \to E'$ *such that* $src ; f_V = f_E ; src'$ *and* $trg ; f_V = f_E ; trg'$. *The* $;$ *operator denotes the composition of functions:* $(f ; g)(x) := g(f(x))$.

Definition 3.1 can be lifted in a straightforward manner to triple graphs and triple morphisms. A *triple graph* connects a source graph and a target graph via a correspondence graph. The correspondence graph indicates which elements of source and target graph are semantically interrelated, expressed by a *correspondence* link in between. Source and target graph are interchangeable, such that we choose the SysML model to be the source graph and the Event-B model to be the target graph for the remainder of this thesis, while the opposite choice would be possible just as well.

Definition 3.2 (Triple Graph (Morphism))**.**
A ***triple graph*** $G = G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T$ *consists of graphs* G_S, G_C, G_T *and graph morphisms* $\gamma_S : G_C \to G_S$ *and* $\gamma_T : G_C \to G_T$. *elem(G) denotes the union elem(*G_S*)* $\cup$ *elem(*G_C*)* $\cup$ *elem(*G_T*). A* ***triple morphism*** $f : G \to G'$ *with* $G' = G'_S \overset{\gamma'_S}{\leftarrow} G'_C \overset{\gamma'_T}{\rightarrow} G'_T$, *is a triple* $f = (f_S, f_C, f_T)$ *of graph morphisms where* $f_X : G_X \to G'_X$, $X \in \{S, C, T\}$, $\gamma_S ; f_S = f_C ; \gamma'_S$ *and* $\gamma_T ; f_T = f_C ; \gamma'_T$.

A first example for a triple graph is shown in Fig. 3.7. It covers only a small subset of the introductory example of Sect. 3.1 and 3.2, but is sufficient to demonstrate the building blocks of triple graphs according to Def. 3.2. The source, correspondence and target graph are depicted in this order from left to right. The source graph consists of three nodes, i. e., a state machine `sm`, a region `r` and a port `p`, as well as two edges that connect `sm` to the other two nodes of the source graph. The target graph also consists of three nodes, namely a machine `m`, a variable `v` and an invariant `i`, and two edges connecting `m` and the other two nodes. In between the two graphs, the nodes of the correspondence graph are depicted as hexagons, which are connected to the nodes of the source and target graph that shall correspond to each other[4]. There is, however, no 1:1 mapping between source and target nodes: The port `p` corresponds to both the variable `v` and the invariant `i`, whereas the region `r` does not have any corresponding target node.

Up to here, we defined (triple) graphs based on the involved sets of nodes and edges, but treated each of the two sets as a uniform collection of elements. However, it was necessary to use additional information for describing the triple graph of Fig. 3.7 in a reasonable way: A *type* was assigned to each node and edge to state that, e. g., `sm` is a `Statemachine` and `p` is a `Port` and not the other way round. Formally, typing information can be introduced by demanding a type (triple) morphism to a chosen type (triple) graph. In the following, all (triple) graphs and (triple) morphisms are assumed to be typed unless explicitly stated otherwise.

Definition 3.3 (Typed Triple Graph (Morphism))**.**
A typed triple graph $(G, type)$ *is a triple graph* G *together with a triple morphism* $type : G \to TG$ *to a distinguished type triple graph* TG. *A* ***typed triple morphism*** $f : \hat{G} \to \hat{G}'$ *is a triple morphism* $f : G \to G'$ *with* $type = f ; type'$, *where* $\hat{G} = (G, type), \hat{G}' = (G', type')$.

[4] By definition, the nodes of the correspondence graph can also be connected via edges. As this option is not used within this thesis, we restrict ourselves to correspondence graphs that only consist of nodes.

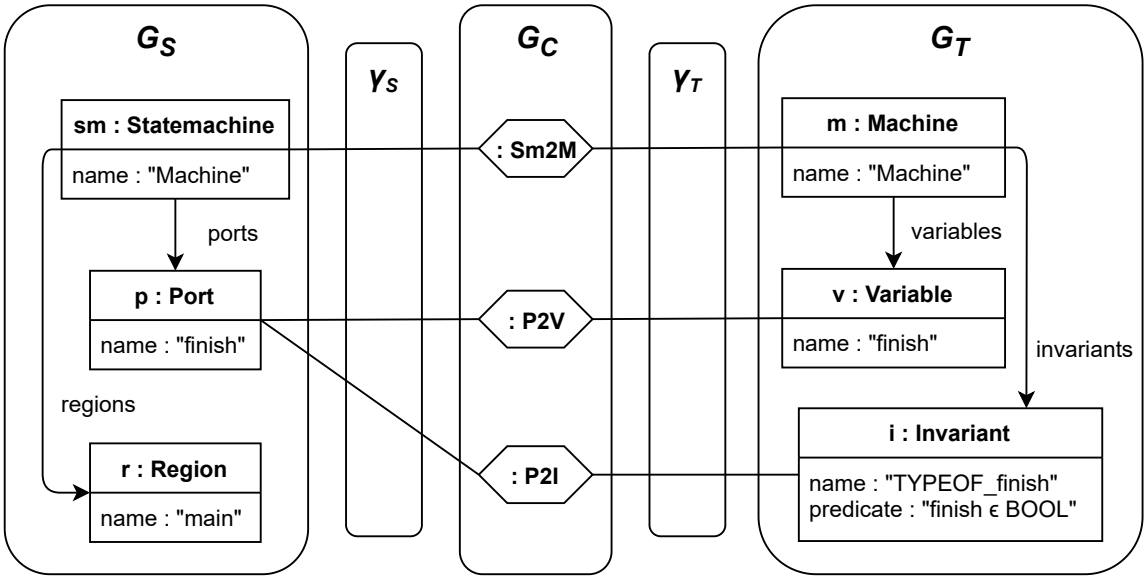


Figure 3.7: Triple graph instance

As already mentioned, (typed) graphs are a form of representing models of a software system. In this regard, type graphs assume the role of *metamodels*: A model G is an instance of a metamodel TG, if there exists a mapping $type : G \to TG$ that assigns each element of G a type of the type graph TG. Figure 3.8 depicts the relevant excerpt of the SysML metamodel for state machines to the left, the Event-B metamodel to the right, and the mapping in form of correspondences, forming the triple metamodel for the running example that is used, e. g., to type the triple graph instance of Fig. 3.7. To improve readability, only multiplicities different from `1` for the source and `0..*` for the target of an association are depicted.

The `Statemachine` class defines the primary behaviour of the modelled system and consists of `Regions` and `Ports`. The Event-B `Machine`, which consists of `Variables`, `Invariants` and `Events`, has the same purpose, so these classes correspond to each other. In the SysML metamodel, a `Region` consists of `States` and `Transitions` but has no direct correspondence in the Event-B metamodel. The `States` of a SysML state machine correspond to `Variables` in the Event-B model. As they can also be involved in the definition of `Invariants`, correspondence links to this language construct are also required. The same holds for `Ports` in SysML models, which are used for communication with external components. `Invariants` in Event-B specify the properties that a `Variable` must satisfy before and after each `Event`. For example, the data type and value ranges of variables that correspond to SysML `States` are specified by `Invariants`.

A `Transition` in SysML is annotated with `Triggers`, `Guards` and `Effects`, and it represents the directed relation between a source and a target `State`. Activating a `Transition` is similar to the occurrence of an `Event` in Event-B, thus these elements are connected via a correspondence. An `Event` incorporates `Guards` to restrict its occurrences and `Actions` that take place during the `Event`. `Guards` and `Actions` in Event-B thus correspond to `Triggers`, `Guards`, and `Effects` in SysML as depicted in Fig. 3.8. In Sect. 3.1, we made a distinction between `States` and `Pseudostates`, with the latter

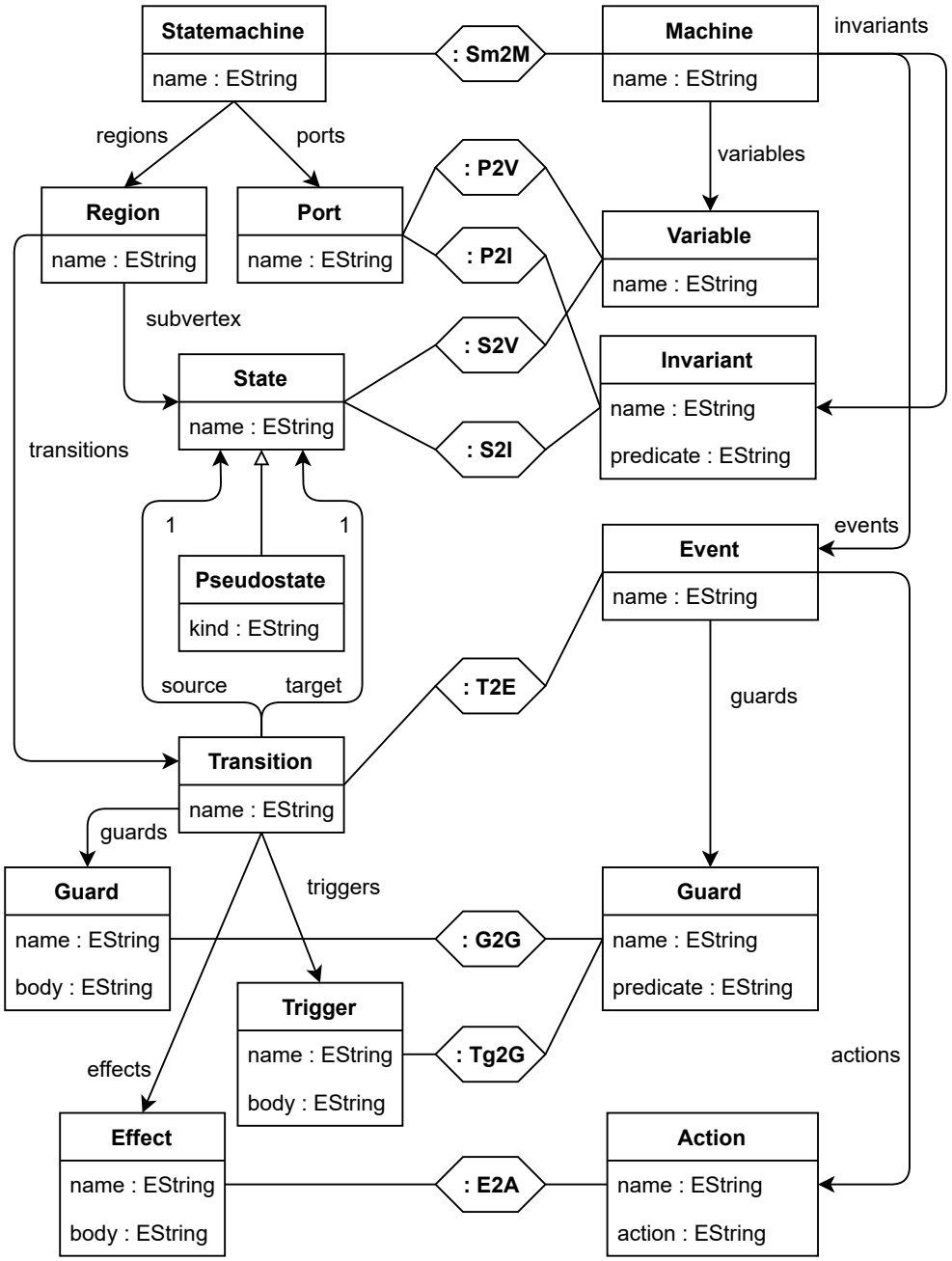


Figure 3.8: Triple metamodel: SysML to Event-B

being a subclass[5] of the former in the SysML metamodel. `Pseudostates` do not have a direct correspondence in Event-B and do not need to be translated. Their adjacent `Transitions`, however, correspond to `Events` (cf. Fig. 3.6), such that the semantics of `Pseudostates` has an indirect influence on the Event-B model. Such details cannot be expressed solely using the triple metamodel, however, as it is only used for typing model elements and defining mappings between nodes of particular types. Instead, TGG rules are used to fully specify the desired consistency relation between the languages, which is presented next.

A (triple) graph morphism can be interpreted as a *monotonic (triple) rule*, used to describe how to add structure to one (triple) graph to produce a new (triple) graph in a single transformation step.

Definition 3.4 (Triple Rule)**.**
*A **triple rule** $r : L \to R$ is a monomorphic (injective) triple morphism.*

The rule *PortToVariable* is depicted in Figs. 3.9 and 3.10 in two different notations. In Fig. 3.9, the graphs L and R and the morphism r are directly visible. L and R represent the left- and right-hand side of the rule, respectively, and r is a monomorphic arrow that maps all elements in L to elements in R in a structure-preserving manner. Elements that only appear in R are added when applying the rule, whereas elements that appear both in L and R (also denoted as *context elements*) are expected to exist already before rule application. The context elements in L and R are connected with dashed lines in this visualisation. A statemachine `sm` and a corresponding machine `m` are required to exist already in the host graph, such that a port `p` can be added on the SysML side, and linked to a variable `v` and an invariant `i` of the Event-B machine.

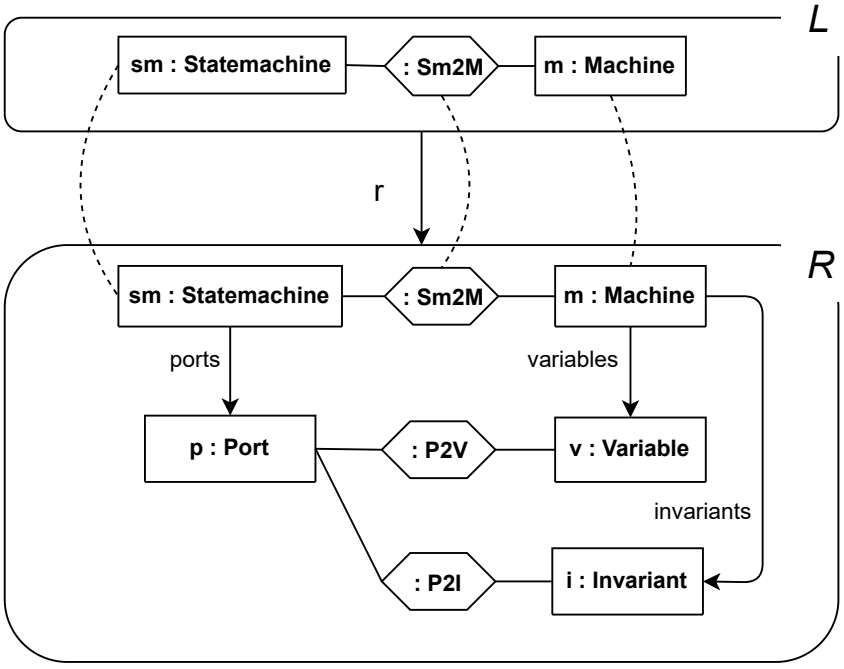


Figure 3.9: Rule: PortToVariable

[5]Inheritance for type graphs according to Def. 3.3 is not supported, but can be simulated by duplicating the respective nodes of the metamodel. We stick to a type graph with inheritance relations for presentation purposes.

The more compact visual notation for the rule *PortToVariable* is shown in Fig. 3.10. The triple graphs L and R are merged into a single diagram, whereby created elements and context elements can be distinguished by their colours and mark-ups: Green colour together with a ++ mark-up indicates that the respective nodes and edges are created by the rule application, while context elements are coloured black and do not have a mark-up.

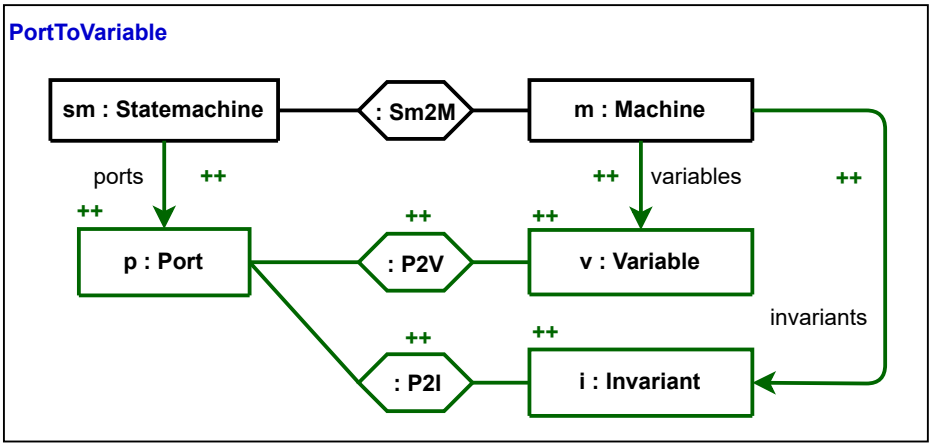


Figure 3.10: Rule: PortToVariable (compact notation)

Figure 3.11 shows another example for a TGG rule which creates a statemachine `sm` in the SysML model, and connects it to an Event-B machine `m` via a correspondence node. Furthermore, the rule adds a region `r` to the source model, which does not have a corresponding language construct in the target model as we have seen in Figs. 3.7 and 3.8 already. In contrast to *PortToVariable*, the rule *StatemachineToMachine* does not involve any context elements. Such rules are also denoted as *axioms*.

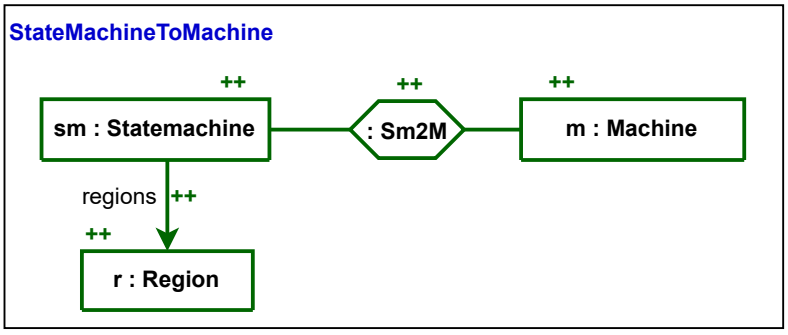
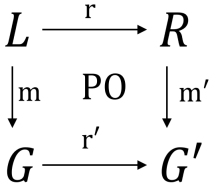


Figure 3.11: Rule: StatemachineToMachine

To actually *apply* a rule r on a triple graph G, a match m for the left-hand side L is required. m is a triple morphism, which again maps all elements of L to equally typed elements of G. The left-hand side L solely consists of context elements, which means that these are expected to already exist in order to apply the rule r on a match m on the triple graph. (Triple) rules are applied by constructing a *pushout*, which can be viewed as a generalised union of (triple) graphs R and G over a common sub-(triple) graph L:

Definition 3.5 (Triple Rule Application)**.**
*A direct derivation $G \overset{r@m}{\Longrightarrow} G'$ via a triple rule r, is constructed as depicted to the right by building a pushout over r and a triple monomorphism $m : L \to G$ called a match. A **derivation** $D : G \overset{*}{\Longrightarrow} G_n = G \overset{r_1@m_1}{\Longrightarrow} G_1 \overset{r_2@m_2}{\Longrightarrow} \ldots \overset{r_n@m_n}{\Longrightarrow} G_n$ is a sequence of direct derivations. We denote by $\mathcal{D} = \{d_1, \ldots, d_n\}$ the underlying set of direct derivations included in D.*



By means of pushout construction, a new triple graph G' is formed that shows the triple graph instance after rule application. In Fig. 3.12, the pushout construction is illustrated with a second application of *PortToVariable* on the instance of Fig. 3.7. Attributes and typing information are omitted to improve readability. The specification of pushouts ensures that G' contains only one statemachine `sm`, i. e., the nodes of G and R are mapped to the same node in G', whereas this does not hold for the ports `p1` of G and `p` of R, such that two distinct ports exist in G'. For further details, the interested reader is referred to Ehrig et al. [EEPT06].

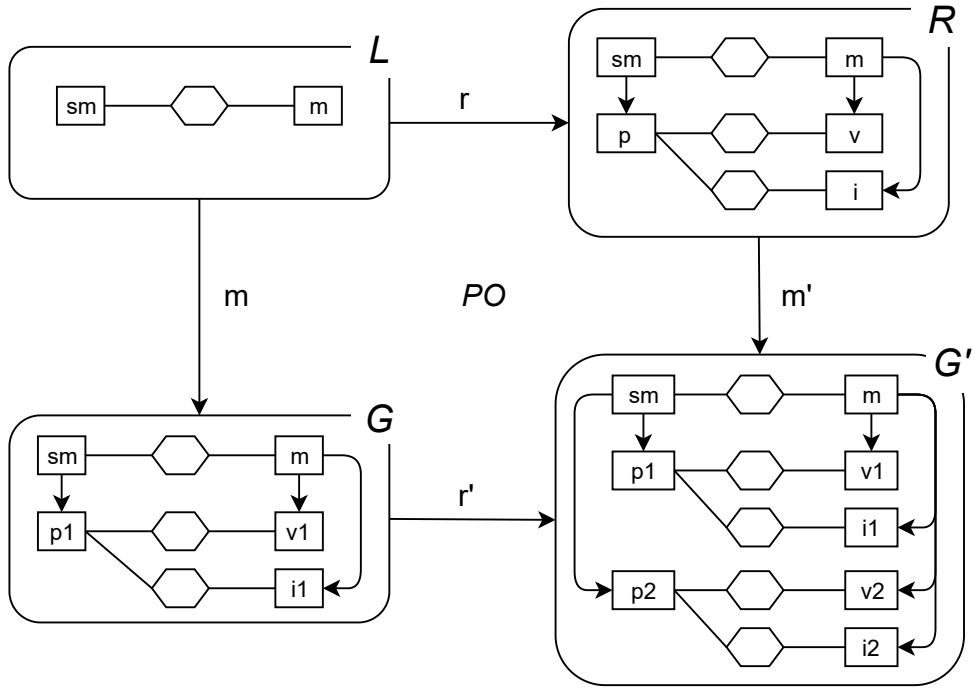


Figure 3.12: Second application of PortToVariable on the instance of Fig. 3.7

Each TGG consists of a finite set of rules, which operate on all three graphs at the same time and extend them by creating new nodes and edges in case their required context – i. e., the left-hand side of the respective rule – already exists. Each triple graph that can be generated by finitely many non-deterministic rule applications on an empty triple graph is contained in the language of the TGG. While these TGG rules are of declarative nature, they can be operationalised to perform consistency management tasks like forward and backward transformation, model synchronisation or consistency checking, which will be discussed in Chap. 5 in detail.

Definition 3.6 (Triple Graph Grammar)**.**
*A **triple graph grammar** $TGG = (TG, \mathcal{R})$ consists of a type triple graph TG, and a finite set $\mathcal{R}$ of triple rules.*

The triple metamodel of Fig. 3.8 together with the rules *PortToVariable* (Fig. 3.10) and *StatemachineToMachine* (Fig. 3.11) form a small TGG, which will be extended throughout this thesis, and then used to demonstrate the fault-tolerant approach to consistency management. To formally define a consistency relation, a TGG describes a *language* of triple graphs in a rule-based manner. The language of triples generated by the rules of a TGG is defined in terms of derivations, i. e., sequences of triple rule applications (cf. Def. 3.5) starting from an empty triple graph $G_\emptyset$ which contains neither nodes nor edges.

Definition 3.7 (Language of a Triple Graph Grammar)**.**
Let $TGG = (TG, \mathcal{R})$ be a ***triple graph grammar****. The* ***triple graph language*** *of TGG is defined as $L(TGG) = \{G_\emptyset\} \cup \{G \mid \exists\ D\ : G_\emptyset \overset{*}{\Longrightarrow} G\}$, where $G_\emptyset$ is the* ***empty triple graph****.*

To check whether a triple graph such as the instance of Fig. 3.7 is contained in the language of the example TGG in its current state, one has to find an application sequence that generates the instance using the rules *PortToVariable* and *StatemachineToMachine*. The starting point is the empty triple graph $G_\emptyset$, such that the rule *PortToVariable* is not applicable because the required context elements do not exist yet. The rule *StatemachineToMachine*, in contrast, can be used in the first place as this rule solely consists of green elements, i. e., no context elements are required. In a second step, *PortToVariable* can be applied because the first rule application created the required statemachine `sm`, the machine `m`, and the correspondence node. The sequential application of the two rules in the correct order results in the triple graph instance shown in Fig. 3.7, which means that it is contained in the language of the TGG.

3.4 Summary and Discussion

In this chapter, we have seen how basic behavioural modelling is accomplished using the SysML, a de-facto standard in the area of systems engineering. As a second language, Event-B is introduced, providing a formal semantics that makes it possible to guarantee that global constraints are always satisfied. These constraints are usually specified in form of invariants or theorems, which do not exist in SysML. Only small subsets of both languages have been introduced to keep the example as small and comprehensible as possible.

Furthermore, TGGs as a declarative, rule-based approach to bidirectional model transformations were introduced. With help of TGGs, it is possible to specify a consistency relation between two languages, such as SysML and Event-B in the running example of this thesis. A TGG consists of a typed triple graph – which represents the (triple) metamodel – and a set of rules. All triples that can be generated via finitely many rule applications form a language that ultimately defines the consistency relation.

Without additional language features, the formalism is insufficient to reasonably describe fault-tolerant consistency management, though, which is one of the ten requirements listed in Sect. 1.2. For instance, the previously presented rules are not expressive enough to specify constraints, neither for attribute values, nor for particular patterns that must (not) occur in the model instances. Therefore, the TGG formalism is extended by further language features in Chap. 4: With help of attribute and application conditions, the applicability of rules can be restricted, whereas multi-amalgamation enables us to define for-each-like structures for TGG rule applications. It is further shown that these features indeed increase the expressive power of the formalism and therefore raise the potential of applying the subsequently presented consistency management framework in practice.

4 A Feature-Based Classification of Triple Graph Grammar Variants

As motivated in Sect. 3.4, the basic form of TGGs is not expressive enough to solve consistency management problems in practice. Based on the original specification, various TGG variants were developed that support additional language features, such as attribute conditions [AVS12,LHGO12], (negative) application conditions [GEH11], or multi-amalgamation [LAST17]. These language features pose further conditions on the applicability of rules aiming at increasing the *expressiveness* of the formalism. Some extensions of the basic approach just make the specification of rules easier and more user-friendly, while others increase the expressiveness of TGGs allowing the specification of consistency relations that are not possible without this feature. Therefore, it is important to know about the expressive power of language features to, e. g., choose a tool that is expressive enough for the application scenario at hand (cf. Sect. 1.1).

In this chapter, we provide an overview of the most common language features of TGGs. Based on this, we discuss different TGG variants formally and develop a classification of TGG language features with respect to their expressiveness. We evaluate whether certain language features increase the expressiveness of TGGs or just improve the usability and simplify the specification. While the expressiveness of language features was partially investigated in previous work already, we will systematically analyse selected features, and aggregate the findings to a feature model. More specifically, this chapter shall address the following questions:

- How can expressiveness be defined for different TGG variants?
- Which language features are there, and how can they be used to classify TGG approaches and tools?
- Why do the respective language features (not) increase the expressiveness, compared to the original specification of TGGs?

The remainder of this chapter is organized as follows: Section 4.1 demonstrates that existing work concentrates on single TGG language features and their expressiveness in isolation, before they are used to create an overview in form of a a feature model in Sect. 4.2. Subsequently, selected language features are presented and compared with respect to their expressiveness: Besides basic rules (Sect. 4.3), which comply with the TGG definition of Sect. 3.3, attribute conditions (Sect. 4.4), application conditions (Sect. 4.5), and multi-amalgamation (Sect. 4.6) are introduced. Section 4.7 completes the rule set for the example transformation from SysML to Event-B, before Sect. 4.8 sums up the main findings and connects them to the superordinate topic of fault-tolerance.

4.1 Existing Work on TGG Language Features

There exist various survey papers which provide an overview of foundational results on TGGs. Schürr and Klar name consistency, completeness, efficiency and expressiveness as

desirable properties in their roadmap for future research on TGGs [SK08]. In subsequent work [ALK+15], concurrency and fault-tolerance were added as further dimensions, underpinning that these aspects are topics of interest in recent MDE research. Regarding expressiveness in particular, an overview of TGG language features is given by Kindler and Wagner [KW07], while features are only described on an intuitive level. A precise definition of expressiveness is lacking as well as an analysis of features increasing the expressiveness. Additionally, selected TGG tools are compared with respect to several criteria of interest. The TGG tools MoTE [GHL14], TGG Interpreter [GPR11] and eMoflon::TiE [LAS14a] are compared with respect to usability, expressiveness and other formal properties in the context of forward and backward transformations [HLG+13]. This survey was extended towards comparing the tools with respect to incremental updates [LAS+14b]. However, only three selected tools are considered, and the supported language features are not discussed with respect to expressiveness.

For being able to compare TGG tools and other approaches to BX, there have been continuous efforts to establish benchmark examples. Fundamental concepts and a list of requirements for benchmark examples in the context of BX were presented by Czarnecki et al. [CFH+09]. Based on these requirements, a repository for BX examples was created and continuously extended[1], such that it was possible to compare BX tools independent of the underlying formalisms [ACG+14, ADJ+17, ABW+20]. Although being an important step towards comparing the power of TGG approaches, the rather tool-driven concepts do not contain a formal underpinning of expressiveness.

The particular language features were intensively studied for algebraic graph transformation and transferred to the TGG context in most cases as well. Negative application conditions for triple rules were initially defined by Ehrig et al. [EHS09], providing proofs of correctness and completeness alongside. Requirements for propagating constraints from source to target is discussed in subsequent work [EET11], while an exemplary integration of those into the TGG tool MoTE is presented by Hildebrandt et al. [HLBG12]. A formal specification for nested application conditions for TGG and a proof for increased expressiveness is proposed by Golas et al. [GEH11]. Attribute conditions for TGGs were introduced by Lambers et al. [LHGO12], while challenges and solutions for operational attributed TGG rules were presented as well. An extension towards complex attribute manipulation compatible with existing TGG formalizations was made in subsequent work by Anjorin et al. [AVS12]. While amalgamation of two rules was already studied for decades [BFH87], multi-amalgamation was recently introduced by Golas et al. [GEH10] and combined with application conditions in subsequent work [GHE14]. The concepts were transferred to TGGs later on by Leblebici et al. [LAST15] and enriched by a practical implementation within the TGG tool eMoflon::TiE [LAS15]. In an extended version, it was shown that this language feature increases the expressive power of TGGs [LAST17]. All these papers rather focus on a detailed description of a single language feature instead of comparing the expressiveness of different features. Furthermore, some language features introduced in this chapter (basic rules affecting only one model, attribute conditions that are restricted to checking for equality, application conditions generated from constraints) are not dealt with in related work.

A TGG language feature which is not considered in this thesis is rule refinement [ASLS14], specifying an inheritance relation between rules that share a common part. This common part is expressed using abstract rules, which cannot be applied themselves but form the basis for concrete rules, that also incorporate the differences. The main advantage of rule refinement is to avoid redundancies in the rule base, keeping it concise and understandable.

[1] http://bx-community.wikidot.com/examples:home

This concept is not only applicable to TGGs but constitutes an important issue context of MDE. An overview of tool support for inheritance among rules is provided by Wimmer et al. [WKK+11, WKK+12].

While BX is restricted to maintaining consistency for two models, the concepts are generalised for arbitrarily many models in *multi-directional transformations*. A considerable amount of conceptual work was recently proposed by Trollmann et al. [TA16, TA17], Klare et al. [KG19, Kla21] and Stünkel et al. [SKLR20, SKLR21] in this regard, but in the scope of this thesis, we will restrict ourselves to the binary case. Consistency relations between more than two models will be split up into binary relations for each model pair.

4.2 Feature Model and Expressiveness

In Sect. 3.3, only the *basic* form of TGGs is described, whereas there exist multiple extensions to this formalism. In this chapter, different TGG variants are described by extracting a minimum set of possible rules as "basic rules" and identifying extensions to those basic rules. The language features are taken from the most recent research roadmap for TGGs [ALS15], in which expressiveness is listed as one of five research dimensions and related to the language features a TGG-based approach or tool supports.

Figure 4.1 shows a feature model containing the language features discussed in this chapter. Some subfeatures (e. g., application conditions generated from constraints) are left out for clarity in the feature model, but will be dealt with in the respective sections. Each language feature is presented with a description and an example, followed by a formal definition of the feature and a discussion whether the language feature increases the expressiveness of the respective TGG variant.

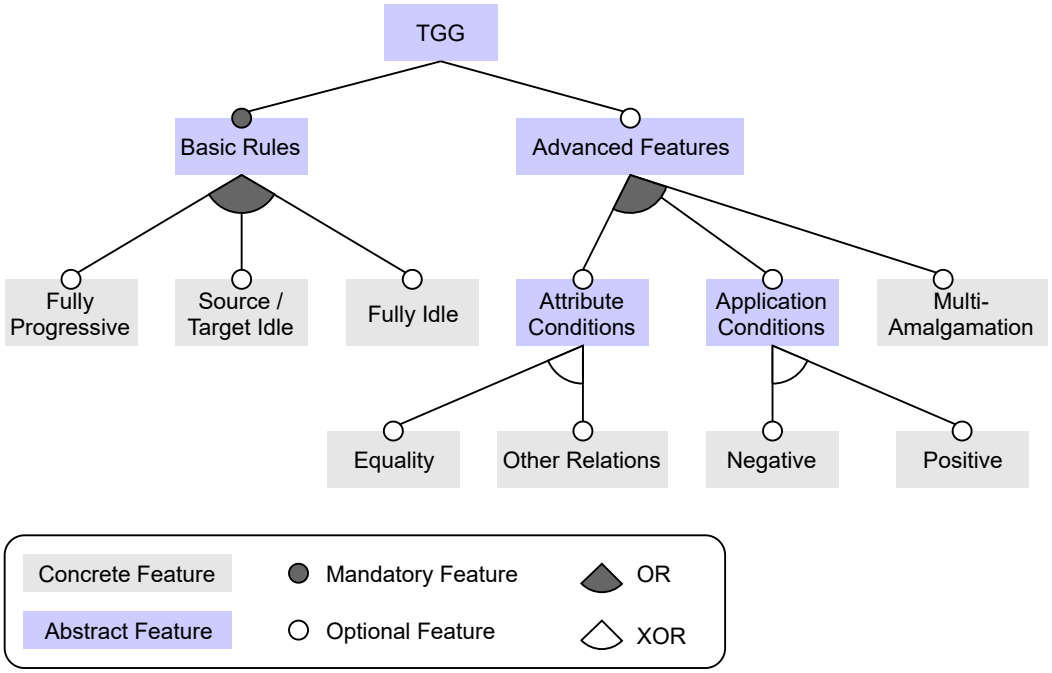


Figure 4.1: Classification of TGG variants as a feature model

In order to compare different language features, it is of major importance to define the meaning of *expressiveness* for a TGG variant, i. e., a set of TGGs that use a certain language feature. We denote the TGG variant using such a feature as *more expressive* than

the basic form of TGGs, if it enables us to form languages that cannot be specified only with basic rules. More formally, the expressiveness of sets of TGGs (i. e., TGG variants) can be defined as follows:

Definition 4.1 (Expressiveness of sets of Triple Graph Grammars)**.**
Let $\mathcal{TGG}_1$ and $\mathcal{TGG}_2$ be sets of TGGs.

- *$\mathcal{TGG}_2$ is at least as expressive as $\mathcal{TGG}_1$ ($\mathcal{TGG}_2 \geq_E \mathcal{TGG}_1$) iff $\forall\ TGG_1 \in \mathcal{TGG}_1$ $\exists\ TGG_2 \in \mathcal{TGG}_2 : L(TGG_1) = L(TGG_2)$.*
- *$\mathcal{TGG}_1$ and $\mathcal{TGG}_2$ are equally expressive ($\mathcal{TGG}_1 =_E \mathcal{TGG}_2$) iff $\mathcal{TGG}_1 \geq_E \mathcal{TGG}_2$ and $\mathcal{TGG}_2 \geq_E \mathcal{TGG}_1$.*
- *$\mathcal{TGG}_2$ is more expressive than $\mathcal{TGG}_1$ ($\mathcal{TGG}_2 >_E \mathcal{TGG}_1$) iff $\mathcal{TGG}_2 \geq_E \mathcal{TGG}_1$ and not $\mathcal{TGG}_1 =_E \mathcal{TGG}_2$.*
- *In all other cases, $\mathcal{TGG}_1$ and $\mathcal{TGG}_2$ are incomparable ($\mathcal{TGG}_1 \neq_E \mathcal{TGG}_2$).*

Intuitively, a TGG variant $\mathcal{TGG}_2$ is at least as expressive as $\mathcal{TGG}_1$, if it is possible for each $TGG_1 \in \mathcal{TGG}_1$ to find a $TGG_2 \in \mathcal{TGG}_2$ that generates the same language. Based on this notion, we will investigate whether a language feature is able to increase the expressiveness and, therefore, enriches the possibilities of meta-tool developers when including this language feature into a TGG-based consistency management tool.

4.3 Basic Rules

To examine if certain language features increase the expressiveness, the set of TGGs that only use *basic rules* is introduced as a reference point. The previously introduced rules *StatemachineToMachine* and *PortToVariable* (cf. Fig. 3.11 and 3.10) are examples for basic rules. The two rules add elements to both models, which is not enforced by the definition of a triple rule (Def. 3.4), though. It is, therefore, worthwhile to investigate whether it makes a difference with respect to expressiveness if basic rules can be restricted to always affect both source and target models. As an example for a rule which only affects a single model, we consider the rule *AddRegion* (Fig. 4.2). It creates a new region `r` in the SysML model, while nothing changes in the Event-B model, as there is no corresponding language construct in the target language.

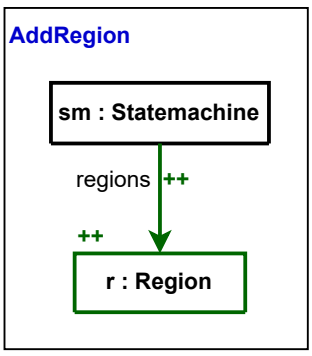


Figure 4.2: Rule: AddRegion

Formal Definition

Let $\mathcal{TGG}_{\text{Basic}}$ denote the set of all TGGs that only use basic rules. This set may be further divided with respect to the effect that the rule has on source and target graph.

Definition 4.2 (Basic Rules)**.**
Let $r : L \to R, L = L_S \xleftarrow{\gamma_S} L_C \xrightarrow{\gamma_T} L_T, R = R_S \xleftarrow{\gamma_S} R_C \xrightarrow{\gamma_T} R_T$ be a triple rule. Define:

- *Fully progressive rules:* $\mathcal{R}^{=} := \{r \mid L_S \neq R_S \wedge L_T \neq R_T\}$
- *Source-idle rules:* $\mathcal{R}^{S} := \{r \mid L_S = R_S \wedge L_T \neq R_T\}$
- *Target-idle rules:* $\mathcal{R}^{T} := \{r \mid L_S \neq R_S \wedge L_T = R_T\}$
- *Fully idle:* $\mathcal{R}^{\emptyset} := \{r \mid L_S = R_S \wedge L_T = R_T\}$

Based on Def. 4.2, the TGG variant for basic rules can be defined:

Definition 4.3 (TGG Variant: Basic Rules)**.**
Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar. The following TGG variants are defined:

- $\mathcal{TGG}^{=} := \{TGG \mid \mathcal{R} \subseteq \mathcal{R}^{=}\}$
- $\mathcal{TGG}^{S} := \{TGG \mid \mathcal{R} \subseteq \mathcal{R}^{=} \cup \mathcal{R}^{S}\}$
- $\mathcal{TGG}^{T} := \{TGG \mid \mathcal{R} \subseteq \mathcal{R}^{=} \cup \mathcal{R}^{T}\}$
- $\mathcal{TGG}^{*} := \{TGG \mid \mathcal{R} \subseteq \mathcal{R}^{=} \cup \mathcal{R}^{S} \cup \mathcal{R}^{T}\}$

$\mathcal{TGG}^{*}$ is also referred to as $\mathcal{TGG}_{Basic}$.

As $\mathcal{TGG}^{S}$ and $\mathcal{TGG}^{T}$ are symmetric, only $\mathcal{TGG}^{=}$ and $\mathcal{TGG}^{*}$ need to be compared with respect to expressiveness to determine whether using rules from $\mathcal{R}^{S}$ or $\mathcal{R}^{T}$ increases the expressiveness compared to using only rules from $\mathcal{R}^{=}$.

According to Def. 4.3, the rules *StatemachineToMachine* and *PortToVariable* are contained in $\mathcal{R}^{=}$, as they add elements to both the SysML and the Event-B model. Therefore, a TGG consisting of these two rules is an example for a TGG in $\mathcal{TGG}^{=}$. The rule *AddRegion*, in contrast, is contained in $\mathcal{R}^{S}$, since the rule application only adds elements to the SysML model, while the Event-B model remains unchanged. Adding this rule would create a TGG in $\mathcal{TGG}^{*} \setminus \mathcal{TGG}^{=}$.

Expressiveness

First of all, one can examine these two sets of TGGs with respect to the size of graphs they can generate.

Definition 4.4 (Size of a Graph)**.**
Let $G = (V, E, src, trg)$ be a graph. The size of G is defined as the sum of vertices and edges contained in G: $|G| = |V| + |E|$.

Lemma 4.1. *Let $TGG = (TG, \mathcal{R}) \in \mathcal{TGG}^{=}$ and let $G_S \xleftarrow{\gamma_S} G_C \xrightarrow{\gamma_T} G_T$ be a triple graph. Let $G_S \xleftarrow{\gamma_S} G_C \xrightarrow{\gamma_T} G_T$ be generated by a sequence $G_\emptyset \overset{r_1}{\Rightarrow} \ldots \overset{r_n}{\Rightarrow} G_S \xleftarrow{\gamma_S} G_C \xrightarrow{\gamma_T} G_T, r_1 \ldots r_n \in \mathcal{R}$ of direct derivations from the initial triple graph $G_\emptyset$. The ratio of the sizes of the source graph G_S and the target graph G_T is bounded by a constants $c, d \in \mathbb{Q}^{+}$ after at least one rule application: $c \leq \dfrac{|G_T|}{|G_S|} \leq d$.*

Proof. For each rule $r_i \in \mathcal{R}$, let s_j be the number of elements added to G_S and t_i be the number of elements added to G_T when applying r_i. Set c := $\frac{min_{1\leq i\leq n}\{t_i\}}{max_{1\leq j\leq n}\{s_j\}}$ and d := $\frac{max_{1\leq i\leq n}\{t_i\}}{min_{1\leq j\leq n}\{s_j\}}$. Since $s_j > 0$ and $t_i > 0$ by definition of $\mathcal{TGG}^=$, c and d are well-defined.

$$c = \frac{min_{1\leq i\leq n}\{t_i\}}{max_{1\leq j\leq n}\{s_j\}} \leq \frac{\sum_{i=1}^{n} t_i}{\sum_{j=1}^{n} s_j} = \frac{|G_T|}{|G_S|} \leq \frac{max_{1\leq i\leq n}\{t_i\}}{min_{1\leq j\leq n}\{s_j\}} = d$$

□

In Fig. 3.7, the triple graph resulting from rule applications of *StatemachineToMachine* and *PortToVariable* is depicted. The left hand side shows the SysML model (G_S), the right hand side the Event-B model (G_T). G_S has a size of 5, as well as G_T. Further applications of *PortToVariable* would increase the size G_S by 2 and G_T by 4, whereas applying *StatemachineToMachine* would increase G_S by 3 and G_T by 1. Therefore, $\frac{1}{3} \leq \frac{|G_T|}{|G_S|} \leq 2$ will always hold. In general, TGGs in $\mathcal{TGG}^=$ can produce graphs of unequal sizes, but it is always possible to find a lower and an upper bound for the ratio of the two graph sizes.

Lemma 4.2. *Let $TGG = (G_\emptyset, \mathcal{R}) \in \mathcal{TGG}^*$ and let $G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T$ be a triple graph. Let $G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T$ be generated by a sequence $G_\emptyset \overset{r_1}{\Rightarrow} \ldots \overset{r_n}{\Rightarrow} G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T, r_1 \ldots r_n \in \mathcal{R}$ of direct derivations from the empty triple graph $G_\emptyset$. The ratio of the sizes of the source graph G_S and the target graph G_T is unbounded.*

Proof. Let $r \in \mathcal{R}$ be an applicable rule that adds elements only to G_T. The repeated application of r on $G_S \overset{\sigma}{\leftarrow} G_C \overset{\tau}{\rightarrow} G_T$ increases the ratio $\frac{|G_T|}{|G_S|}$ in each step, because $|G_T|$ gets arbitrarily large while $|G_S|$ remains constant. Due to symmetry of $\mathcal{TGG}^S$ and $\mathcal{TGG}^T$, the same holds vice versa. □

Let us consider a TGG that also contains the rule *AddRegion*. As an application of this rule only affects the SysML model, it can grow arbitrarily large when applying *AddRegion* often enough, while the Event-B model stays the same for all rule applications. Demanding rules to add elements to both source and target graphs decreases the expressive power of a set of TGGs, as shown in Thm. 4.1.

Theorem 4.1 (Expressiveness of $\mathcal{TGG}^*$)**.** $\mathcal{TGG}^* >_E \mathcal{TGG}^=$

Proof. $\mathcal{TGG}^* \geq_E \mathcal{TGG}^=$ holds trivially, because $\mathcal{TGG}^= \subseteq \mathcal{TGG}^*$. It remains to show: $\mathcal{TGG}^* \neq_E \mathcal{TGG}^=$. Let $TGG_1 = (TG, \mathcal{R}) \in \mathcal{TGG}^=$ and let $G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T$ be a triple graph. Let $G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T$ be generated by a sequence $G_\emptyset \overset{r_1}{\Rightarrow} \ldots \overset{r_n}{\Rightarrow} G_S \overset{\gamma_S}{\leftarrow} G_C \overset{\gamma_T}{\rightarrow} G_T, r_1 \ldots r_n \in \mathcal{R}$ of direct derivations from the initial triple graph $G_\emptyset$. According to Lemma 4.1, the ratio of the sizes of the source graph G_S and the target graph G_T is bounded by constants c and d.
Let $G_S^* \overset{\gamma_S}{\leftarrow} G_C^* \overset{\gamma_T}{\rightarrow} G_T^*$ be a triple graph with $\frac{|G_T^*|}{|G_S^*|} = k > d$. $G_S^* \overset{\gamma_S}{\leftarrow} G_C^* \overset{\gamma_T}{\rightarrow} G_T^*$ cannot be generated by TGG_1. Let $TGG_2 = (G_\emptyset, \mathcal{R} \cup r^*) \in \mathcal{TGG}^*$, whereby r^* is a rule that adds elements only to the target graph G_T. $G_S^* \overset{\gamma_S}{\leftarrow} G_C^* \overset{\gamma_T}{\rightarrow} G_T^*$ can be generated by TGG_2, though, because every application of r^* strictly increases the ratio $\frac{|G_T^*|}{|G_S^*|}$. □

4.4 Attribute Conditions

Attribute conditions specify the consistency relation between attributes in source and target model (cf. [AVS12, GLO09, LHGO12]). In the running example, it seems to be reasonable to keep the names of corresponding language constructs equal, such that, e. g., the SysML statemachine's name attribute (`sm.name`) and the respective attribute of the Event-B machine (`m.name`) have the same value, which is "Machine" in example instance of Fig. 3.7). To include this requirement into the TGG rule base, the rule *StatemachineToMachine* (Fig. 3.11) is enhanced with an attribute condition that uses the equals operator:

```
sm.name = m.name
```

Often, however, checking for equality is not sufficient for expressing the consistency relation between attributes precisely. When considering the attribute values of the SysML port `p` and the corresponding variable `v` and invariant `i` in Event-B, one can see that the `p`'s name is used to form the two attribute values of `i` by *string concatenation*. More concretely, the following three attribute conditions must be added to the rule *PortToVariable* (Fig. 3.10):

```
p.name = v.name
'TYPEOF_ '+ p.name = i.name
p.name + '∈ BOOL '= i.predicate
```

Attribute conditions are usually placed beneath the TGG rule they refer to in a textual form. The rule *StateToVariable* (cf. Fig. 4.3) adds a state `s` to the SysML model and links it to a new variable `v` and a new invariant `i` of the Event-B code. Furthermore, three attribute conditions specify how the attribute values are formed. Apart from the typing of the source model, the rules *StateToVariable* and *PortToVariable* are identical.

Formal Definition

Attribute conditions are formally defined by Ehrig et al. [EEPT06] for algebraic graph transformations and Anjorin et al. [AVS12] for TGGs, which we refer to for further details. To deal with attribute values, an extension of graphs (cf. Def. 3.1) to data graphs is needed, which introduce additional sets for data vertices and vertex attribute edges:

Definition 4.5 (Data Graph)**.**
A data graph $G = (V_G, E_G, src_G, trg_G, V_D, E_D, src_D, trg_D)$ *consists of the sets*

(1) V_G *and* V_D*, called the graph vertices and data vertices,*

(2) E_G *and* E_D*, called the graph edges and vertex attribute edges, and the source and target functions*

(3) $src_G : E_G \to V_G$*,* $trg_G : E_G \to V_G$ *for graph edges, and*

(4) $src_D : E_D \to V_G$*,* $trg_G : E_D \to V_D$ *for vertex attribute edges.*

Based on the notion of data graphs and the previously introduced attribute condition types, respective TGG variants can be introduced:

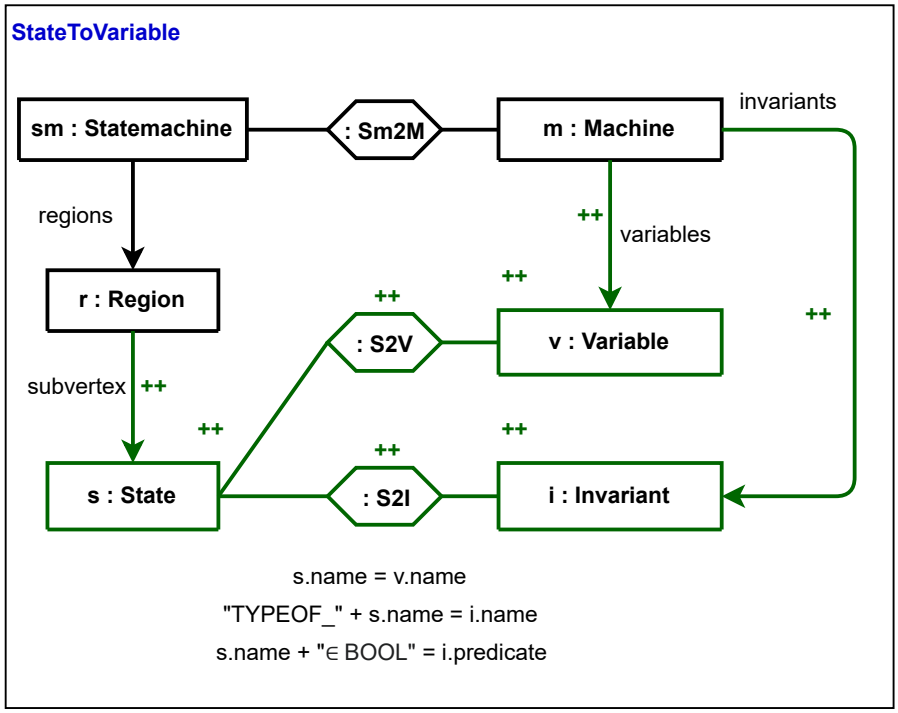


Figure 4.3: Rule: StateToVariable

Definition 4.6 (TGG Variant: Rules with Attribute Conditions)**.**

- *Let* $\mathcal{TGG}_{AttrC^{=}}$ *be the TGG variant supporting basic rules and attribute conditions using only the equals operator.*
- *Let* $\mathcal{TGG}_{AttrC^{*}}$ *be the TGG variant supporting basic rules and arbitrary attribute conditions.*
- *Let* $\mathcal{TGG}_{AttrC} = \mathcal{TGG}_{AttrC^{=}} \cup \mathcal{TGG}_{AttrC^{*}}$.

Expressiveness

Suppose that TGGs supporting attribute conditions are not more expressive compared to $\mathcal{TGG}_{\text{Basic}}$, then there must be a way to express attributes and their consistency relation in $\mathcal{TGG}_{\text{Basic}}$. Attributes can be represented as an association to objects of their type (e. g., `EString`) – this adjusts the representation of attributes in the metamodels[2] as shown in Fig. 4.4.

To improve readability, the correspondences are omitted in this visualisation. The multiplicities for all new associations are 0..1 (optional attributes) or 1 (required attributes). In the adjusted metamodels, the SysML and the Event-B model have common classes for all data types, which is only `EString` in this example. Each `EString` can be referenced as often as required.

Figure 4.5 shows an attribute value change for the instance depicted in Fig. 3.7 in the data graph representation. The added data vertices and vertex attribute edges (new attribute values) are depicted in green with a ++ mark-up, while the deleted data vertices

[2]Only parts of the metamodels which are necessary to describe the example instance of Fig. 3.7 are depicted, the conversion of the remaining attributes works similarly.

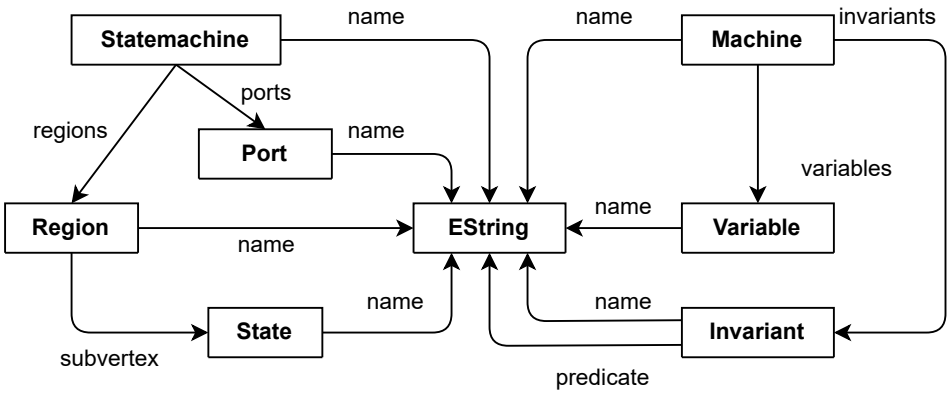


Figure 4.4: Partial metamodel with data vertices and vertex attribute edges

and vertex attribute edges (old attribute values) are coloured red and marked up with `--`. The name attribute of the SysML statemachine `sm` and the Event-B machine `m` were changed from "Machine" to "Statemachine". In this case, the consistency relation can be expressed with basic rules (i. e., the required TGG would still be in $\mathcal{TGG}_{\text{Basic}}$) as the attribute change is equal to deleting and creating nodes and edges of a graph. The vertex attribute edges simply point to the same `EString` object, as shown in Fig. 4.5.

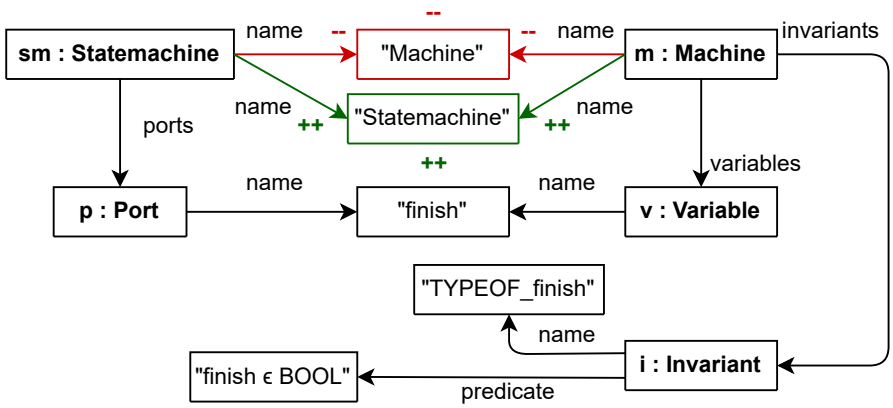


Figure 4.5: Changed attribute values (`equals` operator)

In Fig. 4.6, the situation is slightly different. The names of the port `p` in the SysML model and the corresponding variable `v` of the Event-B model were changed from "finish" to "end". This, however, makes it necessary to adjust the name and predicate of the corresponding invariant `i` as well, assigning the new values "TYPEOF_end" and "end $\in$ BOOL", respectively. With the chosen encoding of attributes as data vertices, this consistency relation cannot be modelled using only basic rules, as there is no way to define the relation between the `EString` data vertices. In contrast to the first example, one would need a way to create new `EString` data vertices for `i`'s attributes (or find existing data vertices with this value) and create an association to these data vertices.

Alternatively, all symbols of the alphabet could be encoded as separate data vertices, such that attribute values are represented by a linked list of these vertices. In this case, string concatenation can be simulated by connecting the respective linked lists with an additional edge. It is not possible to express arbitrary arithmetic operations with basic TGG rules, though, as stated in Thm. 4.2.

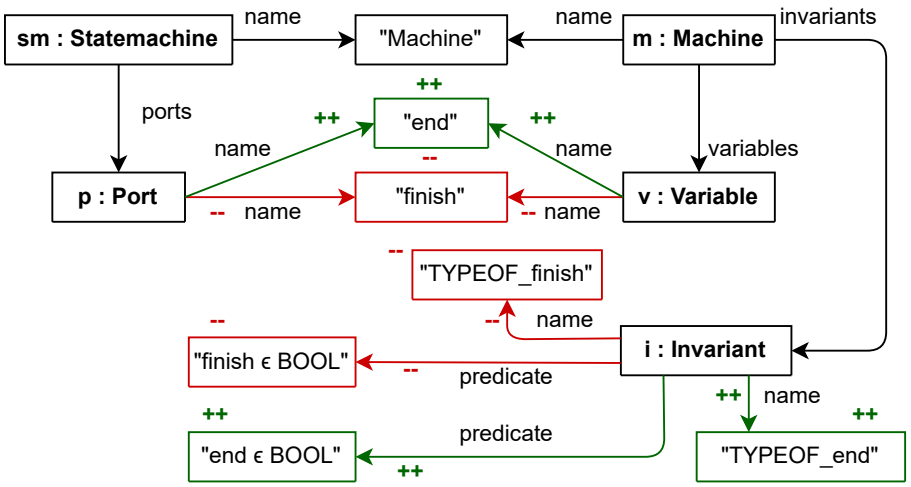


Figure 4.6: Changed attribute values (`concat` operator)

Theorem 4.2 (Expressiveness of Attribute Conditions)**.**

1. $\mathcal{TGG}_{AttrC^{=}} =_E \mathcal{TGG}_{Basic}$
2. $\mathcal{TGG}_{AttrC^{*}} >_E \mathcal{TGG}_{Basic}$
3. $\mathcal{TGG}_{AttrC} =_E \mathcal{TGG}_{AttrC^{*}}$

Proof.

1. To show $\mathcal{TGG}_{\text{AttrC}^{=}} =_E \mathcal{TGG}_{\text{Basic}}$, one must find a way to specify the equality of attributes using just basic rules. As shown in Fig. 4.5, this can be done by referencing the same data vertex in both models. If the attribute changes, the reference changes accordingly.

2. To show $\mathcal{TGG}_{\text{AttrC}^{*}} >_E \mathcal{TGG}_{\text{Basic}}$, several existing graph-theoretical results have to be combined. Courcelle and Engelfriet have shown that the expressive power of graph grammars ($\mathcal{GG}$), which are more expressive than basic TGGs as they allow to delete elements, is equivalent to monadic second-order logic ($\mathcal{MSO}$) [Cou90, CE12]. For graphs with bounded tree depth, first-order logic ($\mathcal{FO}$) and $\mathcal{MSO}$ are equivalent according to Elberfeld et al. [EGT16]. Together with Schweikardt's results on $\mathcal{FO}$ [Sch05], i. e., that using arithmetic operators increases the expressive power ($\mathcal{FO}(+,\times)$), the following statement about the expressiveness of arbitrary attribute conditions can be made:

   $$\mathcal{TGG}_{\text{AttrC}^{*}} \geq_E \mathcal{FO}(+,\times) >_E \mathcal{FO} = \mathcal{MSO} = \mathcal{GG} >_E \mathcal{TGG}_{\text{Basic}}$$

   It is important to note that the tree depth of graphs is only bounded for TGGs with acyclic type graphs; thus, in general, the precondition for using the result of Elberfeld et al. [EGT16] is not fulfilled. For TGGs with cyclic type graphs, the proof whether attribute conditions increase the expressive power is left to future work.

3. $\mathcal{TGG}_{\text{AttrC}} =_E \mathcal{TGG}_{\text{AttrC}^{*}}$ follows directly from Def. 4.6: $\mathcal{TGG}_{\text{AttrC}} = \mathcal{TGG}_{\text{AttrC}^{=}} \cup \mathcal{TGG}_{\text{AttrC}^{*}}$

□

4.5 Application Conditions

Application conditions pose additional restrictions on rules, which have to be fulfilled in order to apply them. Such application conditions can be explicitly defined by the tool developer and integration expert, or automatically generated from constraints, such as multiplicity constraints of the metamodels, to ensure that these constraints hold after applying the rule. Negative Application Conditions (NACs) forbid rule applications if certain patterns exist already, while Positive Application Conditions (PACs) enforce that certain patterns exist before rule application.

Figure 4.7 shows an example for a PAC. The rule *TransitionToEvent* adds a transition `t` to the SysML model to connect two states later on, and links it to a corresponding event `e` in the Event-B model with the same name. The application condition ensures that all other transitions (`t2`) of the respective region `r` must already have a source and a target state (`s1` and `s2`). The blue elements with `!` mark-up stand for the *premise* of the PAC, while the violet elements marked up with `!!` represent the *conclusion*. A rule with a PAC is applicable if there is a match for the premise and for at least one of (possibly multiple) conclusions, or if the premise cannot be matched. It has the form of an implication: A match for the premise implies at least one match for a conclusion. Applied to the concrete example, the PAC would express the following condition:

"If the region already has one (or more) transition(s), the transition(s) must have outgoing source and target edges that connect the transition(s) to the respective state."

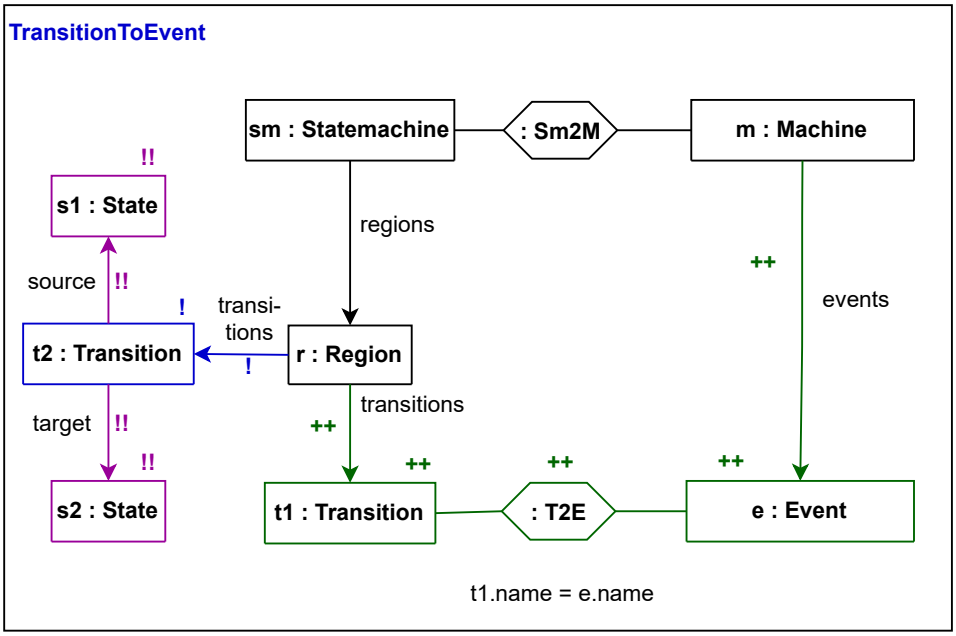


Figure 4.7: Rule: TransitionToEvent with a PAC

While PACs *allow* the application of a rule for certain graphs, NACs specify when a rule application is *forbidden*. The PAC in Fig. 4.7 could be transformed into a NAC by leaving out the conclusion (i. e., the violet nodes and edges). The resulting NAC would state that a transition `t` can only be added to a region `r` if the `r` does not have another transition `t2` before, which would restrict the number of transitions to 0 or 1. As this restriction is not reasonable in this context, two different rules are equipped with NACs in the following.

Up to here, there are no rules in the example TGG that can create the necessary source and target links that connect states and transitions. Therefore, a rule *SourceStateToEnterAction* (Fig. 4.8) is added to the TGG. In the SysML model, the rule links a transition `t` to a state `s1` of the same region `r` (both are required as context elements), and adds an action `a` and a guard `g` to the Event-B code. The attribute conditions give an indication about the purpose of the added Event-B language constructs: `a` expresses that the source state `s1` is left via `t`, setting the corresponding variable to `FALSE`, which we can assume to exist due to the structure of the rule *StateToVariable* (Fig. 4.3). The guard `g` ensures that this variable had the value `TRUE` before, i. e., that the statemachine assumed the state `s1` before. To restrict the rule's applicability, a NAC ensures that `t` does not have another source state `s2`. Without this condition, it would be possible to add arbitrarily many source states to a transition, which would lead to malformed SysML state machines.

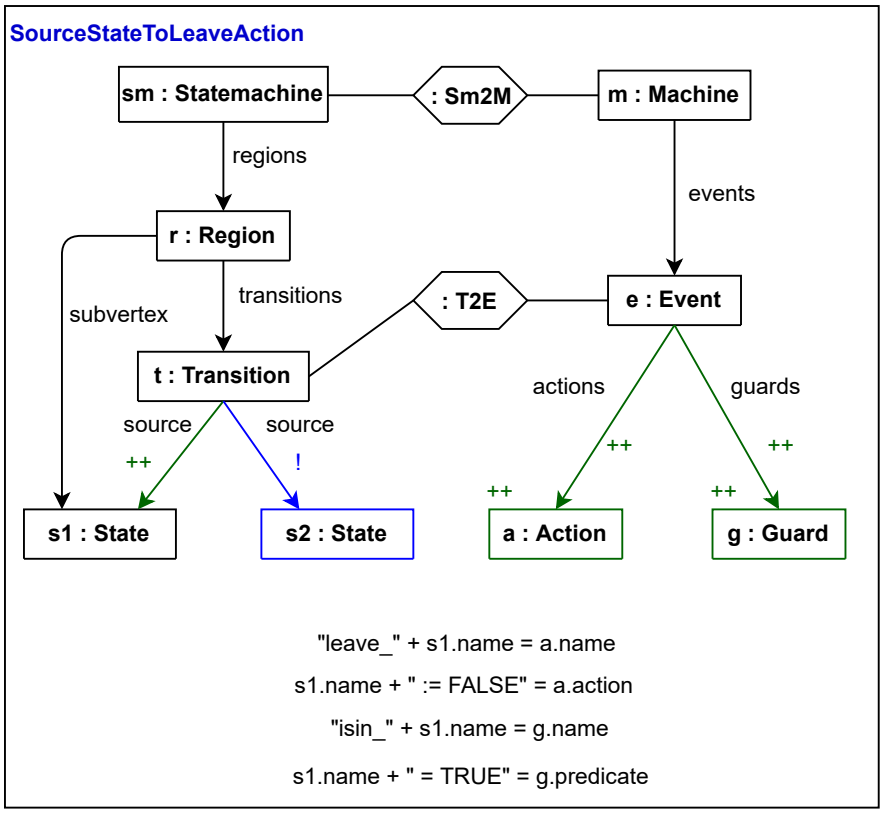


Figure 4.8: Rule: SourceStateToLeaveAction with a NAC

Likewise, the rule *TargetStateToEnterAction* connects a transition `t` to its respective target state `s1` (Fig. 4.9). Again, a NAC makes sure that there does not exist another target state `s2` before. Another action `a` is added to the event `e` in the Event-B code that sets the variable that corresponds to `s1` to `TRUE`. It is not necessary to specify another guard, because the condition for firing the transition `t` is sufficiently specified by the rule *SourceStateToEnterAction*.

Formal Definition

Application conditions are made up of two parts: the rule to be applied on the triple graph on the one hand and the condition that must hold for the graph on the other hand.

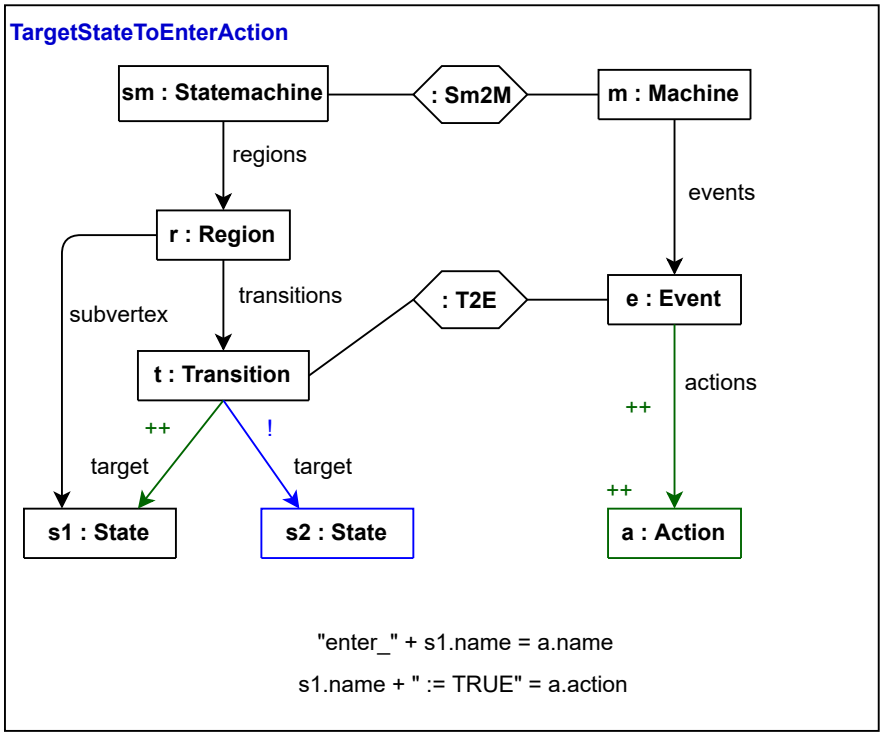


Figure 4.9: Rule: TargetStateToEnterAction with a NAC

Definition 4.7 (Graph Condition)**.**
A graph condition over a graph L is a pair $gc = (p : L \to P, \{c_i : P \to C_i \mid i \in I\})$, for some index set I. L is referred to as the context, P the premise, and $\{C_i \mid i \in I\}$ the conclusions of the graph condition gc.

A graph condition is either satisfied trivially, if there does not exist a match for the premise P, or if there exists at least one match for a conclusion C_i.

Definition 4.8 (Satisfaction of Graph Conditions)**.**
Let gc be a graph condition over L for some index set I, i. e., $gc = (p : L \to P, \{c_i : P \to C_i \mid i \in I\})$. An arrow $m : L \to G$ satisfies gc, denoted by $m \models gc$, iff $\forall\, m_p : P \to G, [(m = p; m_p) \Rightarrow (\exists\, i \in I\, \exists\, m_{c_i} : C_i \to G, [m_p = c_i; m_{c_i}])]$, where $m_p, (m_{c_i})_{i \in I}$ are monomorphisms.

Given a rule r, creating a graph condition for the left-hand side of the rule forms an application condition. For every match that can be found for the left-hand side in a concrete host graph, it can be determined if the rule is applicable.

Definition 4.9 (Application Condition)**.**
Given a monotonic rule $r : L \to R$, an application condition ac for r is a graph condition $(p : L \to P, \{c_i : P \to C_i \mid i \in I\})$ over L. A direct derivation $d = G \overset{r@m}{\Longrightarrow} G'$ with r at match m satisfies ac, denoted by $d \models ac$, iff $m \models ac$ according to Def. 4.8.

A NAC is a special case of an application condition, where the set of conclusions is empty. As a result, the rule is applicable iff the premise does not hold.

Definition 4.10 (Negative Application Condition)**.**
Given a monotonic rule $r : L \to R$, a NAC n for r is an application condition for r of the form $(n : L \to N, \{\})$.

Based on Def. 4.9 and 4.10, a new TGG variant for TGGs with rules that have application conditions can be defined.

Definition 4.11 (TGG Variant: Rules with Application Conditions)**.**

- *Let $\mathcal{TGG}_{PAC}$ be the TGG variant supporting basic rules and rules with a finite set of PACs.*
- *Let $\mathcal{TGG}_{NAC}$ be the TGG variant supporting basic rules and rules with a finite set of NACs.*
- *Let $\mathcal{TGG}_{AC} = \mathcal{TGG}_{PAC} \cup \mathcal{TGG}_{NAC}$.*

In the following, a subtype of application conditions that can be generated from constraints is introduced and analysed with respect to expressiveness. Constraints are a special type of graph conditions. They do not require any context, which means that everywhere the premise can be matched, one of the conclusions must hold.

Definition 4.12 (Graph Constraint)**.**
A graph constraint is a graph condition of the form $(p_\emptyset : G_\emptyset \to P, \{c_i : P \to C_i \mid i \in I\})$.

Similar to NACs, there are also negative constraints that can be demanded for a host graph. This means that there must not be a match for the premise to the host graph, the constraint is violated otherwise.

Definition 4.13 (Negative Constraint)**.**
A negative constraint is a graph condition of the form $(n_\emptyset : G_\emptyset \to N, I = \{\})$.

In the example shown in Fig. 4.10, the condition for the rule application has the form of a negative constraint, as the rule does not require any context elements. It creates a triple of a statemachine `sm1` and a machine `m1` corresponding to each other, unless at least one node of these types already exists, which is expressed by the blue SysML statemachine `sm2` and the blue Event-B machine `m2` outside the rule, representing the negative constraint. As a result, the existence of two statemachines in the SysML model or two machines in the Event-B model is forbidden.

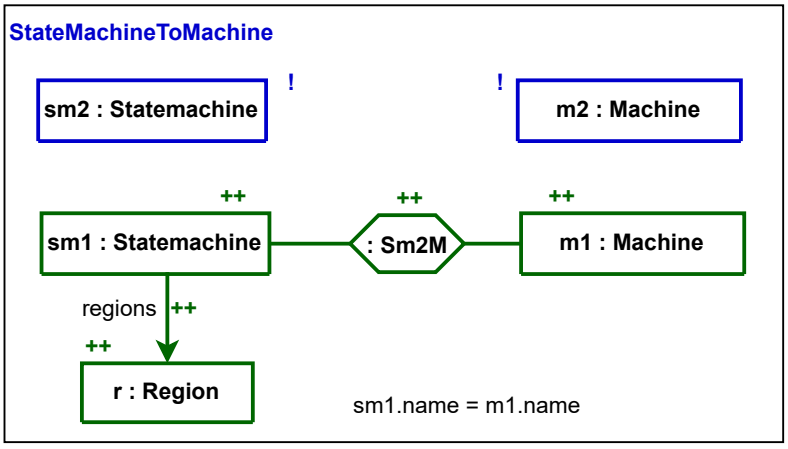


Figure 4.10: Rule: StatemachineToMachine with a negative constraint

As we have seen, it is possible to equip TGG rules with application conditions in such a way, that an application of this rule does not lead to a constraint violation. Due to

the minimality of this example, it was possible to formulate the application condition intuitively, which can get tedious and error-prone for more complex rules and constraints. The question that arises at this point is whether it is possible to generate such application conditions from constraints in a systematic way. It means that there must be a construction that transforms a constraint into an application condition, i. e., some algorithm that gets a constraint and a TGG rule as an input and outputs an equivalent application condition for that rule. More formally and in a broader context, Ehrig et al. state that the generation of application conditions from constraints is always possible as such [EEPT06, p. 157]:

Theorem 4.3 (Construction of Application Conditions from Graph Constraints)**.**
There is a construction Acc such that for every graph constraint c [. . .] and every graph R, Acc(c) is an application condition over R with the property that, for all morphisms n: $R \to H$, $n \models Acc(c) \Leftrightarrow H \models c$.

Proof. The theorem was proven by Ehrig et al. [EEPT06, p. 157-159]. Due to its complexity, the proof is not repeated here. □

$n \models Acc(c)$ means that the match for the right-hand side R of the rule to the graph H resulting from the rule application satisfies the generated application condition, while $H \models c$ means that the resulting graph satisfies the graph constraint. Note that Thm. 4.3 assumes that application conditions are evaluated for the right-hand side of a rule. However, Ehrig et al. also describe and prove a bidirectional transformation algorithm for left and right application conditions [EEPT06, pp. 160-162], while left application conditions are those which are dealt with in the scope of this thesis.

While Ehrig et al. have proven the *existence* of an algorithm for generating application conditions from constraints, Anjorin et al. describe and prove a *construction* for NACs from negative constraints [AST12], which is sketched by the diagram depicted in Fig. 4.11.

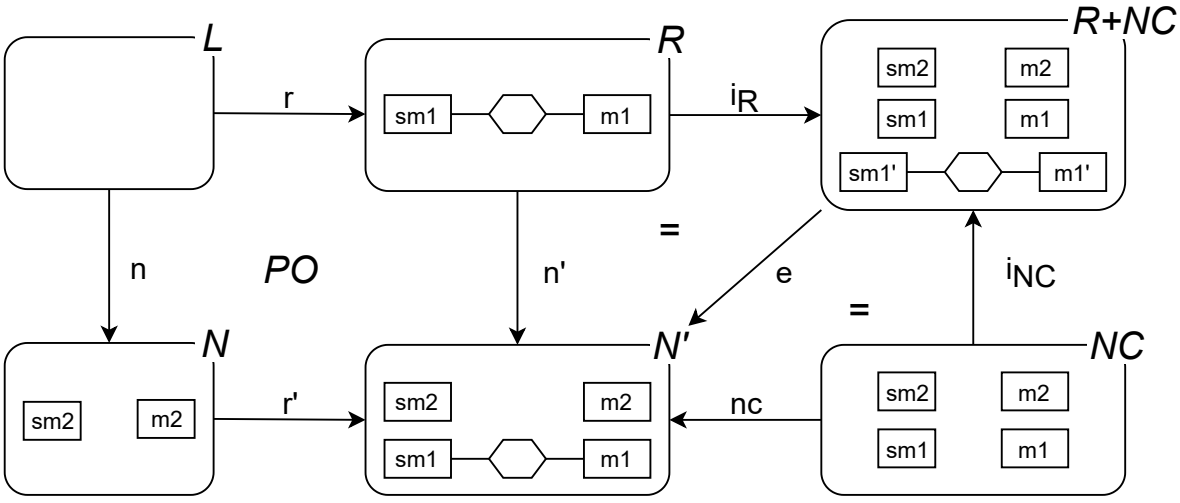


Figure 4.11: Construction algorithm for generating a NAC from a negative constraint

By this construction, the NAC for the example rule *StatemachineToMachine* (Fig. 4.10) can be generated from a negative constraint, forbidding two SysML statemachines (`sm1`, `sm2`) and Event-B machines (`m1`, `m2`), respectively (NC), as this subgraph represents an illegal state *after* rule application. L and R as left-hand and right-hand side of the rule are given, as well as the negative constraint NC and the corresponding arrows r and nc'. The goal is to find N and n that form the application condition together with the given rule.

First, the disjoint union $R + NC$ of R and NC is constructed. Second, R and N' form the graph condition for the target graph G', whereby e is an epimorphic arrow from $R + NC$ into N'. In the concrete example, the statemachines `sm1` and `sm1'`, and the machines `m1` and `m1'`, respectively, can be mapped to the same nodes in N'. Finally, N is constructed by pushout decomposition, such that $r : L \to R$ and $n : L \to N$ form the new rule *StatemachineToMachine* with a NAC. The algorithm yields an existing SysML statemachine `sm2` and an Event-B machine `m2` as application condition (N), which is exactly the case of Fig. 4.10. If N can be matched on the host graph G, i. e., if there exists a morphism $p : N \to G$, the rule is not applicable.

Expressiveness

It is proven that the use of NACs in addition to basic rules increases the expressive power of TGGs.

Theorem 4.4 (Expressiveness of NACs)**.**

$$\mathcal{TGG}_{NAC} >_E \mathcal{TGG}_{Basic}$$

Proof.

- $\mathcal{TGG}_{\mathbf{NAC}} \geq_E \mathcal{TGG}_{\mathbf{Basic}}$**:** This holds trivially because for every triple graph grammar $TGG \in \mathcal{TGG}_{\text{Basic}}$, it also holds that $TGG \in \mathcal{TGG}_{\text{NAC}}$ because all basic rules are allowed in $\mathcal{TGG}_{\text{NAC}}$. It follows that $\mathcal{TGG}_{\text{Basic}} \subseteq \mathcal{TGG}_{\text{NAC}}$.
- $\mathcal{TGG}_{\mathbf{NAC}} \neq_E \mathcal{TGG}_{\mathbf{Basic}}$**:** To show that the two sets of TGGs are not equal, consider a triple graph grammar $TGG_{NAC} = (TG, R_{NAC})$, whereby TG is a type graph as shown in Fig. 3.8 and R_{NAC} only contains the rule depicted in Fig. 4.10. $L(TGG_{NAC})$ contains only the empty triple graph and a triple graph consisting of exactly one SysML statemachine with a region, one Event-B machine, and a correspondence link. The graph condition forbids further rule applications if either of these nodes already exist. $L(TGG_{NAC})$ cannot be generated using only basic rules:
  - Let us assume that there exists $TGG_{Basic} = (TG, R_{Basic}) \in \mathcal{TGG}_{\text{Basic}}$ with $L(TGG_{Basic}) = L(TGG_{NAC})$
  - A triple containing one SysML statemachine and a corresponding Event-B machine is contained in $L(TGG_{Basic})$, so there must be a rule $r \in R_{Basic}$ to create the triple.
  - r cannot require any context because it must be applicable on $G_\emptyset$.
  - Since r does not require any context, it is always applicable. This means that arbitrarily many statemachines and machines can be generated, which contradicts the assumption $L(TGG_{Basic}) = L(TGG_{NAC})$.

□

Finally, it is proven that adding PACs to basic rules yields a higher expressiveness than adding NACs, i. e., that $\mathcal{TGG}_{\text{PAC}}$ is more expressive than $\mathcal{TGG}_{\text{NAC}}$.

Theorem 4.5 (Expressiveness of PACs)**.**

$$\mathcal{TGG}_{NAC} <_E \mathcal{TGG}_{PAC}$$

Proof.

- $\mathcal{TGG}_{\text{NAC}} \leq_E \mathcal{TGG}_{\text{PAC}}$: This proposition follows directly from the fact that every *NAC* n for a rule r is an application condition for r of the form $(n : L \to N, \{\})$ according to Def. 4.10.
- $\mathcal{TGG}_{\text{NAC}} \neq_E \mathcal{TGG}_{\text{PAC}}$: For this direction of the proof, the reader is referred to Wagner [Wag95]. The main argument used in this proof is that the number of elements that are added within one rule application is finite, while the number of possible matches for the premise that have to be considered can be arbitrarily large, which makes it impossible to cover all possible constellations with NACs only. A concrete example is very complicated and out of the scope at this point.

□

4.6 Multi-Amalgamation

Multi-amalgamation (first presented by Boehm et al. [BFH85] for graph transformations and Leblebici et al. [LAST17] for TGGs) leverages TGGs by partitioning a rule into a "kernel rule" to be applied once and a set of "multi-rules" that extend the kernel and can be applied multiple times.

To illustrate the concept of multi-amalgamation, we consider the last language constructs which are not yet covered by our set of TGG rules. In the example SysML instance (Fig. 3.1), there is a third state `s0` which represents the initial state of the state machine. As it is left immediately via a transition to the start state `s1`, its concrete type is `Pseudostate`. In the SysML model, the transition does not have any guards or actions, whereas the corresponding Event-B event is used to initialise all variables to their default values (cf. Fig. 3.6). The desired graph is depicted in Fig. 4.12. To improve readability, only nodes and edges that are directly relevant for this example are depicted (i. e., all variables, invariants and the second transition and its corresponding event are omitted).

We now want to create a rule to express that when adding a pseudostate `p` and an initialisation transition `t` to an existing region `r`, for all states `sm1` of `r` and all ports `pm2`, actions `am1` and `am2` are added to the Event-B model that initialise the corresponding variables with their default values. The variable corresponding to the target state `sk` of the initialisation transition `t` is set to `TRUE`, all other variables are set to `FALSE`. All actions are added to the initialisation event `e` that corresponds to `t`. Creating such a rule is not possible using only basic rules, as for adding `p` and `t` to the SysML model, n actions have to be added in the Event-B model, depending on the number of existing ports and states. For a concrete example with a fixed number of ports and states, a rule could be created that adds the respective actions when the pseudostate `p` is created. But as there can be arbitrary many ports and states, infinitely many rules would be required, contradicting the definition of a TGG.

To overcome this problem, the foreach-like notion of multi-amalgamated rules can be used. A multi-amalgamated rule is an interaction scheme consisting of one kernel-rule and arbitrary many multi-rules [GEH10]. The multi-rules have to contain the kernel rule, but can extend it by further nodes and edges that can be both context elements or created elements. To apply a multi-amalgamated rule, one has to find a match for the kernel rule. Then the multi-rules are applied everywhere the multi-rule can be matched. Rule application is restricted to maximal and unique matches, meaning all multi-rule matches that are available over the same kernel are joined and a multi-rule application over the same match is never joined twice [LAST17].

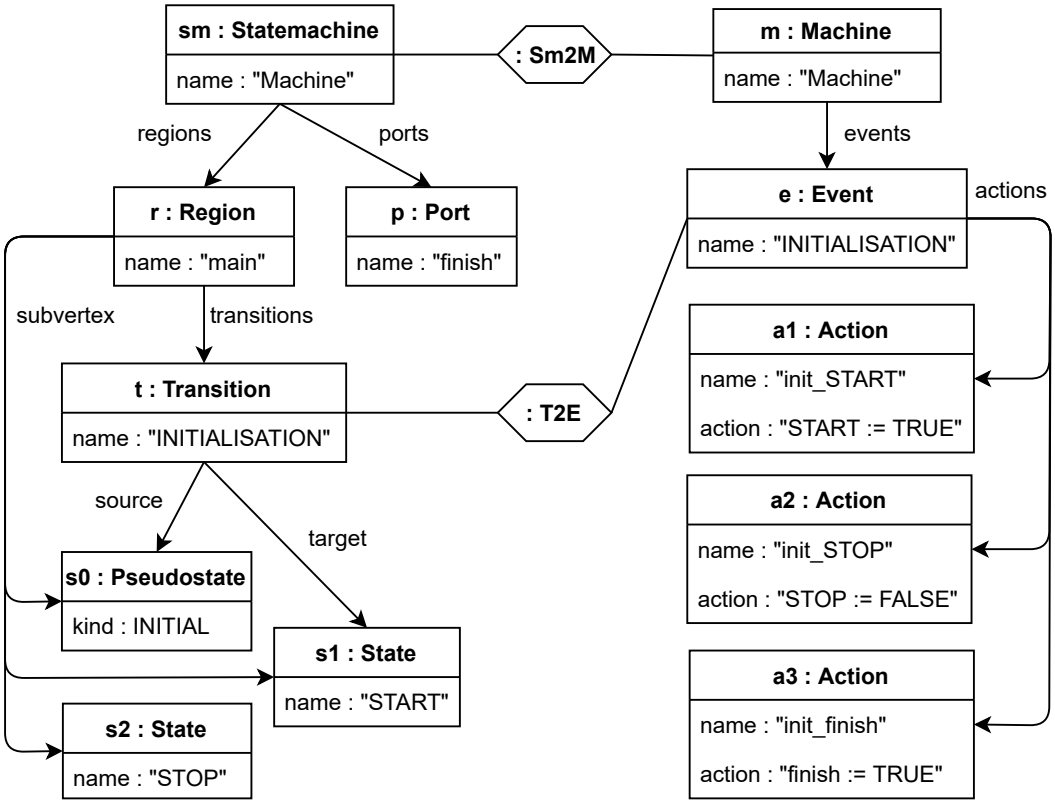


Figure 4.12: Desired target graph after adding the initial state

The rule in Fig. 4.13 can be matched on the example instance of Fig. 4.12 as shown in Tab. 4.1. The matched nodes are referred to by their name attribute values to avoid naming conflicts. The kernel rule K creates a pseudostate and an initialisation transition within the main region. Furthermore, the multi-rule M_1 is joined over this match, meaning that for each existing state of the same region, an action with the respective value assignment is created in the Event-B model. Similarly, the multi-rule M_2 creates a respective action for all ports of the statemachine. Due to the demand of maximality, this ensures that all variables are initialised. After applying the multi-amalgamated rule, the graph looks as shown in Fig. 4.12.

Formal Definition

The definitions 4.14 – 4.17 are taken from seminal work by Leblebici et al. [LAST17] in an adapted version and are subsequently used to assess the expressive power of multi-amalgamation according to Def. 4.1. We distinguish between kernel rules and multi-rules, which are grouped into interactions schemes.

Definition 4.14 (Kernel Rule, Multi-Rule, Interaction Scheme)**.**

Given triple rules r_o and r_1, a kernel morphism $k_1 : r_0 \to r_1$ consists of two triple graph morphisms $k_{1,L} : L_0 \to L_1$ and $k_{1,R} : R_0 \to R_1$ such that the square of Fig.4.14 [LAST17] is a pullback, i. e., r_1 must demand and create at least the same elements as r_0 but can demand and create additional elements. In this case, r_0 is called the ***kernel rule*** *and r_1 the* ***multi-rule****. A kernel rule r_0 and a set of multi-rules $\{r_1, \ldots, r_n\}$*

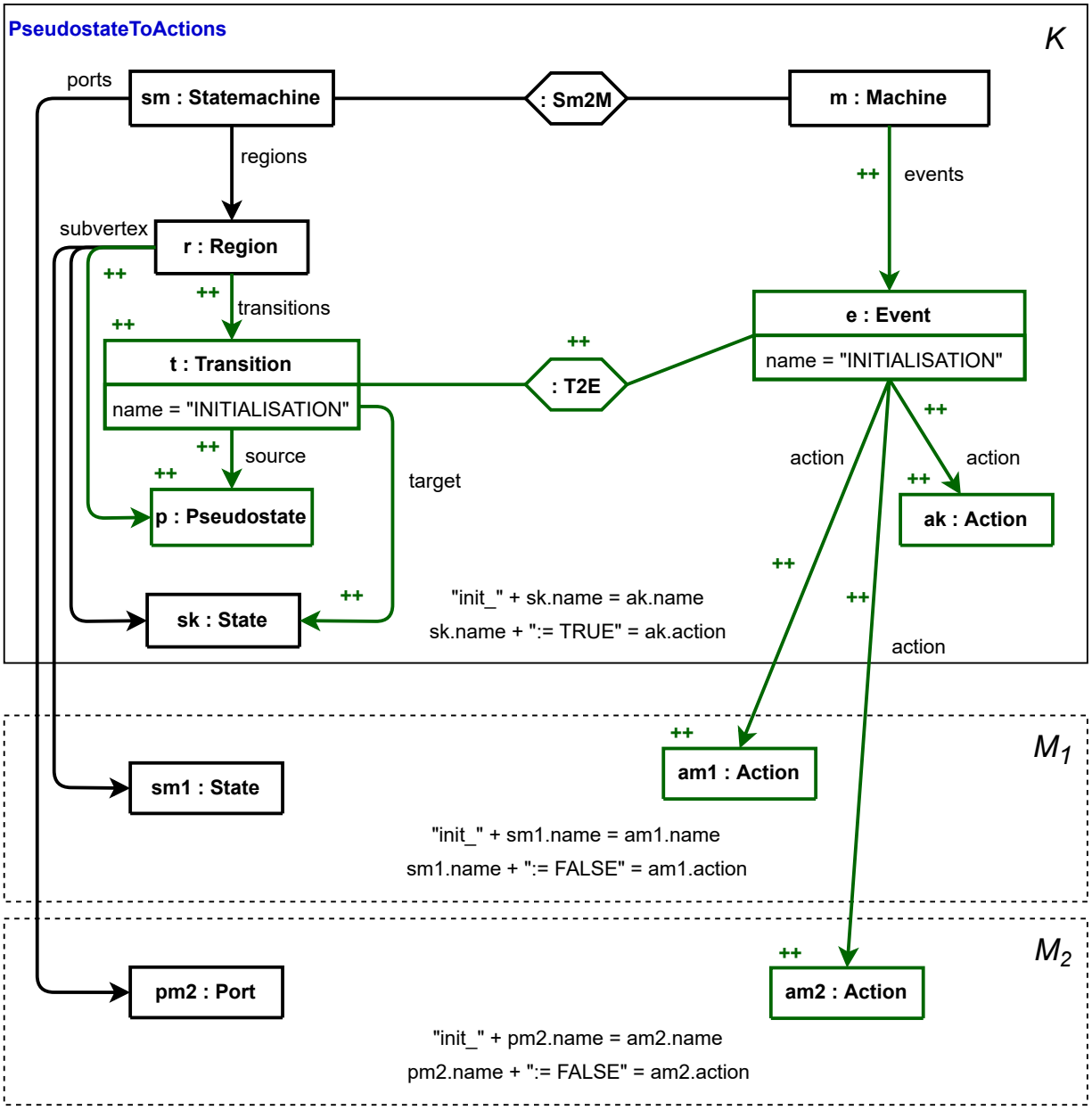


Figure 4.13: Rule: PseudostateToActions using multi-amalgamation

Rule	Match (SysML)	Match (Event-B)
K	sm ↦ Machine r ↦ main t ↦ INITIALISATION p ↦ finish sk ↦ START	m ↦ Machine e ↦ INITIALISATION ak ↦ init_START
M_1	sm1 ↦ STOP	am1 ↦ init_STOP
M_2	pm2 ↦ finish	am2 ↦ init_finish

Table 4.1: Match of PseudostateToActions (Fig. 4.13) in the example instance (Fig. 4.12)

with respective kernel morphisms $\{k_1, ..., k_n\}$ *form an* ***interaction scheme****. For* $n = 0$*, the interaction scheme consists of the kernel rule only.*

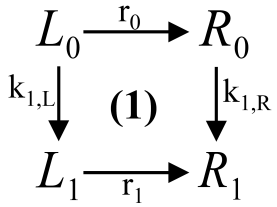


Figure 4.14: Construction of a multi-amalgamated rule

Definition 4.15 (Maximally Amalgamable)**.**
Given an interaction scheme s *with kernel rule* r_0 *and triple graph* G*, let* $d_0 : G \overset{r_0@m_0}{\Longrightarrow} G_0$ *be a direct derivation via* r_0 *and* $D : \{G \overset{r_i@m_i}{\Longrightarrow} G_i\}_{i=1,...,t}$ *a bundle of direct derivations, where* $\{r_1, ..., r_t\}$ *are multi-rules of* s *with kernel morphisms* $\{k_1, ..., k_t\}$*. D is* ***amalgamable*** *for* d_0 *if* $\forall p, q \in \{1, ..., t\}$ *all multi-rule matches are unique (i. e.*$p \neq q \Rightarrow m_p \neq m_q$*) and agree on the kernel match* m_0 *(i. e.*$m_p \circ k_p = m_q \circ k_q = m_0$*). D is* ***maximally amalgamable*** *for* d_0 *if* $\nexists d_z : G \overset{r_z@m_z}{\Longrightarrow} G_z$ *such that* $(D \cup \{d_z\})$ *is amalgamable for* d_0*.*

Definition 4.16 (Multi-Amalgamated Rule)**.**
Given an interaction scheme s *with kernel rule* r_0*, a direct derivation* $d_0 : G \overset{r_0@m_0}{\Longrightarrow} G_0$ *and a bundle* $D : \{G \overset{r_i@m_i}{\Longrightarrow} G_i\}_{i=1,...,t}$ *of direct derivations that is maximally amalgamable for* d_0*. Let* $\tilde{K} : \{k_i = r_0 \to r_i\}_{i=1,...,t}$ *be the bundle of respective kernel morphisms for D. The* ***multi-amalgamated rule*** $\tilde{r} : \tilde{L} \to \tilde{R}$ *is constructed by gluing multi-rules over the kernel rule via iterated pushouts with the kernel morphisms in* $\tilde{K}$ *as depicted in Fig. 4.15 [LAST17], where the grey region marks the results after each iteration.*

The construction starts with $\tilde{r_0} = r_0$*, i. e.the kernel rule, and ends with* $\tilde{r} = \tilde{r_t}$*. After each iteration* $i \in \{1, ..., t\}$*, the pushouts* $(i)_L$ *and* $(i)_R$ *construct* $\tilde{L_i}$ *and* $\tilde{R_i}$ *respectively. The rule morphism* $\tilde{r_i} : \tilde{L_i} \to \tilde{R_i}$ *is induced via the universal property of the pushout* $(i)_L$*, i. e.*$\tilde{r_i} \circ u_{i,L} = u_{i,R} \circ \tilde{r_{i-1}}$ *and* $\tilde{r_i} \circ e_{i,L} = e_{i,R} \circ r_{i-1}$*. We call* $G \overset{\tilde{r}@\tilde{m}}{\Longrightarrow} G'$ *a* ***multi-amalgamated direct derivation*** *where the* ***multi-amalgamated match*** $\tilde{m} : \tilde{L} \to G$ *is determined uniquely by the kernel rule match* m_0 *as well as the multi-rule matches in D, i. e.*$\tilde{m} \circ e_{t,L} = m_t$ *and* $\tilde{m} \circ (u_{t,L} \circ ... \circ u_{q+1,L}) \circ e_{q,L} = m_q$*,* $\forall q \in \{0, ..., t-1\}$*.*

In the following, the definition of TGGs is extended by interaction schemes for multi-amalgamated rules.

Definition 4.17 (Multi-Amalgamated Triple Graph Grammar)**.**
A triple graph grammar $TGG = (TG, S)$ *consists of a type triple graph* TG *and a finite set* S *of interaction schemes. The generated language* $L(TGG) \subseteq L(TG)$ *is defined as follows:* $L(TGG) := \{G_\emptyset\} \cup \{G | \exists \tilde{d} : G_\emptyset \overset{\tilde{r_1}@\tilde{m_1}}{\Longrightarrow} G_1 \overset{\tilde{r_2}@\tilde{m_2}}{\Longrightarrow} ... \overset{\tilde{r_n}@\tilde{m_n}}{\Longrightarrow} G\}$*, where* $G_\emptyset$ *is the empty triple graph, each* $\tilde{r_i}$ *with* $i \in \{1, ..., n\}$ *is a multi-amalgamated rule derived from an interaction scheme* $s_i \in S$*. If* S *is empty, we get* $L(TGG) := \{G_\emptyset\}$*. We refer to* TGG *as a* ***plain TGG*** *if each interaction scheme in* S *contains only a kernel rule and no multi-rule. Otherwise, we refer to* TGG *as a* ***multi-amalgamated TGG****.*

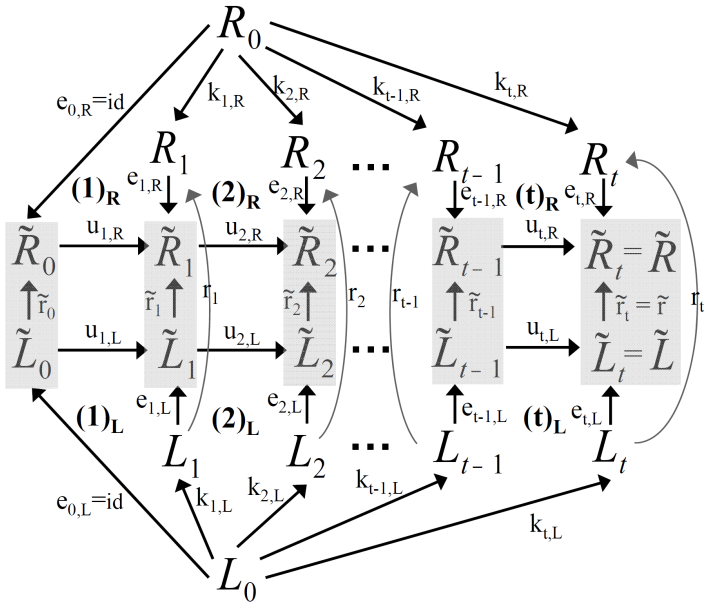


Figure 4.15: Construction of a multi-amalgamated rule (details)

Definition 4.18 (TGG Variant: Multi-Amalgamation)**.**
Let $\mathcal{TGG}_{MA}$ be the TGG variant supporting basic rules and multi-amalgamated rules.

Expressiveness

The following explanations are an adapted version of prior work by Leblebici et al. [LAST17] and embedded into our classification framework.

Theorem 4.6. *$\forall\ TGG_{MA} \in \mathcal{TGG}_{MA}\ \exists\ TGG_{Basic} \in \mathcal{TGG}_{Basic}$ for which $\mathcal{L}(TGG_{MA}) \subseteq \mathcal{L}(TGG_{Basic})$*

Proof. Direct derivations with multi-amalgamated rules can be obtained by applying multi-rules in combinations after a kernel rule. As multi-rules are triple rules, one can create a $TGG_{Basic} \in \mathcal{TGG}_{\text{Basic}}$ by taking all kernel rules and multi-rules from $TGG_{MA} \in \mathcal{TGG}_{\mathcal{MA}}$. Therefore, all graphs that can be created by TGG_{MA} can be created by TGG_{Basic} as well. Note that this strategy is only applicable in settings *without* application conditions (cf. Sect. 4.5), because otherwise, the application of a kernel rule could restrict the applicability of the respective multi-rules. □

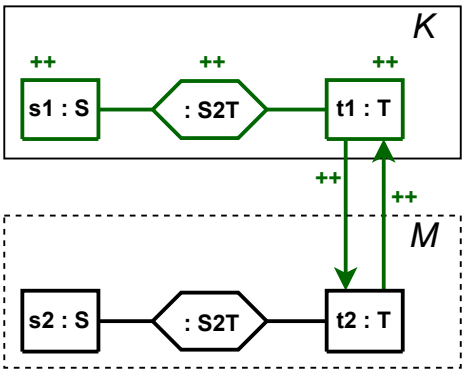
Hence multi-amalgamated TGGs do not enlarge the language of basic TGGs but restrict it for more precise consistency specifications. Now we reason why this preciseness cannot be achieved by basic rules by taking a look at the model size relations of the created source and target models. The distinction between basic TGGs that contain rules only adding elements to either source or target graph ($\mathcal{TGG}^{\mathcal{S}}$, $\mathcal{TGG}^{\mathcal{T}}$) and TGGs that do not contain such rules ($\mathcal{TGG}^{=}$) is already done in Sect. 4.3.

In $\mathcal{TGG}_{\text{MA}}$, there exist TGGs for which the maximal proportion is neither restricted to a constant nor infinite, but is a function in size of the source graph. This characteristic exists if the multi-amalgamated TGG's kernel rule is not in $\mathcal{TGG}^{\mathcal{S}}$ or $\mathcal{TGG}^{\mathcal{T}}$ but some multi-rules are. In this case, finitely many applications of the kernel rule are needed to create a source graph G_S, because elements are created in both graphs. The number of possible multi-rule applications is also finite, because the application is controlled by

the semantics of multi-amalgamation. TGGs supporting multi-amalgamation are more expressive than basic TGGs:

Theorem 4.7 (Expressiveness of Multi-Amalgamation)**.**
$\exists\, TGG_{MA} \in \mathcal{TGG}_{MA}$ *such that* $\nexists\, TGG_{Basic} \in \mathcal{TGG}_{Basic}$ *with* $\mathcal{L}(TGG_{MA}) = \mathcal{L}(TGG_{Basic})$.

Proof.
We want to show that the maximal proportion of source and target graphs are not always restricted to a constant or infinite in contrast to basic TGGs. Let TGG_{MA} consist only of the interaction scheme depicted to the right. For this example, every target graph G_T of every triple graph $G \in \mathcal{L}(TGG_{\text{MA}})$ consist of $|G_S|$ vertices and $|G_S|^2 - |G_S|$ edges, leading to a maximal proportion of



$$\frac{|G_T|}{|G_S|} = \frac{|G_S| + |G_S|^2 - |G_S|}{|G_S|} = |G_S|$$

Therefore, the maximal proportion is a linear function in size of the source graph, a characteristic that cannot be found for basic TGGs. Hence multi-amalgamated TGGs are more expressive than basic TGGs. □

4.7 Completed Running Example

In Sect. 4.3, 4.4, 4.5 and 4.6, all relevant TGG language features for this thesis could be presented with example rules for a consistency management scenario involving the languages SysML and Event-B. These rules, however, are not yet sufficient, as events, guards and triggers of transitions have not been considered yet. Therefore, some further rules are added to the example TGG in the following, which get along with the previously introduced language features. The rule *EffectToAction* (cf. Fig. 4.16) adds an effect `ef` (e. g., a variable assignment) to a transition `t` and links it to an action `a` in the Event-B code. Attribute conditions ensure that the names and the bodies of `ef` and `a` are equal.

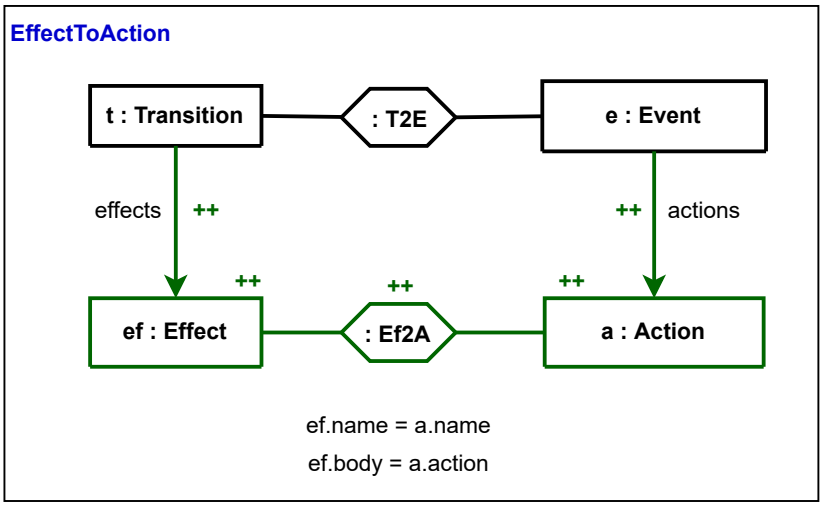


Figure 4.16: Rule: EffectToAction

Let us have a short recap of the introduced rules that form the example TGG at this point. Considering the SysML state machine of Fig. 3.1, we are able to transform the example instance to Event-B, except for the trigger and the guard which pose further restrictions on when the transition can be executed. Figure 4.17 shows the parts of the state machine which can be transformed already, together with the produced Event-B model and the respective correspondence nodes. In order to check that the triple is consistent, a derivation sequence, i. e., a sequence of rule applications, must be found that generates the triple graph. In total, ten rule applications are necessary: The rule *StateToVariable* (Fig. 4.3) is applied twice, whereas all other rules (*StatemachineToMachine* (Fig. 3.11), *PortToVariable* (Fig. 3.10), *AddRegion* (Fig. 4.2), *TransitionToEvent* (Fig. 4.7), *SourceStateToLeaveAction* (Fig. 4.8), *TargetStateToEnterAction* (Fig. 4.9), *EffectToAction* (Fig. 4.16), *PseudostateToActions* (Fig. 4.13)) are applied once. As the source model has two regular states (`s1`, `s2`) and one port `p`, the kernel rule of *PseudostateToActions* is applied once, and the multi rule is applied twice.

The last two rules which are required to fully transform the SysML example state machine transform SysML triggers `tg` (Fig. 4.18) and guards `g1` (Fig. 4.19) into Event-B guards (`g`, `g2`), respectively. These rules are structurally similar to the *EffectToAction* rule. They both involve two attribute conditions each, which check for equality of names and bodies. An interesting point to consider is the non-determinism induced by the two rules for the backward direction: Translating an Event-B guard into either a trigger or a guard is equally possible with the example TGG, but one can assume that the choice is not completely arbitrary as the two SysML language constructs have different semantics. This leads us back to the introductory example for multiple transformation solutions as shown in Fig. 1.4, which was used to motivate fault-tolerant system behaviour.

The goal of this chapter was to present a complete overview of TGG language features and compare them with respect to their expressive power. The fault-tolerant approach is able to support these features to a large extent, but some restrictions have to be made to further proceed with the example TGG:

- Source- and target-idle rules as well as ignore rules will only be supported for a subset of all possible consistency management operations. The concrete conditions will be explained in detail in Chap. 5, 6 and 7, such that the source-idle rule *AddRegion* (cf. Fig. 4.2) will remain in the rule set of the example TGG.

- Application conditions are a means of restricting the applicability of TGG rules, and thereby indirectly guaranteeing that the produced models fulfil the intended constraints. Theorem 4.3 shows that it is always possible to generate an application condition from a constraint. The fault-tolerant approach will provide direct support for constraints, but can, in turn, not handle application conditions. Therefore, we will remove all application conditions from the example rules *StatemachineToMachine* (Fig. 4.10), *TransitionToEvent* (Fig. 4.7), *SourceStateToLeaveAction* (Fig. 4.8) and *TargetStateToEnterAction* (Fig. 4.9), and add respective graph constraints to the TGG that serve the same purpose in Chap. 6.

- The fault-tolerant approach – in its current form – cannot handle multi-amalgamation. The only situation in which this language feature is required is, however, when default values are assigned to all variables in the Event-B model, i. e., when the initialisation transition fires. In the following, we will assume that all variables are initialised with `FALSE` per default, such that the multi-rules can be removed from *PseudostateToActions* (Fig. 4.13).

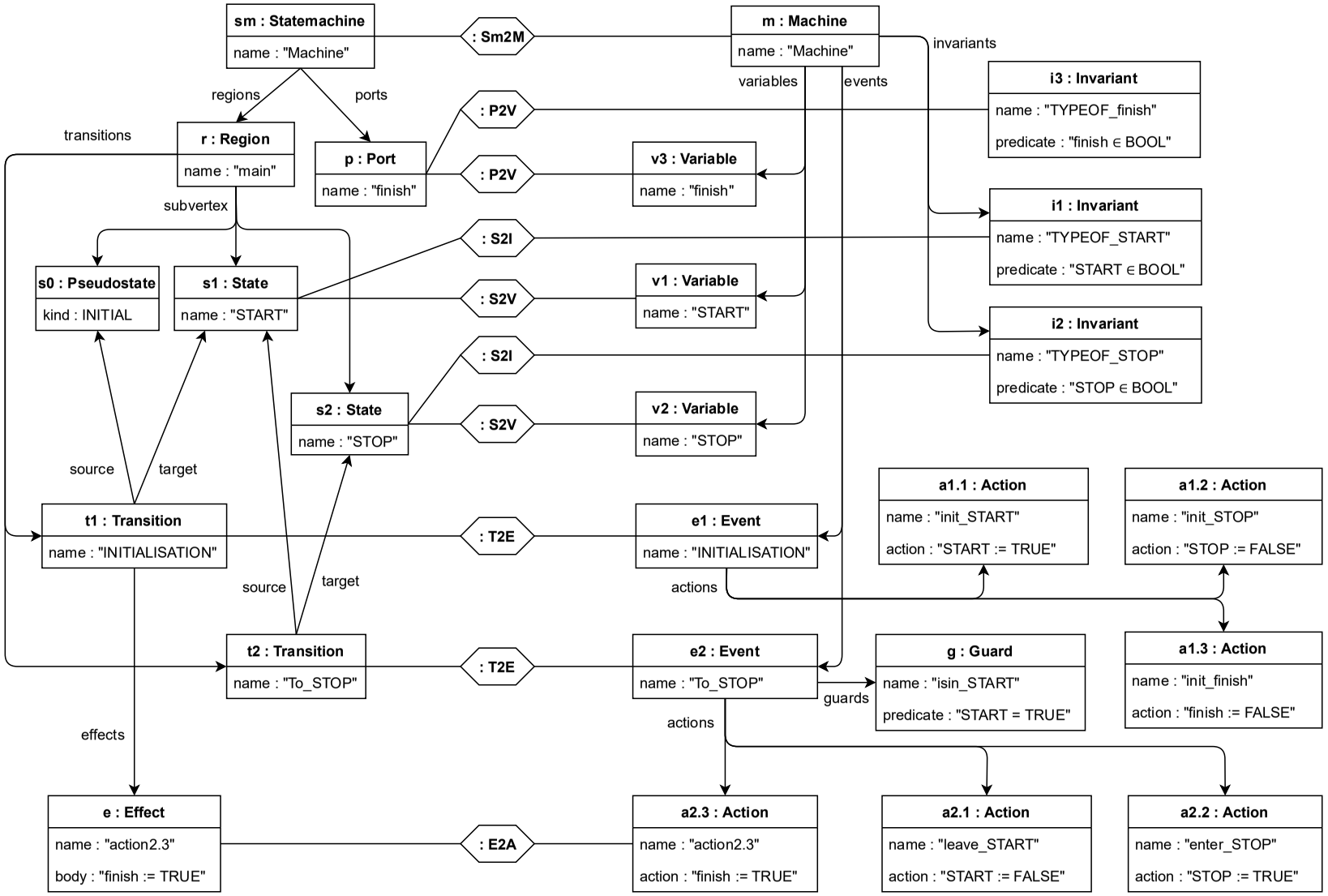


Figure 4.17: Sample instance without guards and triggers

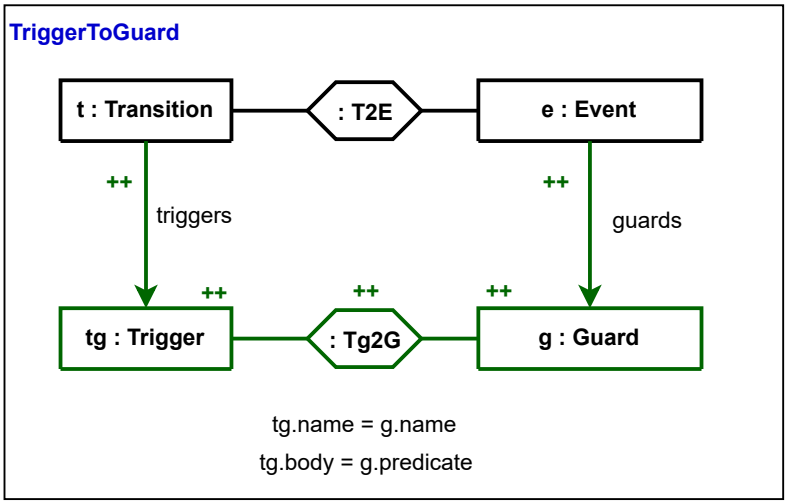


Figure 4.18: Rule: TriggerToGuard

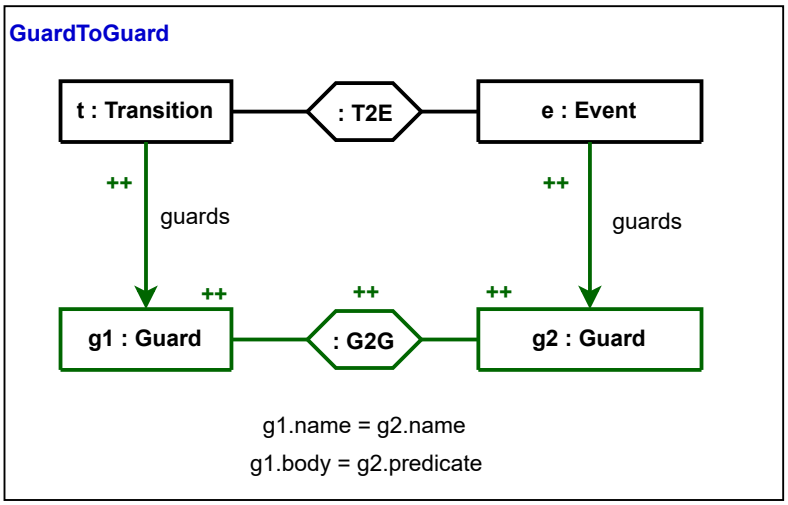


Figure 4.19: Rule: GuardToGuard

4.8 Summary and Discussion

In this chapter, different language features of TGGs have been introduced, providing a basis for the conceptual work on fault-tolerant consistency management in the remainder of this thesis. We identified and described language features of TGGs, summed up in a feature model that can be used to classify TGG variants (cf. Fig. 4.1).

In addition, we evaluated the expressiveness of TGG variants depending on the supported language features and provided examples illustrating where particular language features can be useful. We have proven that having rules which affect only one model increases the expressiveness (Thm. 4.1). While attribute conditions that only check the equality of values can be simulated with data vertices, we have shown that for TGGs with acyclic type graphs, arithmetic expression cannot be specified using only basic rules (Thm. 4.2). With a minimum example, we have shown that application conditions increase the expressiveness compared to using only basic rules (Thm. 4.4). For the difference between NACs and PACs (Thm. 4.5) and multi-amalgamation (Thm. 4.7), we referred to existing proofs that demonstrate the increased expressive power of such language features.

All results regarding the expressiveness of different TGG variants are summarized in Fig. 4.20: Each node represents one of the previously presented TGG variants. An arrow between two nodes A and B illustrates that the TGG variant A is more expressive than B. As an outlook on Chap. 5 and 6, TGG variants supported by the fault-tolerant framework are coloured blue, whereas unsupported TGG variants are coloured grey. With the exception of multi-amalgamation, all presented TGG variants are supported, as application conditions can be constructed from graph constraints according to Ehrig et al. [EEPT06, p. 157-159].

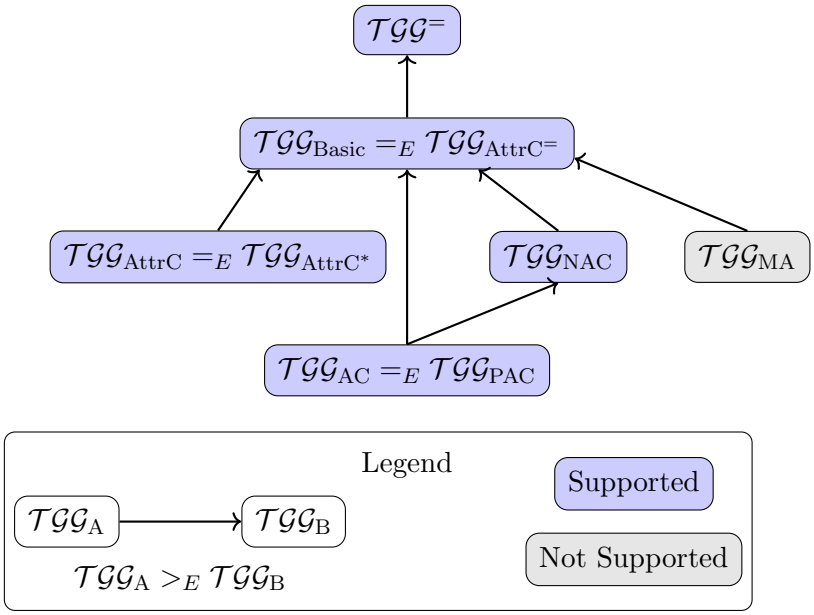


Figure 4.20: Summary: Expressiveness of TGG variants

The question whether TGG variants which combine multiple language features increase the expressive power even more is still left open, though. While we are convinced that attribute conditions are a powerful feature for TGGs in practical use, the question whether they generally increase the expressiveness is undecided. Furthermore, it needs to be investigated if general application conditions are more expressive than application conditions generated from constraints, or if these two TGG sets are equally expressive. It would be also worthwhile to investigate whether the expressive power of nested conditions as specified by Habel and Pennemann [HP09] can be leveraged for TGGs. Formal properties for graph transformations with nested conditions, such as correctness and completeness, have been proven by Poskitt and Plump in prior work [PP14].

Our feature model could be extended with further language features, such as rule refinement or rules that can delete elements. To be able to find an appropriate tool for one's needs, the support of the different language features in current TGG tools must be analysed, e. g., by a classification according to the feature model given in Fig. 4.1. An overview of tool support would assist people specifying the consistency between two model spaces using TGGs with finding a tool supporting all features needed by them.

After the first four chapters have laid the foundations for fault-tolerant consistency management in MDE, the conceptual solution of this thesis can be presented in the following Chap. 5 - 8.

Part II

Conceptual Solution

5 Fault-Tolerant Model Transformation and Consistency Checking

To address the requirement of performing model transformations in the presence of faults, a hybrid framework is proposed that synergetically combines TGGs as a rule-based BX approach, and ILP, an exact algorithm for solving combinatorial optimisation problems. With these two techniques, tolerance towards faults in the input models is achieved by computing a triple that maximises the number of translated elements, denoted as the "largest consistent sub-triple" in the following, in case of inconsistent input models. The framework is able to perform two sorts of consistency checks, as well as forward and backward transformations using a common problem definition and solution strategy. An experimental performance evaluation indicates that a reasonable trade-off between flexibility and scalability could be made.

This chapter is structured as follows: The idea of combining algorithmic and search-based approaches to consistency management is motivated in Sect. 5.1. Section 5.2 discusses related work, before Sect. 5.3 presents the overall work-flow of constructing the ILP based on the input models for different consistency management tasks. Sections 5.4, 5.5, 5.6, and 5.7 present each main step in more detail. The results of a runtime performance evaluation comparing the four operations for different model sizes is presented in Sect. 5.8. Finally, the results are summarised in Sect. 5.9.

5.1 Motivation

While a substantial amount of conceptual work on fault-tolerance was identified in Chap. 2, it became apparent that there is indeed a lack of research on model transformations in the presence of faults. Based on the review of existing work, one can state that there is no framework for efficiently solving multiple consistency management tasks in a fault-tolerant manner. Existing (fault-intolerant) approaches to consistency management can be divided into *algorithmic* and *search-based* approaches:

Algorithmic approaches typically enable full control over consistency management and exploit knowledge about the involved data structures and consistency management operations. While this allows for scalability with respect to model size, the price is a relatively fixed and inflexible strategy, as well as numerous assumptions regarding supported changes and input models. Furthermore, algorithmic approaches are highly optimised for one specific consistency management operation, such that the strategies for different operations are substantially heterogeneous. Both of these factors mean that it is hard to adapt algorithmic approaches to fault-tolerant scenarios. Examples of algorithmic approaches (formal foundations and tool support) include work on lenses [KZH16, KH06, BFP+08], and on TGGs [ALS15, GH09, EHGB12].

Search-based approaches, in contrast, view consistency management as a search problem and often leverage generic, flexible constraint solvers to determine an *optimal* solution [MC13, CRE+10]. Instead of an exact solver, some search-based approaches use evolutionary strategies [FTW16, KSB08, DJVV14]. Almost no assumptions on input models and changes are made, such that search-based approaches seem to be a promising solution

strategy for fault-tolerant consistency management. The primary drawback, however, is that they cannot cope with large models due to rapid search space explosion [MC13]. Purely constraint-based approaches often suffer from severe scalability problems as they operate on the level of model elements and, therefore, involve numerous constraints that guarantee basic graph properties.

For a fault-tolerant approach to consistency management, we propose to synergetically combine algorithmic and search-based approaches to yield a hybrid solution that is both reasonably scalable and at the same time more flexible than existing algorithmic approaches. Our proof-of-concept is a concrete combination of TGGs as an algorithmic approach and ILP as an exact search-based approach. The idea originates from Leblebici et al. [Leb16, LAS17, Leb18], who used this hybrid strategy to specify a TGG-based consistency checker - as they argue that this task is hardly solvable in a purely algorithmic manner. A search space of *potential* rule application candidates is constructed, whereby matches are determined via graph pattern matching and associated with variables of an ILP. In this way, it is possible to keep the search space relatively small (compared to purely search-based approaches), while still benefiting from a high degree of flexibility.

Our contribution is to substantially extend and generalise the work of Leblebici et al. towards a powerful framework for fault-tolerant consistency management. While the original approach focused solely on consistency checking for a given pair of models (i. e., without correspondence links), our extension uniformly covers consistency checking for a given *triple* of models (i. e., with correspondence links), as well as (unidirectional) forward and backward transformation. Further extensions towards graph constraint handling and the synchronisation of concurrent changes on the two models follow in Chap. 6, 7, and 8. A key advantage of having a generic problem definition is that all operations share a common work-flow. As a result, the implementation of these operations can make use of a common code base, which is easier to maintain than separate implementations.

The work of Leblebici et al. focussed on proving formal properties for consistent inputs, i. e., that the approach is *correct* (if a solution is returned, it is consistent) and *complete* (if a consistent solution exists, it will be returned). From the perspective of fault-tolerance, however, the approach also yields a useful "by-product" for inconsistent inputs: In case no strictly consistent solution exists, the *largest* consistent sub-triple is returned, i. e., the part of the given models maximising the number of elements that can be consistently translated, determined via an optimisation process for a configurable objective function. In this way, the consistency management engine returns either a consistent solution (if the input models were consistent) or a solution which is *as consistent as possible* by forming a union of the largest consistent sub-triple and the remaining input elements.

5.2 Related Work

There exists a considerable amount of work on encoding the structure of models, metamodels, and transformation definitions into constraints that can be directly passed on to a constraint solver.

The Python-based tool *CoReS* [KN18] encodes graph structures into both Boolean Satisfiability Problem (SAT) and SMT formulas and hands them over to a corresponding solver. Similar to ILP, SMT models a constraint satisfaction problem by inequalities, which are indeed not restricted to integers or real numbers. In the area of consistency management, the Janus Transformation Language (JTL) framework [CRE+10] supports unidirectional transformation and model synchronisation derived from a common specification by deriving constraints that are handed over to a DLV solver in a second step.

Similarly, Macedo et al. [MC13] developed *Echo*, a transformation engine that supports metamodels enriched with OCL constraints and that is able to perform different consistency management operations derived from a common constraint-based specification. In the case of Echo, Alloy is used as an underlying solver. FunnyQT [Hor17] (used in our evaluation) is another constraint-based approach that uses the *core.logic*[1] of Clojure for constraint solving. While all these approaches can (at least potentially) easily return a model which is "closest" to a consistent solution in case of inconsistent inputs, scalability remains a crucial challenge as the underlying solvers cannot fully exploit the graph structure of (meta)models.

If the hard guarantee of providing an exact solution (if it exists) can be sacrificed, then such approaches as proposed by Fleck et al. [FTW16], can be used to combine OCL constraints and graph pattern matching with techniques of local and global search. The corresponding tool MOMoT is also able to exploit evolutionary algorithms both with single and multiple objectives. Denil et al. [DJVV14] use the multi-objective optimisation technique Design Space Exploration (DSE) in combination with the T-core transformation framework [SVL15]. In their tool *MOTOE* [KSB08], Kessentini et al. extract transformation blocks from examples and use particle swarm optimisation as a search technique. Basic evaluations on transforming UML diagrams into relational database schemas - similar to our running example - have shown that only about half of the transformation runs find a correct solution, though.

In general, while this group of search-based approaches can potentially scale much better than exact search-based approaches, they do this at the price of giving up hard guarantees for finding either a perfectly consistent solution if it exists, or finding a globally optimal solution in case of inconsistent inputs. Especially for consistency checks, exactness is crucial to gain the users' trust in the software system, such that we decided not to use heuristics to build up the consistency management framework.[2]

At the other end of the spectrum (efficient, exact, but rather inflexible), numerous algorithmic approaches exist often based on either lenses [BFP+08], or TGGs [ALS15]. BiGUL (used in our evaluation) is a putback-based bidirectional transformation language, which means that having implemented the backward "put" transformation, the forward "get" direction is derived for free [KZH16] with well-behavedness guarantees from the lenses framework. Other comparable lens-based approaches include the biXid programming language [KH06] for XML data, and Boomerang [BFP+08, FPP08] for string data.

MOTE [GH09,LHGO12] supports incremental model synchronisation based on the TGG formalism. HenshinTGG [EHGB12] is an extension of the Graph Transformation (GT) tool Henshin[3] and also implements support for TGGs. eMoflon::TiE [LAS14a] - the predecessor of eMoflon::IBeX and eMoflon::Neo - is a TGG tool capable of transforming and synchronising models, as well as checking for consistency. While all these algorithmic approaches and tools are both exact and (highly) efficient, they pose numerous restrictions: BiGUL's derived functions are partial and simply reject invalid input (producing nothing), MOTE, eMoflon::TiE, and Henshin-TGG all assume certain conditions regarding the structure of the underlying TGG and input models (confluence, local completeness, conflict-freeness, etc).

There have also been proposals for hybrid solutions similar to our approach: Callow and Kalawski [CK13] introduce a combination of model transformation by pattern matching and mixed integer linear programming with a high priority of target model compliance in

[1] https://github.com/clojure/core.logic

[2] The concurrent synchronisation operation can be seen as an exception, where an alternative solution strategy based on different heuristics was developed (cp. Chap. 8).

[3] https://www.eclipse.org/henshin/

contrast to maximizing the number of matches. This approach, however, only considers forward and backward transformations and does not support consistency checking.

Our ILP-based operations build upon and extend the seminal work by Leblebici et al. [LAS17, Leb18], which focused on consistency checking without correspondence links. This chapter shows that the basic idea can be extended and generalised – both formally and implementation-wise – to a wider range of consistency management operations.

5.3 Solution Overview

We begin with a high-level overview of the work-flow of fault-tolerant consistency management, which is equal for all operations. The proposed process for consistency management is depicted in Fig. 5.1 in form of a UML activity diagram with object flows. It can be divided up into five main steps: Based on the declarative TGG specification, *operational rules* are generated for the different consistency management tasks, such as consistency checks or forward and backward transformation (A). For these operational rules, *rule application candidates* are collected via graph pattern matching on the input models (B). The pattern matching step itself is an iterative process: Multiple iterations are necessary to find all rule application candidates, as operational rules create new elements that can provide context for dependent rule applications.

Based on these candidates and their interdependencies, an *ILP* is generated (C). It consists of an objective function that maximises the number of "translated" elements, and of constraints that guarantee language membership. The *optimisation* step determines the optimum solution for the ILP (D). Finally, the rule applications are *filtered* according to the determined solution of the ILP to form a consistent output triple in case of consistent inputs, and the largest consistent sub-triple otherwise (E). In the latter case, the framework informs the user which elements remain untranslated, such that faults can be removed to eventually restore consistency.

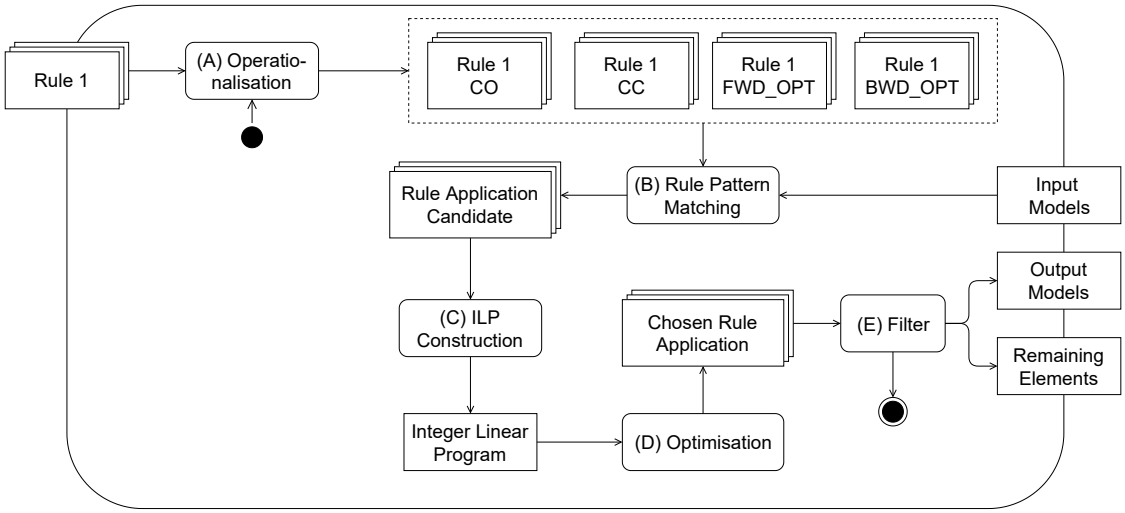


Figure 5.1: Process for fault-tolerant consistency management

In the following subsections, each step is described in detail, including formal definitions and an illustrative forward transformation in the context of our running example.

5.4 Operationalisation

TGGs according to Def. 3.7 *generate* triples by creating elements in each domain (source, correspondence, and target). In this way, a language of (consistent) triples is defined. A TGG as specified by an end user can be automatically *operationalised* to support different consistency management tasks. In this chapter, we focus on the following four operations – two of them *check for consistency*, whereas the other two perform a unidirectional model transformation in both directions:

- The operation **CO** (Check Only) is a read-only operation that accepts a complete triple as input and checks if it is consistent or not.
- Provided with a source and a target model, the operation **CC** (Correspondence Creation) attempts to create a correspondence model that completes the given pair of models to a consistent triple.
- The **FWD_OPT** (OPT for optimisation) operation takes a source model as input and attempts to perform a forward transformation, i. e., complete the input to a consistent triple.
- Analogously, **BWD_OPT** takes a target model as input and attempts to complete it to a consistent triple via a backward transformation.

To determine if a concrete triple graph G is a member of the language of a TGG, one searches for a derivation sequence starting with the initial triple graph (cf. Def. 3.7) and producing G. The language is composed of all graphs that can be *generated* by applying triple rules, creating elements in each domain (source, correspondence, and target) simultaneously. However, this definition is not sufficient to specify, e. g., forward and backward transformations, because parts of the triple are already given. These parts differ for all operations and can be combinations of the source, target and correspondence graphs, which we denote as the *starting triple graph*.

Definition 5.1 (Starting Triple Graph)**.**
For each operation $op \in \{CC, CO, FWD\_OPT, BWD\_OPT\}$*, a starting triple graph* G_0 *for a triple graph* G *is defined as:*

Operation	***Starting Triple Graph*** (G_0)
CC	$G_S \leftarrow \emptyset \rightarrow G_T$
CO	$G_S \leftarrow G_C \rightarrow G_T = G$
FWD_OPT	$G_S \leftarrow \emptyset \rightarrow \emptyset$
BWD_OPT	$\emptyset \leftarrow \emptyset \rightarrow G_T$

The starting triple graph of an operation is a *consistent input* for an operation, if it can be extended to a *consistent solution*, i.e., a triple graph that is contained in the language of the TGG, by applying the rules of the operation.

Definition 5.2 (Consistent Input and Consistent Solution)**.**
Given a triple graph grammar $TGG = (TG, \mathcal{R})$*, a starting triple graph* $G_0 = G_S \leftarrow G_C \rightarrow G_T$ *is said to be consistent input iff* $\exists\, G' = G'_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG)$*, such that:*

Operation	*Conditions*
CC	$G_S = G'_S, G_T = G'_T$
CO	$G_S = G'_S, G_C = G'_C, G_T = G'_T$
FWD_OPT	$G_S = G'_S$
BWD_OPT	$G_T = G'_T$

G' is referred to as a consistent solution for G_0 in each case.

Figure 5.2 depicts an example for a consistent triple, which can be generated by applying the rules *StatemachineToMachine* and *PortToVariable* in this order. The source model is depicted to the left with a blue background, the target model to the right with green, and the correspondence model in the middle with yellow. Above the models, the operations that require the respective model as input are listed: The correspondence model is only needed for CO, while all operations except BWD_OPT (FWD_OPT) require the source (target) model as input.

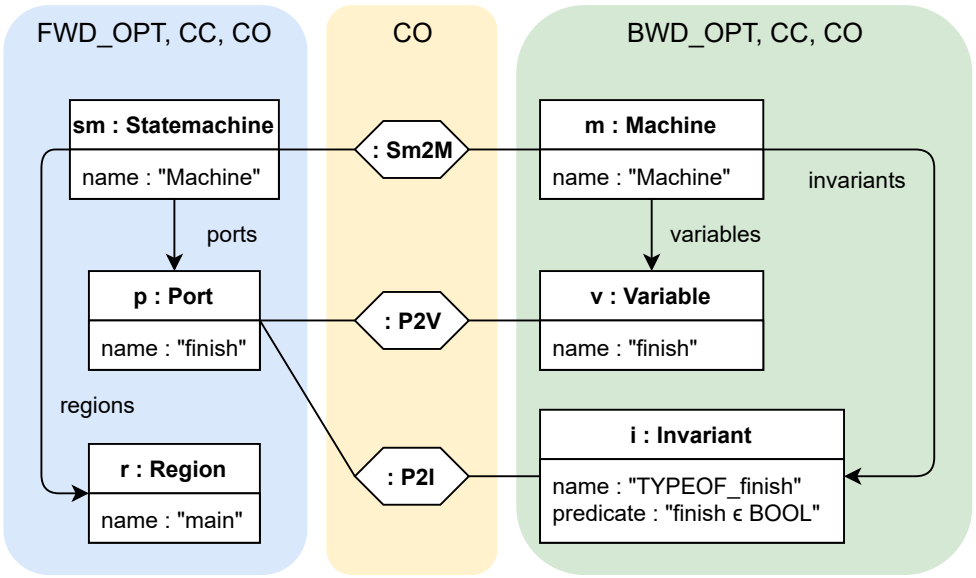


Figure 5.2: Consistent triple indicating which models are required as input per operation

The elements of the given graphs are denoted as *markable elements*, and the elements to be created by the operation as *created elements*.

Definition 5.3 (Markable Elements and Created Elements)**.**
The sets mrkElem(G) and crtElem(G) are defined for a triple graph $G = G_S \leftarrow G_C \rightarrow G_T$ as follows:

Operation	***mrkElem(G)***	***crtElem(G)***
CC	$elem(G_S) \cup elem(G_T)$	$elem(G_C)$
CO	$elem(G)$	$\emptyset$
FWD_OPT	$elem(G_S)$	$elem(G_C) \cup elem(G_T)$
BWD_OPT	$elem(G_T)$	$elem(G_S) \cup elem(G_C)$

Consequently, $elem(G) = mrkElem(G) \cup crtElem(G)$.

In addition to varying input and output, the rules of the TGG have to be suitably manipulated for each operation. To express that these rules partly mark and create elements, we denote them as *operational* rules derived from the original, declarative rules (Def. 5.4). The basic idea for operational rules is to extend the context of the original TGG rule to cover all additional input elements. The green (created) elements of the declarative TGG rule are either marked (in the given input) or created (in the remainder of the triple). An operational rule, derived from a respective declarative rule, requires the markable elements (as specified in Def. 5.3) for the respective operation as additional context, marks them, and adds all created elements on application.

Definition 5.4 (Operational Rule and Marking Elements)**.**
Given a triple rule $r : L \to R$, let L_0 and R_0 be the starting triple graphs for L and R. The ***operational rule*** *$or : OL \to OR$ for r is constructed as depicted in Fig. 5.3. OL is formed via pushout construction of L and R_0 over L_0. It holds that $OR = R$, and $or : OL \to OR$ is induced via the universal property of the pushout. An element $e \in mrkElem(OL)$ is a* ***marking element*** *of or iff $\nexists e' \in mrkElem(L)$ with $r_S(e') = e$ or $r_C(e') = e$ or $r_T(e') = e$.*

Depending on the operation, the left-hand side of the operational rule (OL) can be formed out of the left-hand side of the declarative rule (L), and the starting triple graph for the right-hand side (R_0) via a pushout construction (cf. Fig. 5.3). The right-hand side of the operational rule (OR) is equal to the right-hand side of the original rule. The elements that are added by the morphism $ol : L \to OL$ are denoted as *marking elements* of the operational rule or.

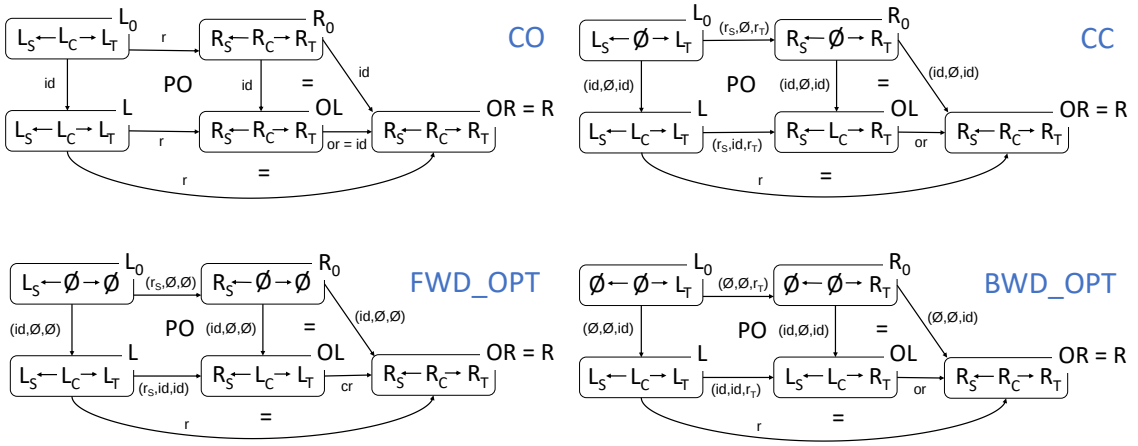


Figure 5.3: Operational rules for CO, CC, FWD_OPT and BWD_OPT

In Fig. 5.4, the construction of the FWD_OPT rule for *TriggerToGuard* is depicted. To improve readability, edge labels are omitted, and the types `T` (transition), `E` (event), `Tg` (trigger) and `G` (guard) are given in their abbreviated form. Based on the graphs L, L_0 and R_0, which can be directly taken from the declarative rule, OL can be determined via pushout construction. The operational rule $or : OL \to OR$ can be derived be requiring that the triangles involving L, OL and OR on the one hand and R_0, OL and OR on the other hand both commute. It becomes apparent that the operational rule for the FWD_OPT operation creates the same elements as the declarative rule in the target and correspondence model, whereas the elements of the source model (`Tg` with its incoming edge) become the *marking elements* of the operational rule.

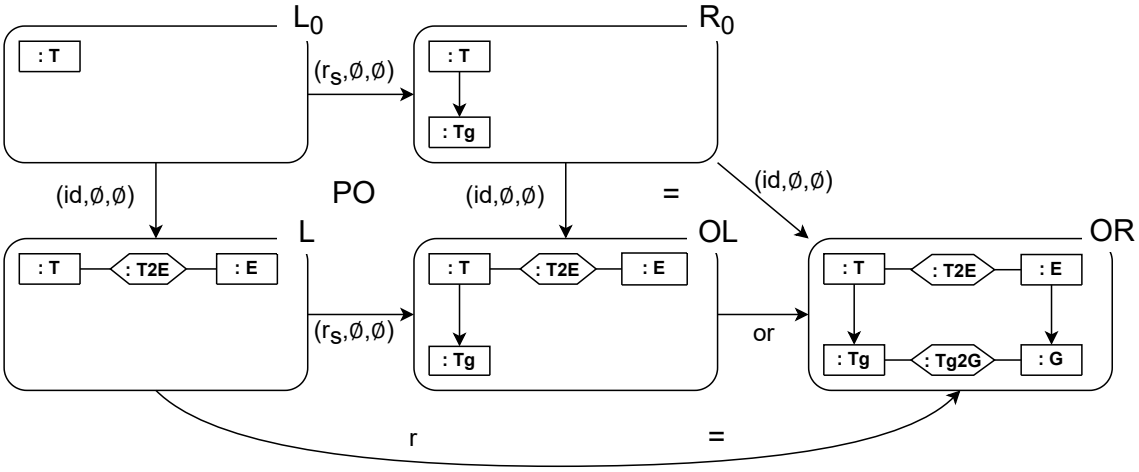


Figure 5.4: Construction of the FWD_OPT rule for TriggerToGuard

Figure 5.5 depicts the operational rules derived from *TriggerToGuard* for each of the four operations in the previously used short-hand notation: Original context elements remain unchanged, whereas green elements are partly replaced by black elements depending on the operation. For CC, only the correspondence link `:T2G` is created, for CO nothing is created, and for FWD_OPT (BWD_OPT) the correspondence link and the guard `g` (trigger `tg`) are created with their incident edges. Elements which are annotated with $\square \rightarrow \boxtimes$ are marked by the operational rule, whereas the $\boxtimes$ annotation indicates that they must be marked already in order to apply the rule. As for declarative rules, created elements have a ++ mark-up and are coloured green, and context elements outside the input are black and do not have any mark-up.

Considering the elements of the host graph that are involved in an operational rule application d, a partition into four sets is possible, as stated in Def. 5.5: Elements can be *created* ($\mathrm{crt}(d)$) or *marked* ($\mathrm{mrk}(d)$) by a rule application, depending on whether they are contained in the starting triple graph. Some elements are *required* as context in both the operational and the original TGG rule. If such a context element is also part of the starting triple graph, it is *required* to be *marked* already ($\mathrm{reqMrk}(d)$), simulating the behaviour of the generative TGG specification. Otherwise, it is required to be created by a previous rule application ($\mathrm{reqCrt}(d)$). To track dependencies between derivations and eventually formulate constraints to characterise correct derivations, we introduce the following sets of marked, created, and required elements of a direct derivation:

Definition 5.5 (Marked, Created and Required Elements)**.**
For a direct derivation $d : G \overset{or@om}{\Longrightarrow} G'$ via an operational rule $or : OL \rightarrow OR$, the following sets are defined:

- *crt(d) = crtElem(G') \ crtElem(G)*
- *mrk(d) = {$e \in$ elem(G) | $\exists\, e' \in$ elem(OL), $om(e') = e$ where e' is a marking element of or}*
- *reqMrk(d) = {$e \in$ mrkElem(G) | $\exists\, e' \in$ elem(OL), $om(e') = e$ where e' is not a marking element of or}*
- *reqCrt(d) = {$e \in$ crtElem(G) | $\exists\, e' \in$ elem(OL), $om(e') = e$}*

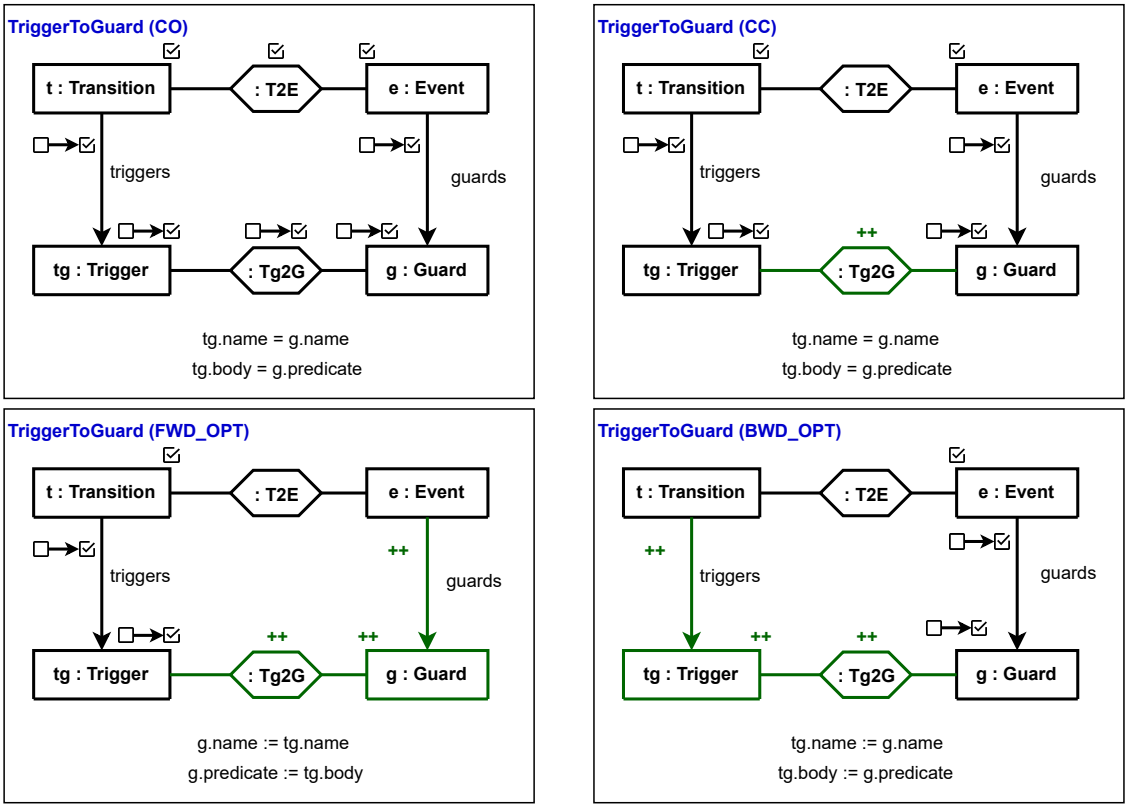


Figure 5.5: Operational rules for TriggerToGuard

5.5 Rule Pattern Matching

The goal of the next step in the process is to (i) determine matches for all operational rules of a chosen operation, (ii) to choose certain matches for rule application, and (iii) to repeat (i) and (ii) until some termination criterion is reached. Typical TGG-based tools apply a greedy strategy by *marking* the elements in each operational rule, which would be created by the original TGG rule, and by storing these markers separately. Such an additional marker data structure helps to increase the probability of generating a consistent result by ensuring that context dependencies are respected.

There is, however, no hard guarantee that greedy strategies cannot fail even for simple examples: We temporarily extend the SysML metamodel, such that `Regions` can be nested (cf. Fig. 5.6). In order to integrate this inheritance relation between `Regions` into the TGG, a new rule *AddSubRegion* (Fig. 5.7) is (temporarily) added to the TGG. Similar to *AddRegion*, this rule only affects the source model.

To show that a greedy choice of forward transformation rules can be problematic, a region `r2` is added to the source model of Fig. 5.2 as a subregion of `r1`. The resulting source model is depicted in Fig. 5.8. Suppose that this source model should be completed to a consistent triple with a greedy forward transformation, i. e., a transformation that applies rules immediately after finding a respective match. Instead of applying *AddSubRegion* to create `r2` as a subregion of `r1`, the rule *AddRegion* could be applied instead, such that the edge that connects `r2` and `r1` remains untranslated. This is a dead end, as the `subregions` arrow (highlighted in red) cannot be translated by any forward transformation rule.

In principle, greedy strategies could correct such wrong decisions by applying backtracking, all TGG tools we are aware of avoid this as it rapidly leads to exponential runtime.

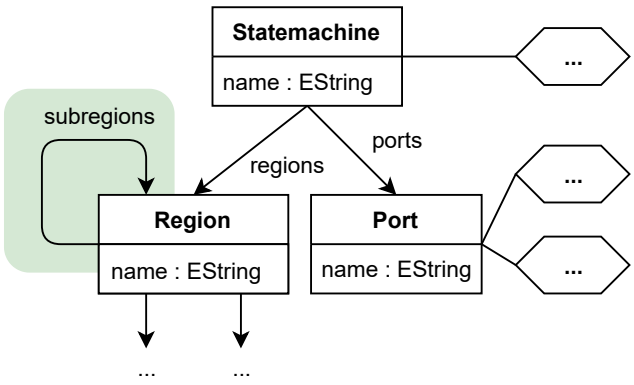


Figure 5.6: Modified source metamodel

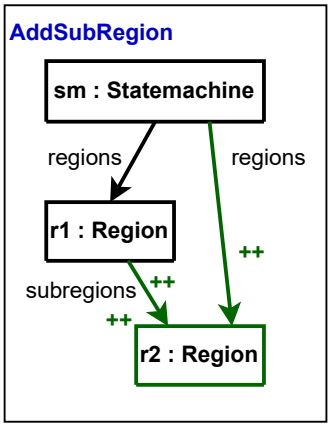


Figure 5.7: Rule: AddSubRegion

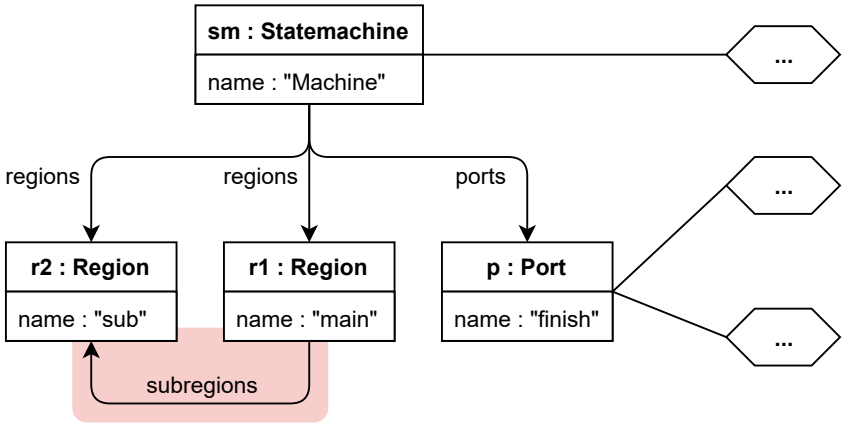


Figure 5.8: Greedy choice of rule applications can lead to dead ends

The alternative solution taken by TGG tools is twofold: (i) numerous (often technical and complex) restrictions are imposed concerning the allowed structure of a TGG and expected input, and (ii) the derived operational rules are enriched with automatically generated application conditions (cf. Def. 4.9) that serve as a check to suitably filter out invalid matches (see, e. g., Fritsche et al. [FLAS17] for a recent and detailed discussion). Unfortunately, there is no guarantee for the derived operations to always return a consistent solution if one exists, as the derived "filter" application conditions might be insufficient to detect whether the transformation can run into dead ends. Our proposed strategy is instead to collect and apply *all* matches for the operational rules as depicted in Fig. 5.5. Obviously this can only yield rule application *candidates* representing a super set of all possible rule applications to be filtered in subsequent steps.

5.6 ILP Construction

In general, there are more rule application candidates that can be found by matching the operational rules than are necessary to form a derivation sequence from the starting triple graph to a consistent solution. The subset of necessary rule applications is therefore determined by transforming the consistency management problem into a search problem to be solved by, e.g., an ILP solver, in which each rule application candidate is associated

to a binary variable. Its value in the retrieved solution (0 or 1) indicates whether the candidate is considered for forming the final derivation sequence.

To filter the rule application candidates collected in the previous pattern matching step, an ILP is generated consisting of constraints guaranteeing that the resulting rule application sequence is *correct*, i. e., results in a consistent triple. As depicted in Fig. 5.9 for the FWD_OPT operation, seven rule application candidates $d_1 \dots d_7$ can be collected. d_1 and d_2 are associated to an application of *StatemachineToMachine* (Fig. 3.11), d_3 and d_4 to applications of *PortToVariable* (Fig. 3.10), d_6 and d_7 to *AddRegion* (Fig. 4.2) and d_7 to an application of *AddSubRegion* (Fig. 5.7). The marked and created elements of each rule application are annotated with the associated variable. Marked elements are coloured black, whereas (potentially) created elements are coloured grey. The reason why there are multiple rule application candidates for translating, e. g., the statemachine `sm`, is the shape of the rule *StatemachineToMachine*: The regions `r1` and `r2` can both be combined with the statemachine `sm`, resulting in two distinct matches for the rule. As we collect all possible rule application candidates, the choice between these candidates for the final solution is made in the optimisation step (Sect. 5.7).

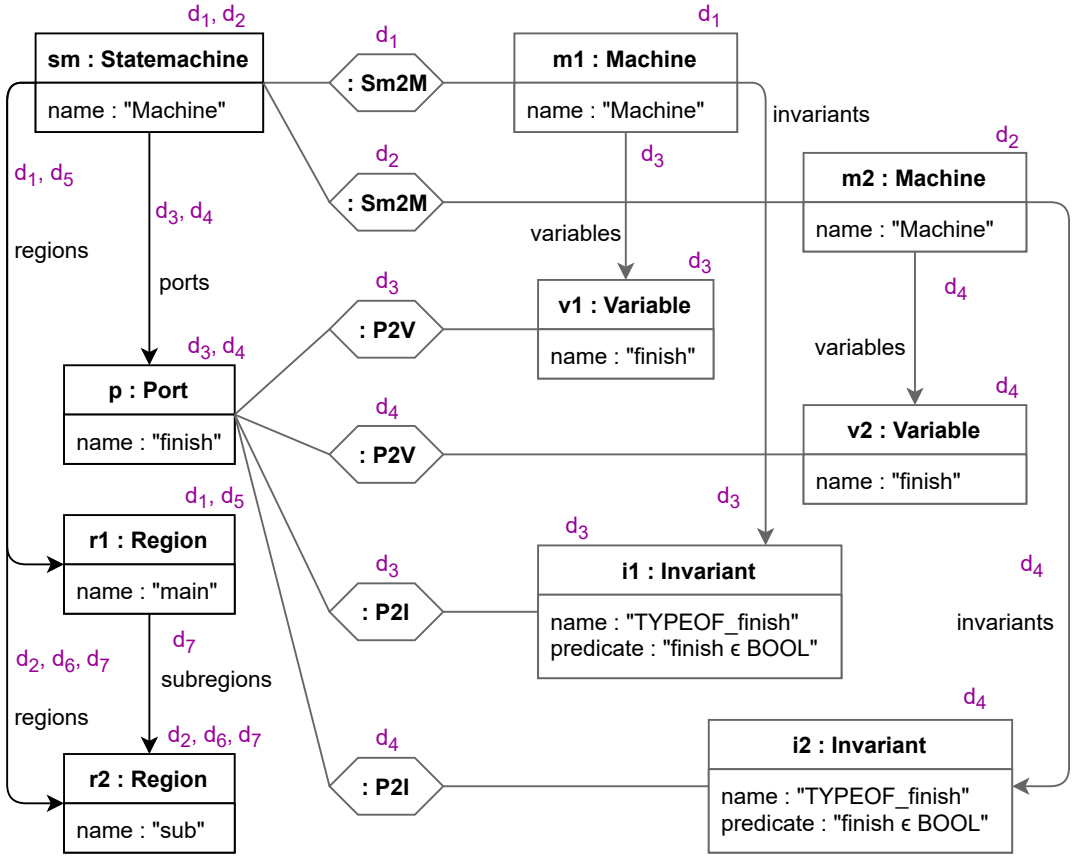


Figure 5.9: Rule application candidates collected for the FWD_OPT operation

In order to translate the consistency management problem into an optimisation problem, multiple constraint types are specified that ensure that a solution to the ILP represents a correct choice of rule applications. All application candidates for operational rules are collected and transformed into constraints that an ILP solver uses to determine a correct and optimal subset. Every rule application is related to a binary variable that is set to 1

if and only if the rule application is chosen as part of the final solution. In the concrete example, the rule applications $d_1 \dots d_7$ are associated to the binary variables $\delta_1 \dots \delta_7$.

Definition 5.6 (Constraints for Derivations)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. For each direct derivation $d_1, \dots, d_n \in \mathcal{D}$, respective binary variables $\delta_1, \dots, \delta_n \in \{0, 1\}$ are defined. A linear constraint $\mathcal{LC}$ for $\mathcal{D}$ is a conjunction of linear inequalities which involve $\delta_1, \dots, \delta_n$. A set $\mathcal{D}' \subset \mathcal{D}$ fulfils $\mathcal{LC}$, denoted as $\mathcal{D}' \models \mathcal{LC}$, iff $\mathcal{LC}$ is satisfied for variable assignments $\delta_i = 1$ if $d_i \in \mathcal{D}'$ and $\delta_i = 0$ if $d_i \notin \mathcal{D}', 1 \leq i \leq n$.

In the following, rule applications and binary variables can be co-located via their indices, i. e., a variable δ belongs to a rule application d, and a variable δ_i to a rule application d_i, respectively. The remainder of this section introduces the three necessary constraint types and the objective function in more detail, which ultimately form the ILP.

Exclusion Constraints

To ensure correctness, i. e., that the final solution triple is a member of the TGG's language, every operational rule application must correspond to a (declarative) rule application of the underlying TGG. As markings in operational rules correspond to the creation of elements in the original rules, it must be prohibited that elements are marked multiple times, because this would mean that an element is created more than once. For each node and edge, a predicate *mrkSum* of type integer is defined that reflects the number of markings per element by counting the rule applications that mark this element.

Definition 5.7 (Sum of Alternative Markings for an Element)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. For each element $e \in elem(G_0)$, let $\mathcal{E}(e) = \{d \in \mathcal{D} \mid e \in mrk(d)\}$. The integer mrkSum(e) denotes the sum of the associated variable assignments for each $d \in \mathcal{E}$:

$$mrkSum(e) = \sum_{d_i \in \mathcal{E}(e)} \delta_i$$

Rule applications that overlap in their marked elements must *exclude* each other as at least one element would be marked twice if more than one of such rules is chosen. Double markings are incorrect as the resulting triple could never be created by the rules of the TGG. To avoid this, the predicate *mrkSum* is used in linear constraints for each element of the starting triple graph to ensure that elements are marked at most once.

Definition 5.8 (Constraint 1: Mark Elements at Most Once)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules:

$$markedAtMostOnce(G_0) = \bigwedge_{e \in elem(G_0)} [\ mrkSum(e) \leq 1]$$

Note that the *mrkSum* predicate is not required to be *equal* to 1: The constraint is also fulfilled if there are elements that are not marked at all, resulting in a $mrkSum$ of 0. Consequently, there are valid solutions that cannot mark the input entirely. The reason for the sum of marked elements not being strictly equal to 1 is the desired treatment of inconsistent input: In this case, the ILP solver can still perform the optimisation, and strive to maximise the number of marked elements.

In Fig. 5.9, the port `p` and the `ports` edge are both marked elements of d_3 and d_4 (rule *PortToVariable*), which means that only one of the two rule applications can be chosen. For each element, we restrict the sum of the variables associated to rule applications that mark this element to 1, which results in a non-trivial linear constraint:

$$\delta_3 + \delta_4 \leq 1$$

The region `r2` and its incoming edge can be even marked by three different rule applications. It can either be part of the axiom rule application of *StatemachineToMachine* (d_2), or be added with an application of *AddRegion* (d_6) or *AddSubRegion* (d_7). As only one rule application candidate can be chosen, the following constraint is added to the ILP:

$$\delta_2 + \delta_6 + \delta_7 \leq 1$$

For all remaining elements, exclusion constraints are constructed in a similar fashion:

$$\delta_1 + \delta_2 \leq 1,\ \delta_1 + \delta_5 \leq 1,$$

Implication Constraints

Another constraint type must ensure that an operational rule is applicable if and only if the respective declarative rule would be applicable in a setting in which the same derivation sequence is followed. For each rule that contains context elements, a requirement for its application is that these context elements are created by some previous rule application(s). This means that context elements of the starting triple graph must be marked, and context elements of the remaining part of the triple must be created already. The dependent rule application thus *implies* all rule applications providing at least one context element. The following constraint type ensures that the required context elements for an operational rule application are provided in the final solution, such that the original TGG rule is guaranteed to be applicable in this situation.

Definition 5.9 (Constraint 2: Guarantee Context for Derivations)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. For each direct derivation $d \in \mathcal{D}$, the following constraints are defined:

$$context(d) = \bigwedge_{e \in req(d)} [\delta \leq mrkSum(e)] \wedge \bigwedge_{d_j \in \mathcal{D}, [reqCrt(d) \cap crt(d_j) \neq \emptyset]} [\delta \leq \delta_j]$$

$$context(D) = \bigwedge_{d \in \mathcal{D}} context(d)$$

For an intuitive understanding of these constraints, it is best to assume that the constraint markedAtMostOnce (Def. 5.8) already holds, i. e., elements are marked by at most one derivation. For created elements, note that the transformation process guarantees inherently that every created element is created by only one derivation (see Def. 3.4). context(d) ensures that all marked elements required by d are indeed present (first part), if d is selected, and that all created elements required by d are created in the final result (second part). In the example instance of Fig. 5.9, $d_3 \ldots d_6$ are dependent on d_1 or d_2, as the statemachine `sm` and the machine `m` are required by each of the other rules. These requirements can be expressed as inequalities to be added as constraints to the generated ILP:

$$\delta_3 \leq \delta_1,\ \delta_4 \leq \delta_2,\ \delta_5 \leq \delta_2,\ \delta_6 \leq \delta_1, \delta_7 \leq \delta_1$$

Additionally, d_7 requires the region `r1` to be marked by either d_1 or d_5, leading to the following constraint:

$$\delta_7 \leq \delta_1 + \delta_5$$

As $\delta_1 \dots \delta_7$ are binary variables, the constraints forbid a variable to be set to 1 (i. e., to choose the associated rule application) if none of the rule applications that could provide the required context elements is chosen. The additional constraint for δ_7 could be removed from the ILP in practical implementations, because this constraint is covered by the stricter constraint $\delta_7 \leq \delta_1$.

Cyclic Markings

Furthermore, there are constellations in which rule application candidates mutually provide context for each other by marking or creating elements that are necessary to apply the respective other rule. In this manner, a *dependency cycle* is formed, such that none of the involved rules can ever be applied first due to missing context elements. To express a cyclic dependency, a relation $\rhd$ among rule applications is introduced in Def. 5.10. Each subset $\{d_1, \dots, d_n\}$ of rule applications that does not contain a cycle can be sequenced over this relation in a proper order.

Definition 5.10 (Dependency Cycles)**.**
Let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations. Relations $\rhd, \rhd^M, \rhd^C \subseteq \mathcal{D} \times \mathcal{D}$ between $d_i, d_j \in \mathcal{D}$ are defined as follows:

$$d_i \rhd^M d_j \text{ iff } reqMrk(d_i) \cap mrk(d_j) \neq \emptyset$$

$$d_i \rhd^C d_j \text{ iff } reqCrt(d_i) \cap crt(d_j) \neq \emptyset$$

$$d_i \rhd d_j \text{ iff } (d_i \rhd^M d_j) \vee (d_i \rhd^C d_j)$$

*A set $cy \subseteq \mathcal{D}$ with $cy = \{d_1, \dots, d_n\}$ of direct derivations is a **dependency cycle** iff $d_1 \rhd \dots \rhd d_n \rhd d_1$.*

The first relation ($\rhd^M$) holds if d_i requires an element to be marked and d_j marks it. Likewise, the second relation ($\rhd^C$) holds if d_i requires an element to be created and d_j creates it.

The following constraint forbids setting all variables that form a dependency cycle to 1 in the final solution. As not all rule applications involved in a dependency cycle can be chosen for the final solution, it is always possible to arrange the remaining rule applications properly.

Definition 5.11 (Constraint 3: Forbid Dependency Cycles)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with the underlying set $\mathcal{D}$ of direct derivations, and let $\mathcal{CY}$ be the set of all dependency cycles $cy \in \mathcal{D}$. A linear constraint acyclic(D) is defined as follows:

$$acyclic(D) = \bigwedge_{cy \in \mathcal{CY}, cy = \{d_1, \dots, d_n\}} \sum_{i=1}^{n} \delta_i < n$$

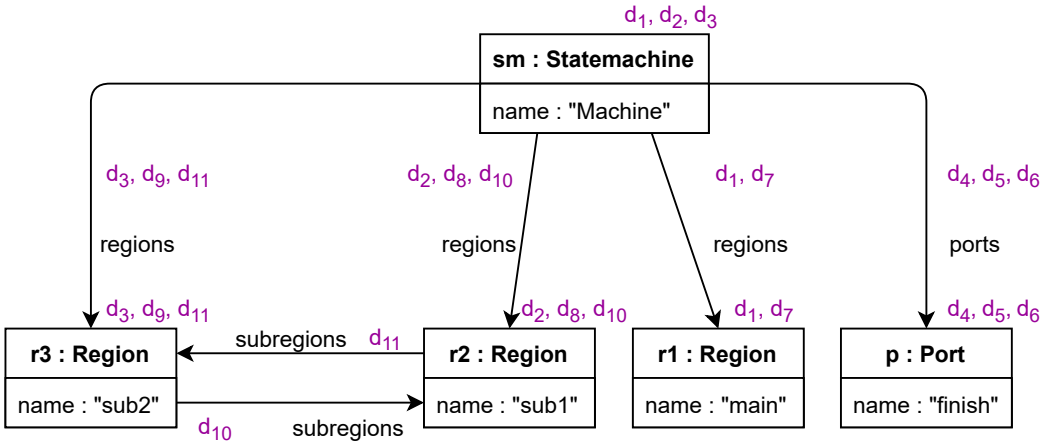


Figure 5.10: Inconsistent source model due to cyclic dependencies

To illustrate this with an example, a third region `r3` with a cyclic inheritance relation to the region `r2` is temporarily added to the example instance as depicted in Fig. 5.10.

This makes the input source model for the FWD_OPT operation inconsistent, as mutual inheritance between two regions is excluded by design: The rule *AddSubRegion* always creates a new region when `subregions` edges are created, such that only tree-like structures can be constructed. When collecting all possible rule application candidates, the two regions can, however, be marked by both *AddRegion* (d_8, d_9) and *AddSubRegion* (d_{10}, d_{11}), whereby d_{10} and d_{11} mutually provide context for each other (and therefore form a dependency cycle). As the two rule applications do not mark any common element, no constraint prevents us from choosing both rule applications, which indeed leads to a solution that contains both d_{10} and d_{11}. Consequently, an additional constraint is necessary to forbid such cycles:

$$\delta_{10} + \delta_{11} < 2$$

As a result, it is not possible to simultaneously choose d_{10} and d_{11}, such that one of the `subregions` edges remains unmarked. In the following, however, we will continue with the instance of Fig. 5.9 to keep the example as simple as possible.

5.7 Optimisation and Filter

In the optimisation step, the generated ILP is solved as a maximisation problem. The default function is a weighted sum of all binary variables, with coefficients reflecting the number of elements marked by the associated rule applications. The reason for this default choice is that a consistent result can only be obtained if *all* input elements can be marked, maximising the objective function. By combining all our correctness constraints, we can now express our optimisation goal: The number of marked elements of the starting triple graph shall be maximised, while ensuring that no correctness constraints are violated.

Definition 5.12 (Optimisation Problem)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules. The ILP to be optimised is constructed as follows:

$$max. \sum_{d \in D} |mrk(d)| \cdot \delta \ s.t.$$
$$markedAtMostOnce(G_0) \wedge context(D) \wedge acyclic(D)$$

For executing the FWD_OPT operation on the source model of Fig. 5.9, the objective function is determined by counting the number of marked elements per rule application:

$$\text{max. } 3\delta_1 + 3\delta_2 + 2\delta_3 + 2\delta_4 + 2\delta_5 + 2\delta_6 + 3\delta_7$$

Setting δ_1, δ_3 and δ_7 to 1 and all other variables to 0, an objective function value of 8 is reached, which is equal to the number of elements in the source model. A consistent forward transformation can thus be performed by choosing these three rule applications.

In a final step, the solution of the ILP is used to choose the set of rule applications to form the transformation result (d_1, d_3 and d_7 for the running example). All elements created by all other rule applications are deleted to produce the final triple, whereas the unmarked elements of the input models are returned separately. If the objective function value of the optimal solution is not equal to the number of input model elements, i. e., if elements remain unmarked, both input and solution are considered as inconsistent. The chosen rule application sequence represents a *best possible* solution and we claim that this (together with an inconsistency report that lists the remaining elements) is substantially more useful than simply terminating with an error and rejecting the input as being invalid.

While this section formally defined the hybrid, fault-tolerant consistency management approach based on TGGs and ILP solving, a proof for correctness, completeness and termination is left open. These properties will be investigated in Sect. 6.5, after extending the approach by graph constraint handling.

5.8 Evaluation

The motivation for synergetically combining algorithmic and search-based concepts is achieving flexibility and scalability at the same time. While flexibility is provided by encoding the graph problem into a generic optimisation problem, this section investigates the scalability of the hybrid approach. We evaluate our implementation by investigating the following research questions:

RQ1 How scalable (with respect to the model size) are TGG+ILP-based operations compared to greedy, TGG-based operations?

RQ2 Which TGG+ILP-based operation scales best (worst)?

RQ3 How does the scalability of our hybrid approach compare to other algorithmic and search-based approaches?

Setup: Our hybrid approach has been integrated into the components eMoflon::IBeX (Sect. 9.4) and eMoflon::Neo (Sect. 9.5). This evaluation is based on the implementation in IBeX in order to avoid biased results for RQ3 due to the underlying technology, as IBeX is more similar to three tools that were chosen for the comparison with respect to its software architecture than Neo (cf. Chap. 9).

Besides the four ILP-based operations CC, CO, FWD_OPT and BWD_OPT, the standard greedy operations FWD and BWD were included in the analysis to investigate RQ1. Five TGGs that differ considerably with respect to metamodel size, number of rules, and average rule size were chosen for the evaluation. All TGGs are standard examples to demonstrate BX approaches[4]. As examples for small TGGs, *CompanyToIT* [Lau13] and *ClassDiagramToDatabaseSchema* [BRST05] consist of four rules with about 10 elements

[4] http://bx-community.wikidot.com/examples:home

(nodes and edges) each, and metamodels with only about 12 elements (classes, references, and attributes) each. CompanyToIT is designed to be challenging in the forward direction: one of the rules yields a lot of matches, of which only one is required and chosen. ClassDiagramToDatabaseSchema requires choosing between multiple alternatives for translating classes of an inheritance hierarchy in the forward direction, similar to the choice between the rules *AddRegion* (Fig. 4.2) and *AddSubRegion* (Fig. 5.7) in the running example.

Larger metamodels and more complex rules are used in the remaining three TGGs: *BlockCodeAdapter*[5] has a weakly typed source metamodel (27 elements) and a smaller, strongly typed target metamodel (11 elements), meaning that many rules depend on attribute conditions as opposed to typing information. *JavaToUML* [Leb16] has rather large, realistic metamodels with 339 and 837 elements, respectively, and an average rule size of 24 elements. Finally, to investigate RQ3, we chose the *FamiliesToPersons* example [ABW17], a standard benchmark for BX languages. For this benchmark, various solutions have been implemented already [ABW+20], which can be used to represent purely search-based and algorithmic approaches.

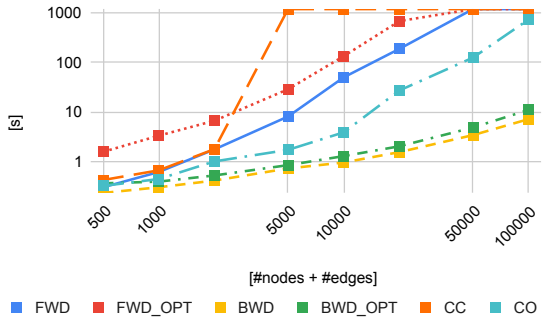
All measurements were conducted for model sizes from 500 to 100,000 elements. For the first four examples, models were generated randomly using a model generator (cf. Sect. 9.4), while the benchmark example has a fixed generation procedure. For each operation and model size, the median of 10 non-consecutive repetitions was taken to minimize the effect of outliers. Using a time-out of 20 minutes, all performance tests were executed on a desktop computer with an Intel Core i5 (3.20 GHz), 16GB RAM, and OS X 64-bit as operating system. An installation of Eclipse Modelling Tools, version Oxygen 4.7.2 with Java Runtime Environment (JRE) version 1.8.0_101 was used. The Java Virtual Machine (JVM) running the tests was allocated a maximum of 12GB memory. For all ILP-based operations, the time required for ILP solving is negligible compared to the total time and is thus not measured separately (cf. Leblebici et al. [LAS17] for similar results in this regard).

Results: We briefly present the evaluation's outcome using one diagram per TGG showing the execution times of the different operations depending on the model size. Figure 5.11 depicts the measurement results for the four example TGGs. The exact numbers for each test run are available online[6] and have already been presented in prior work [WALS19].

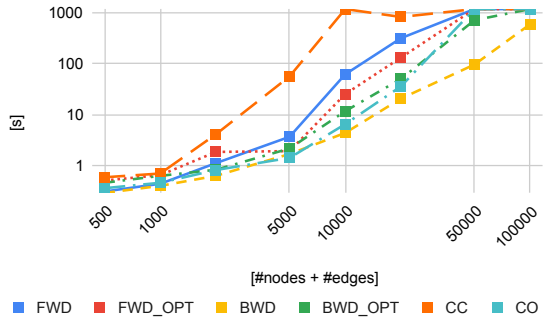
As expected, the forward direction is more challenging for *CompanyToIT* than backward for both greedy and ILP-based operations. FWD_OPT is about 10 times slower than FWD for 500 elements, but this factor decreases to about 3 for 20k elements. BWD_OPT and BWD scale comparably well, with a small and constant factor. CC explodes already for 5k elements, while CO scales better than forward but worse than backward transformations. For *ClassDiagramToDatabaseSchema*, CC is again by far the most time-consuming task. Ignoring the anomaly for FWD caused by a larger spread in the measured runtime values, resulting in a rather misleading median, both greedy and ILP-based strategies perform comparably well, with greedy strategies being a bit faster.

For *BlockCodeAdapter*, the forward direction is more challenging as this involves parsing the weakly typed source model. Interestingly, FWD_OPT actually outperforms FWD for some model sizes, probably because the heuristics and checks used for greedy strategies are designed for strongly typed, well-connected models. In the backward direction, BWD is clearly faster than BWD_OPT. For *JavaToUML*, all operations (even including CC and CO) scale comparably well taking only between 5s and 10s for 100,000 elements. While the greedy strategies are still the fastest, the difference is not really substantial.

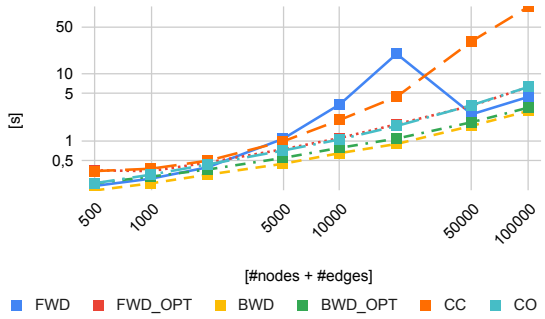
[5] https://github.com/eMoflon/emoflon-ibex-tests/tree/master/BlockCodeAdapter
[6] https://docs.google.com/spreadsheets/d/1HAHP4trkg91LHvrKX2eVupcO02SOsjnh957gRaG_mds



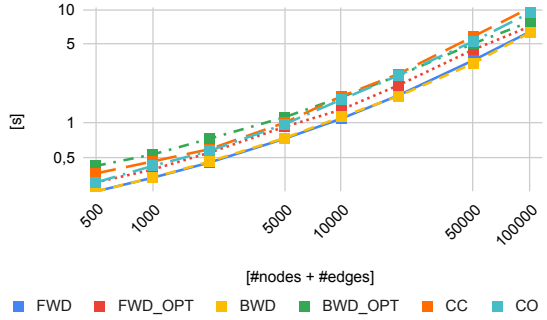
(a) CompanyToIT



(c) BlockCodeAdapter



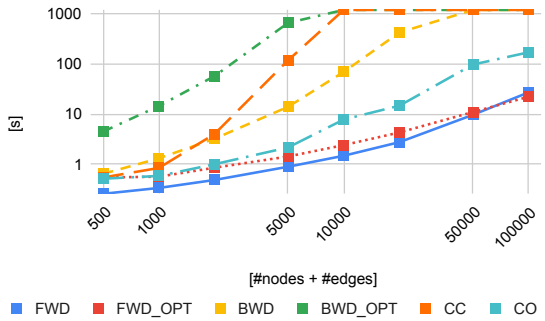
(b) ClassDiagramToDatabaseSchema



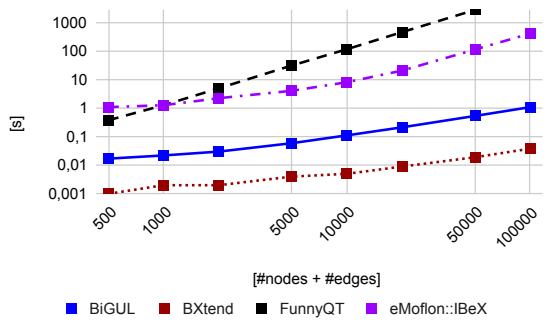
(d) JavaToUML

Figure 5.11: Comparison of greedy and ILP-based operations

To investigate RQ3, we chose BXtend [Buc18] as a pragmatic, manually optimised, algorithmic solution (EMF, Xtend), BiGUL [KZH16] as a lens-based algorithmic solution (Haskell), and FunnyQT [Hor17] as a search-based solution for the FamiliesToPersons benchmark [ABW17]. As the backward direction of the benchmark is highly nondeterministic and requires an integration of user preferences, we were only able to compare our FWD_OPT operation with these algorithmic and search-based approaches (Fig. 5.12). As can be expected for pure algorithmic approaches, BiGUL and especially BXtend scale excellently up to 100,000 elements. While our FWD_OPT operation is actually slower than FunnyQT for small models, the search-based FunnyQT is already a factor 10 slower for 10,000 elements, and times out for 100,000 elements elements.



(a) Different ILP-based operations



(b) Comparison with other tools

Figure 5.12: The FamiliesToPersons benchmark example

Summary: Coming back to our research questions:

(RQ1) While it is clear that greedy strategies scale far better than our hybrid operations, it is surprising that FWD_OPT and BWD_OPT can actually compete with their greedy counterparts for larger models and realistic metamodels. In many cases, the factor between greedy and ILP-based forward / backward transformations is acceptable and does not appear to explode with model size. Our explanation is that while the set of collected solution candidates can theoretically explode, realistic metamodels and patterns ensure that the pattern matcher can already filter relatively well so the superset of all solutions remains manageable. According to our observations, the use of a pattern matcher reduces the complexity of the optimisation problem by far.

(RQ2) Amongst our ILP-based operations, CC is clearly the most time-consuming operation. This is to be expected, as the solution candidates are "all possible pairs" of source and target matches, resulting in a large solution space. CO appears to be much easier, sometimes even faster than forward or backward depending on the particular TGG. This is also to be expected as patterns for CO cover entire triples and thus avoid the combinatorial explosion experienced by CC.

(RQ3) While the price of our hybrid approach is substantial – algorithmic approaches, especially manually programmed solutions can be considerably faster and scale better – our evaluation still indicates that our approach scales (potentially much) better than pure search-based approaches as we perform most of the work using a graph pattern matcher.

Threats to Validity: The chosen examples – except for JavaToUML – do not involve metamodels of realistic size and require only few TGG rules. Generalising our results (RQ1, RQ2) to real-world scenarios thus requires a more extensive and systematic evaluation. Our comparison to other model transformation tools is also restricted to only three other tools, and only one benchmark example in the forward direction. Generalising our results for RQ3, therefore, also requires further tests with other tools and benchmark examples to confirm our indications.

5.9 Summary and Discussion

We presented a hybrid solution to consistency management combining TGGs as an algorithmic and ILP as an exact search-based approach. The consistency checking approach by Leblebici et al. [LAS17] has been extended towards further variants with given correspondence links, as well as towards forward and backward transformation with the same uniform process and formalism. A simple example could demonstrate that there are cases in which greedy strategies can fail to transform consistent input models, whereas our approach is able to complete the input to a consistent model triple. With a performance evaluation, we show that the hybrid approach scales reasonably well for small and medium-sized models. Correspondence creation turns out to be the most challenging operation, while for forward and backward transformations, our hybrid approach is slower than algorithmic but faster than search-based approaches, providing a good compromise between flexibility and scalability.

As the choice of the final rule application sequence is transformed into an optimisation problem that is solved to yield the optimum solution with respect to the number of translated elements, our approach returns a correct solution if it exists. When terminating with an inconsistent result, a solution that covers a maximally consistent sub-graph of the input is determined along with a delta of nodes and edges for which no operational rule applications could be determined. Thereby, a high degree of fault-tolerance is achieved because an optimum solution (with respect to the number of translated input elements) is

computed regardless of the input models' consistency. Although the default configuration for the objective function simply maximizes the number of marked elements in the input models, it can be flexibly configured to pursue different optimisation goals (e. g., by giving more important node types a higher weight).

The initial design of the fault-tolerant consistency management approach opens up several possibilities for extensions, which will be partly dealt with in the remainder of this thesis. First, motivated by the analysis of TGG language features in Chap. 4, the ILP-based approach is extended towards support for graph constraints in the upcoming Chap. 6. At this point, we will show that the formal proof for correctness and completeness of the correspondence creation operation [Leb18] can be transferred to the three new operations with the additional requirement of schema compliance. Second, it seems promising to alter the construction of the objective function in situations where marking the entire input is not the ultimate goal: This holds – as already mentioned – for concurrent synchronisation scenarios, where keeping deleted elements has even a negative influence on the solution quality. The task of synchronising concurrent updates will be investigated in detail in Chap. 7 and 8. Also, it makes sense in some application scenarios to introduce weightings depending on, e. g., the node type, as we will investigate in a practical use case in Chap. 12. With respect to the tool implementation, further scalability tests will compare the runtime consumptions for all four operations in eMoflon::IBeX and eMoflon::Neo (Chap. 9) to demonstrate which technology tends to be more suitable for which task.

6 Integrating Domain Constraint into the Fault-Tolerant Framework

With the first version of the conceptual framework of Chap. 5, model transformations in the presence of faults are possible. In Sect. 4.5, we have seen that application conditions increase the expressive power of TGGs, and help domain and integration experts to ensure that domain constraints are respected by the transformation engine. In this chapter, support for graph constraints is added to the hybrid framework, such that domain constraints can be expressed separately from the TGG in use. While this is considered to be more intuitive for users, Sect. 4.5 has shown that graph constraints and application conditions can be converted into one another. Scalability tests show that negative constraints can be handled efficiently by our implementation, whereas there is room for improvement with respect to implication constraints.

The rest of the chapter is structured as follows: After a brief motivation for this extension in Sect. 6.1, our contribution is compared with related work in Sect. 6.2. Section 6.3 gives an overview of the solution approach, before the formal framework is extended towards support for graph constraints and applied on an example for consistency checking in Sect. 6.4. Correctness and completeness for all four operations are shown in Sect. 6.5. The results of an experimental runtime evaluation are presented in Sect. 6.6. Finally, Sect. 6.7 summarises the contributions and discusses them with regard to fault-tolerant consistency management.

6.1 Motivation

To be suitable for real-world use cases, a transformation language has to be sufficiently expressive; increasing expressiveness while still guaranteeing all formal properties is an open challenge for ongoing research on MDE in general and TGGs in particular [ALK+15]. In Chap. 4, it became apparent that the use of *graph constraints* increases the expressive power of TGGs. In this chapter, we demonstrate how the fulfilment of domain constraints by TGG-based consistency management operations can be guaranteed. We denote this property in the rest of this thesis as *schema compliance*.

A TGG tool is schema compliant if it can take domain constraints into account when performing consistency management tasks. Domain constraints can be formalised as graph constraints (cf. Def. 4.12) and include negative constraints (forbidding certain situations), positive constraints (demanding certain patterns), and constraints enforcing implications between graph patterns. Besides these three types which will be integrated into the fault-tolerant framework, there are also more complex constraint types that allow the nesting of conditions, and thereby reach the expressive power of first-order logic [HP09].

A well-known example for domain constraints are *multiplicity constraints*, which are specified in the metamodels of the respective domains. Suppose that for an association, a multiplicity of `m..n` shall be respected. The upper bound can be guaranteed by forbidding `n+1` occurrences of the respective element. In turn, the lower bound of `n` can be demanded with an implication constraint (as soon as such an association exists, there must be a match for `n` elements).

Most TGG tools only allow the user to introduce constraints indirectly, by attaching Application Conditions (ACs) to rules to restrict their applicability. In Sect. 4.5, we have seen that graph constraints and application conditions can be transformed into one another and therefore serve the same purpose in rule-based consistency management. This strategy is, however, problematic for at least two reasons: First, ensuring compliance to a sufficiently expressive schema for all previously mentioned derived operations is still an open challenge; to the best of our knowledge, all existing TGG tools only support a very restricted subset of application conditions. Second, it is conceptually demanding for the user to indirectly specify domain constraints as application conditions, especially because this has to be completely revisited every time the TGG or domain constraint is changed. Although there is prototypical tool support for generating these application conditions automatically from a given set of constraints, the use of application conditions involves noticeable cognitive efforts. Therefore, it is advisable to provide direct support for schema compliance, which is not achieved by any approach that we are aware of, though.

To address these limitations, we extend the hybrid approach of Chap. 5 towards *schema compliance*, i. e., the support for graph constraints, for all four previously introduced consistency management operations. We can guarantee under very weak assumptions that a consistent solution is found by our approach if and only if one exists. We thereby show that *correctness* and *completeness* can be proven for all four consistency management operations even when schema compliance is to be additionally guaranteed. The proof for consistency checks without additional constraints [Leb18] can be adapted to the described setting, substantially benefiting from the uniform definition of the four operations.

Due to the flexibility of the search-based approach, we take a further step towards fault-tolerant consistency management as all supported operations terminate with a maximal partial solution that is contained in the language of the underlying TGG and respects all posed domain constraints. Conventional TGG-based approaches, in contrast, often separate checking domain constraints from the transformation, such that the user is forced to fix all constraint violations before the actual transformation task can be started. An implementation and experimental evaluation supports our claim of practical applicability.

6.2 Related Work

Numerous approaches to model transformation that take additional constraints into account have been presented already, partly addressing the issue of constraints for multiple domains. Cuadrado et al. translate constraints of the target model to the source model, such that the result of a forward transformation is guaranteed to satisfy all posed constraints [CGdL+17]. The approach was implemented in the anATLyser tool and applied to real-world model instances. OCL is used to define constraints, while the transformation is specified using the Atlas Transformation Language (ATL). Compared to our approach, the supported constraints are more expressive, whereas the transformation is unidirectional. The focus is rather set on forward and backward transformations than on supporting a wide range of operations with the same consistency definition.

Cabot et al. generate OCL invariants from TGG rules and Query/View/Transformation-Relations (QVT-R) specifications to verify and validate model transformations [CCGdL10]. Both the metamodel and the derived invariants can be used to check whether models are well-formed, which resembles the notion of consistency used in this thesis. However, the transformation is decoupled from checking the invariants, whereas our approach integrates both tasks into a uniform algorithm.

To the best of our knowledge, all existing TGG-based approaches ensure schema compliance indirectly by equipping TGG rules with semantically equivalent ACs. Ehrig et al. introduced NACs to TGGs and proved correctness and completeness for unidirectional model transformation [EHS09]. The formal framework was extended by Golas et al. [GEH11] towards general ACs for TGGs. The approach is restricted to the declarative specification of TGGs, though, enabling the rule-based generation of models that adhere to ACs, whereas an operationalisation is left to future work. In neither of the approaches, the formalisation is at this point backed up by an implementation that could show whether the concepts are applicable in practice.

This open challenge was subsequently addressed by Klar et al. [KLKS10], while restricting the class of supported NACs to those which are only used to guarantee schema compliance. A translation algorithm with polynomial runtime was presented that proves this class of NACs to be efficiently supported in practice, whereby correctness and completeness of this strategy can still be guaranteed. The semantic equivalence to negative constraints together with a TGG was shown by Anjorin et al. [AST12] by proposing a constructive algorithm for generating such NACs from negative constraints. While the formalisation and implementation was initially restricted to (unidirectional) model transformation, Leblebici et al. [LAF+17] showed evidence that these concepts can be efficiently transferred to (incremental) model synchronisation.

Hildebrandt et al. propose a static analysis technique for integrating OCL constraints with TGGs [HLBG12]. The approach is implemented as an extension to the TGG-based model transformation tool MoTE. Transformation and constraint checks are decoupled, and only a subset of the OCL is covered, such that the expressiveness of the supported constraints is equal to those in our approach.

Overall, these approaches either support only a subset of ACs or are restricted to a single operation. General ACs, for instance, are only specified for declarative TGG rules, and consistency checks remain totally unaddressed in this regard. Furthermore, all NACs are required to be "domain separable", i.e., restrict the applicability of a rule either for the source or the target model. In contrast, our approach can handle general graph constraints that are also allowed to range over multiple domains including the correspondence model (cf. Fig. 6.2).

There are also purely constraint-based approaches that encode both model structure and consistency relation into constraints and can easily handle schema compliance. The JTL framework supports several consistency management operations by deriving constraints from the input models and computing a valid solution via answer set programming [EPT18].

In recent work on integrating constraints into algebraic graph transformation [KSTZ20], Kosiol et al. propose to consider consistency as a continuous measure rather than a binary decision. While our approach uses the number of elements in the maximal consistent subtriple to measure consistency, the authors consider the ratio between all occurrences of a constraint pattern in the model and the number of violations of this constraint. It is shown that graph transformation rules can be classified as consistency sustaining or improving, which means that their application has a non-negative or positive effect on the model consistency.

Semeráth and Varró developed a strategy for checking constraints for partial models, i.e., models that involve uncertainty. An early detection of potential or guaranteed violations of well-formedness constraints is implemented via graph pattern matching on the partial model [SV17]. Similarly, different solvers were used to generate consistent models, for which it can be guaranteed that structural and attribute constraints are respected [SBL+20, BSV20, MSBV20].

Both approaches are restricted to intra-model consistency, and the results are not directly transferable to a bidirectional scenario, though. The high degree of flexibility of constraint-based approaches, and their potential to support more expressive constraints, such as nested graph constraints, however, comes at the price of scalability, leading to insufficient runtime performance for models of realistic sizes [ABW+20]. Our hybrid approach uses the flexibility of constraint solvers while scaling comparably well, as constraints are formed on the level of rule applications, keeping the size of the optimisation problem manageable.

6.3 Solution Overview

For handling graph constraints, the work-flow of the hybrid framework must be extended by additional steps. The basic work-flow for ILP-based consistency management with constraints is depicted in Fig. 6.1 as an extension of the process without constraints (Fig. 5.1). New objects and activities are highlighted in green. Besides the declarative rules and the input models, the set of graph constraints must be handed over to the operation, such that (potential) matches for these constraint patterns can be collected (B2). This step must succeed the pattern matching step for rule applications, because constraint pattern matches cannot only involve elements of the input models, but also elements that are created while collecting rule application candidates. As a result, a superset of matches for constraint patterns is created, whereby the validity of each match clearly depends on the choice of rule applications. The constraint pattern matching step is not iterative, because no new elements are created here. Based on the collected rule application and constraint match candidates, the ILP can be constructed (C).

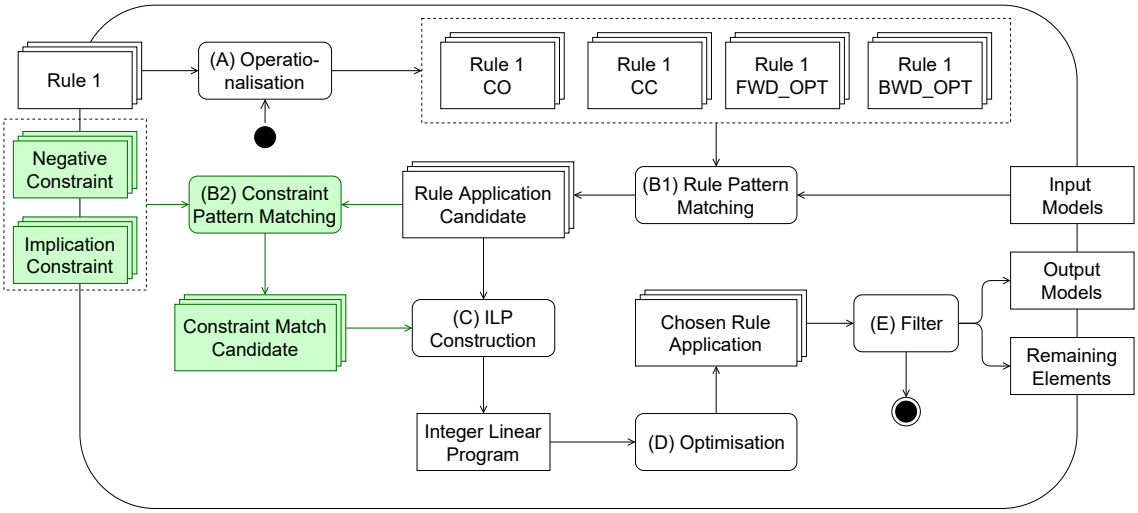


Figure 6.1: Work-flow for fault-tolerant consistency management with constraints

To make these adaptions more concrete, the new concepts will be described formally and illustrated with examples in the next section.

6.4 Integrating Graph Constraints

In this section, the formal framework of Chap. 5 is extended towards graph constraints to increase the expressive power of the class of supported TGGs. Instead of attaching application conditions (cf. Sect. 4.5) directly to TGG rules, global graph constraints are

used, which have to be satisfied by the output models. This procedure is advantageous for the user, because graph constraints are usually easier to grasp than application conditions. A frequently mentioned use case for which additional constraints are needed is the specification of upper and lower bounds for multiplicities in UML class diagrams.

In the following, the formal framework is enriched with definitions to make it applicable in a setting with graph constraints. In particular, (i) the definition of consistent input and solution is extended to graph constraints, (ii) we define how created and marked elements provide context for premise and conclusion patterns, and, based on this redefinition, (iii) the ILP constraints for guaranteeing context are extended. In particular, we create an ILP that maximises the number of marked elements via a suitable objective function, with linear constraints guaranteeing language membership and graph constraint satisfaction.

Besides conformance to the metamodels depicted in Fig. 3.8, we restrict the set of consistent triples for our running example by requiring five additional graph constraints to be satisfied (Fig. 6.2). Two constraints each ensure that the upper and lower bounds of the source and target associations between `Transition` and `State` are respected, whereas the fifth constraint ranges over all three models, and ensures that default values are assigned to variables.

- We *forbid* that a transition `t` has two (or more) source or target states (`s1`, `s2`) with the constraints *NoTwoSourceStates* and *NoTwoTargetStates*, respectively. These constraints are denoted as *negative constraints*.
- Likewise, we enforce that each transition `t` has at least one source and target state `s` via the constraints *TransitionHasSourceState* and *TransitionHasTargetState*. Here, the form of an *implication constraint* is visible: The premise for both constraints is `t`, while the state `s` extends the patterns to form the conclusions in both cases.
- The aforementioned constraints ensure that the produced models fulfil the multiplicity constraints for the source and target edges in the SysML metamodel. Graph constraints can also have a more complex structure, as shown by the implication constraint *AllStateVariablesHaveDefaultValues*. This constraint states that as soon as an initialisation transition `t` exists, for each state of the respective region `r`, the initialisation event `e` must have an action `a` that initialises the respective variables with either `TRUE` or `FALSE`. Compared to the multi-amalgamated rule *Pseudostate-ToActions* (cf. Fig. 4.13), this constraint serves the same purpose, i. e., that each variable is assigned an initial value.

With the help of graph constraints, *schema compliance* can be defined as a second part of our consistency requirement. A schema consists of a type triple graph TG and a set $\mathcal{GC}$ of graph constraints. In the running example, the triple of metamodels in Fig. 3.8 together with the constraints in Fig. 6.2 form such a schema. A (triple) graph *complies* to a schema if it is typed over TG and fulfils all graph constraints in $\mathcal{GC}$.

Definition 6.1 (Schema Compliance)**.**
A schema is a pair $(TG, \mathcal{GC})$ of a type triple graph TG and a set $\mathcal{GC}$ of graph constraints. For a graph constraint $gc \in \mathcal{GC}$, $G \models gc$ means that G satisfies gc according to Def. 4.8. Let $\mathcal{L}(TG, \mathcal{GC}) := \{G \in \mathcal{L}(TG) \mid \forall gc \in \mathcal{GC},\ G \models gc\}$ denote the set of all ***schema compliant*** *triple graphs.*

From here on, schema compliance - along with TGG language membership - defines which input and output models can be considered consistent. Definition 6.2 refines Def. 5.2 by stating that a consistent solution must be member of the respective TGG language and comply to a given schema:

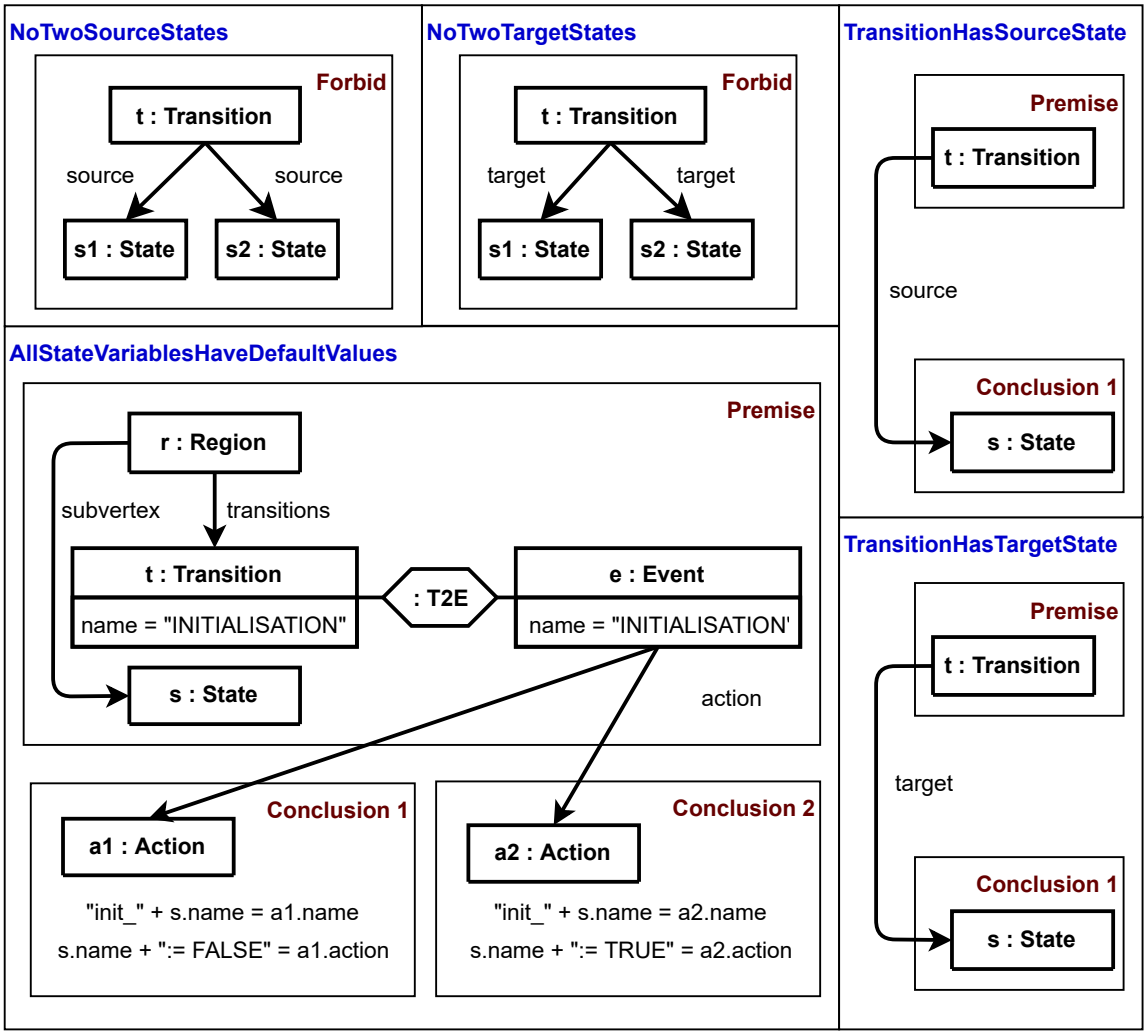


Figure 6.2: Graph constraints for the TGG SysMLToEventB

Definition 6.2 (Consistent Input and Consistent Solution (Refined))**.**
Given a triple graph grammar $TGG = (TG, \mathcal{R})$ *and a schema* $(TG, \mathcal{GC})$*, a starting triple graph* $G_0 = G_S \leftarrow G_C \rightarrow G_T$ *is said to be consistent input iff* $\exists\ G' = G'_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG) \cap \mathcal{L}(TG, \mathcal{GC})$ *according to Def. 5.2.* G' *is referred to as a consistent solution for* G_0 *in each case.*

Based on Def. 6.2, the refined notion of consistency can be integrated into the hybrid framework, such that schema compliance can be guaranteed. First, we start with a short recap of the three constraint types and the objective function as presented in Chap. 5, which are necessary in a setting with graph constraints as well:

- **Exclusions for rules (Def. 5.8):** It must be prohibited that an element is marked more than once, because it would not be possible to create a single element multiple times with declarative rules. For each element that can be marked by multiple rule applications $d_i, \ldots, d_j$, an exclusion constraint $\delta_i + \cdots + \delta_j \leq 1$ is created.
- **Context for rules (Def. 5.9):** There must also be ILP constraints that ensure that the application of a rule depends on the application of all rules that provide context for it. Implication constraints of the form $\delta_i \leq (\delta_{j_1} + \cdots + \delta_{j_m}) \wedge \delta_i \leq \cdots \wedge \delta_i \leq$

$(\delta_{k_1} + \cdots + \delta_{k_n})$ are thus created for all rule applications d_i with required context elements $j, \ldots, k$, and rule applications $(d_{j_1}, \ldots, d_{j_m}, \ldots, d_{k_1}, \ldots d_{k_n})$ that possibly mark these elements.

- **No dependency cycles (Def. 5.11)**: Cyclic dependencies between rule application candidates must be prohibited by excluding at least one candidate from the cycle. For a cycle $d_1, \ldots, d_n$ of length n, a constraint $\sum_{i=1}^{n} \delta_i < n$ guarantees this property.

- **Objective function (Def. 5.12:)** As previously mentioned, the search for a consistent solution is driven by maximising the number of marked elements. If it is possible to mark the input models entirely, they can be completed to a triple that is contained in the TGG's language (and fulfils all graph constraints, which will be shown in the remainder of this section). To form the objective function, a coefficient is computed for each binary variable δ_i that reflects the number of elements that are potentially marked by the rule application d_i. The sum of binary variables δ_i weighted with these coefficients is the goal function to be maximised.

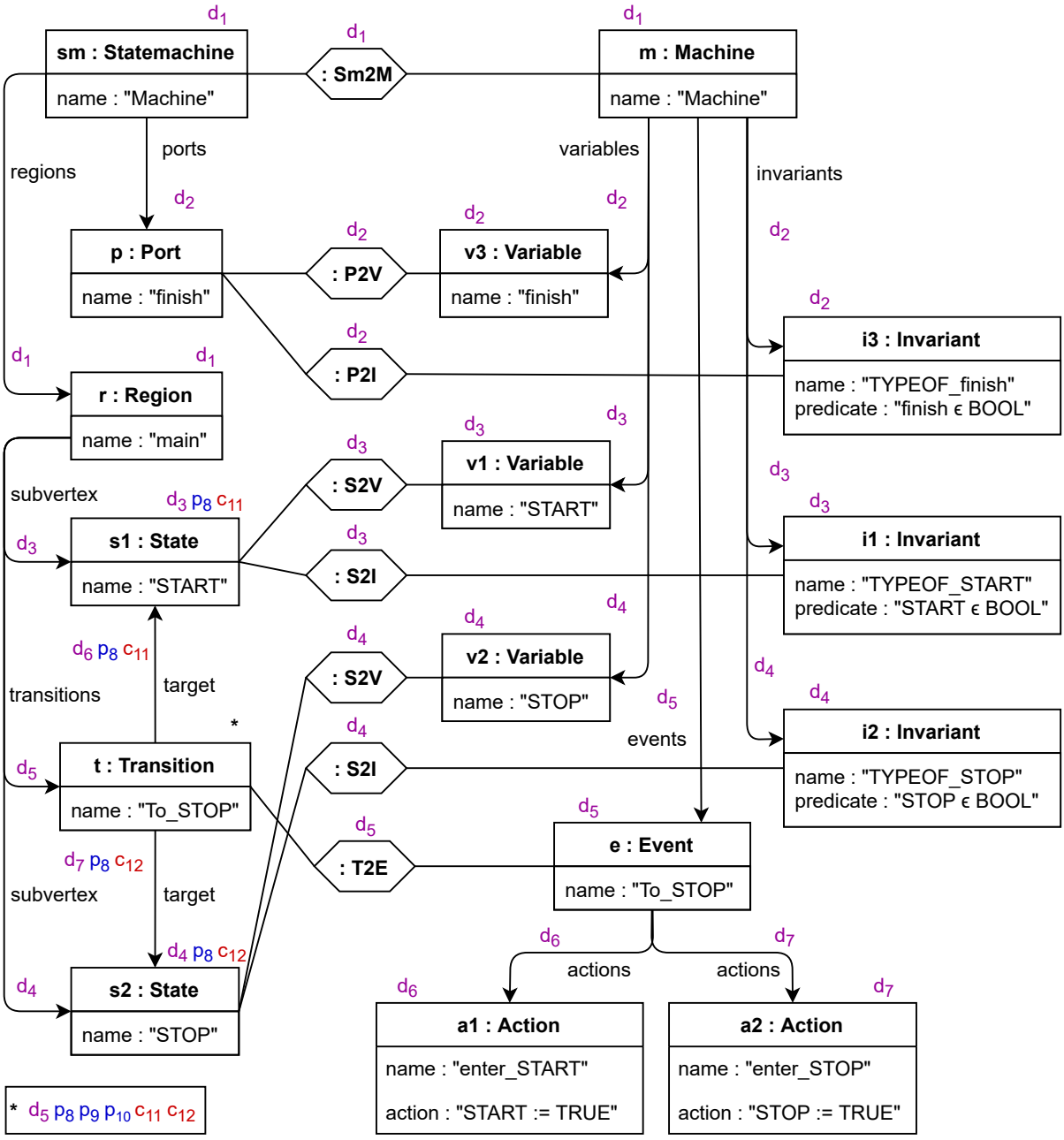
Before encoding the additional constraints of Fig. 6.2 into the optimisation problem, the existing concepts are applied on an example instance for a consistency check with given correspondences (CO) as depicted in Fig. 6.3. In the SysML model, there is a simple statemachine `sm` with one port `p` and one region `r`, which involves two states `s1` and `s2` connected with a transition `t`. According to the model, both states are targets, which should intuitively result in a failed consistency check. In the following, we will see how the hybrid approach can confirm this expectation formally.

To begin with, matches for rule applications are collected for the example instance. In all three models, the nodes and edges are annotated in violet with the rule applications d_i that potentially mark the respective element. In more detail, d_1 represents a rule application candidate for the rule *StatemachineToMachine* (Fig. 3.11), and d_2 a candidate for *PortToVariable* (Fig. 3.10). d_3 and d_4, in contrast, are application candidates of *StateToVariable* (Fig 4.3). The only transition `o`f the SysML model is associated with an event `e` of the Event-B model via d_5, an application candidate of *TransitionToEvent* (Fig. 4.7). Finally, d_6 and d_7 are rule application candidates for *TargetStateToEnterAction* (Fig. 4.9).

When considering the set of potential rule applications, the importance of further constraints becomes apparent: Although the triple is member of the TGG language (applying $\delta_1 \ldots \delta_7$ in ascending order of indices is a valid derivation sequence), the models should be considered incorrect, because one of the states `s1` and `s2` should be the source of `t` instead of the second target. The triple indeed violates the constraints *NoTwoTargetStates* and *TransitionHasSourceState* (cf. Fig. 6.2): The former is violated as the *forbid* pattern can be matched (i. e., the premise of the negative constraint holds), whereas for the latter, there exists a match for the premise, but not for the conclusion.

To encode the graph constraints into the optimisation problem, all possible matches for premises and conclusions are annotated to the involved elements. Negative constraints are hereby represented as graph constraints with a premise but no conclusions, as this is semantics-preserving: As soon as the premise can be matched, the constraint is violated (cf. Def. 4.12).

This leads us to the representation of matches for premise and conclusion patterns in the hybrid approach: Similar to rule applications, matches for graph constraints are also associated to binary variables to ensure that the retrieved solution is schema compliant, i. e., respects all specified constraints. Thereby, matches for premises and conclusions are



Context for rules:

- $\delta_2 \leq \delta_1$
- $\delta_3 \leq \delta_1$
- $\delta_4 \leq \delta_1$
- $\delta_5 \leq \delta_1$
- $\delta_6 \leq \delta_3 \wedge \delta_6 \leq \delta_5$
- $\delta_7 \leq \delta_4 \wedge \delta_7 \leq \delta_5$

Context for premises:

- $\delta_3 + \delta_4 + \delta_5 + \delta_6 + \delta_7 - 5 \leq \pi_8 - 1$
- $\delta_5 \leq \pi_9$
- $\delta_5 \leq \pi_{10}$

Context for conclusions:

- $\gamma_{11} \leq \delta_3 \ \wedge \ \gamma_{11} \leq \delta_5 \ \wedge \ \gamma_{11} \leq \delta_6$
- $\gamma_{12} \leq \delta_4 \ \wedge \ \gamma_{12} \leq \delta_5 \ \wedge \ \gamma_{12} \leq \delta_7$

Implications:

- $\pi_8 \leq 0$
- $\pi_9 \leq 0$
- $\pi_{10} \leq \gamma_{11} + \gamma_{12}$

Objective function: max. $5\delta_1 + 8\delta_2 + 8\delta_3 + 8\delta_4 + 8\delta_5 + 3\delta_6 + 3\delta_7$

Figure 6.3: Inconsistent example instance with annotations for rule applications and constraint matches

separately encoded into constraints as they depend on different sets of rule applications. The matches m_p and m_{c_i} for premises and conclusions are associated with binary variables π and γ (cf. Def. 6.3).

Definition 6.3 (Constraints for Graph Constraints)**.**
Let $\mathcal{GC} = \{(P, \{c_i : P \to C_i \mid i \in I\})\}$ be a set of graph constraints. For each graph constraint $gc \in \mathcal{GC}$, let $\mathcal{P}_{gc} = \{m_p : P \to G\}$ be a set of premises. For each graph constraint $gc \in \mathcal{GC}$ and each premise $m_p \in \mathcal{P}_{gc}$, let $\mathcal{C}_{gc,m_p} = \{m_{c_i} : C_i \to G, i \in I, m_p = c_i \,;\, m_{c_i}\}$ a set of conclusions.

Let $\mathcal{P} = \bigcup_{gc \in \mathcal{GC}} \mathcal{P}_{gc}$ and $\mathcal{C} = \bigcup_{gc \in \mathcal{GC}} \bigcup_{m_p \in \mathcal{P}_{gc}} \mathcal{C}_{gc,m_p}$ be the unions of the respective sets.

For each premise match $m_{p_1}, \ldots, m_{p_m} \in \mathcal{P}$, respective binary variables $\pi_1, \ldots, \pi_m$, and for each conclusion match $m_{c_1}, \ldots, m_{c_n} \in \mathcal{C}$, respective binary variables $\gamma_1, \ldots, \gamma_n$ are defined. A linear constraint $\mathcal{LC}$ for $\mathcal{GC}$ is a conjunction of linear inequalities which involve $\pi_1, \ldots, \pi_m$ and $\gamma_1, \ldots, \gamma_n$. A triple graph G fulfils $\mathcal{LC}$, denoted as $G \models \mathcal{LC}$, iff $\mathcal{LC}$ is satisfied for any variable assignment $\{\pi_1 \ldots \pi_m\} \to \{0,1\}, \{\gamma_1 \ldots \gamma_n\} \to \{0,1\}$.

In the concrete example instance (Fig. 6.3), matches for premises (p_i) and conclusions (c_i) – with non-overlapping index intervals for better readability – are annotated in blue and red, respectively, to the elements which they involve. The five elements annotated with p_8 represent a match for *NoTwoTargetStates*. p_9 refers to the premise of the *TransitionHasSource* constraint, for which no conclusion match can be found. For the premise of *TransitionHasTarget* (p_{10}), in contrast, there are two matches for the conclusions, i. e., c_{11} and c_{12}. In contrast to the binary variables for rule applications (δ_i), the variables π_i for premises and γ_i for conclusions cannot be freely chosen and do not have any influence on the objective function.

Instead, graph constraints pose additional restrictions to the set of valid solutions, which means that they are translated into additional ILP constraints. They encode interdependencies between rule applications and graph constraints, and are formulated in a way that any variable assignment that does not violate them leads to a schema compliant solution. More concretely, these new concepts need to be added to the formal framework for incorporating graph constraints:

- **Context for premises:** Matches for premises depend on rule applications which mark or create the elements that are involved in the graph constraint. In this sense, the premise is fulfilled as soon as all context elements are marked (in the given part of the model) or created (in the remainder of the triple). However, as soon as the context is provided completely, the premise *is* fulfilled. The implication constraint is thus in the opposite direction: Choosing a subset of rule applications $d_i, \ldots, d_j$ that is sufficient to create the context for a premise match p_k implies that p_k is fulfilled.
- **Context for conclusions:** Similar to premise constraints, it must also be reflected in the ILP under which conditions a conclusion of a graph constraint holds. In order to conclude a match for c_k, all involved nodes and edges must be marked or created by at least one rule application each. This subset $d_i, \ldots, d_j$ of rule applications is implied by c_k.
- **Implications for graph constraints:** The semantics of premise and conclusion(s) (cf. Def. 4.8) is reflected in the implications for graph constraints, which define that the presence of a premise match implies the existence of a corresponding conclusion match. Negative constraints are thereby treated as implication constraints with an empty set of conclusions: A solution for the ILP that fulfils this constraint cannot be valid, as the premise holds, but none of the conclusions.

As graph constraints do not mark or create elements, only sets for the elements that are required to be marked or created already are defined, in order to form premise and conclusion, respectively.

Definition 6.4 (Required Elements for Graph Constraints)**.**
For a graph constraint $gc = (P, \{c_i : P \to C_i \mid i \in I\})$ *and morphisms* $m_p : P \to G, (m_{c_i})_{i \in I} : C_i \to G$ *with* $m_p = c_i; m_{c_i}$*, we define:*

- $reqMrk(m_p) = \{e \in mrkElem(G) \mid \exists e' \in elem(P), m_p(e') = e\}$
- $reqCrt(m_p) = \{e \in crtElem(G) \mid \exists e' \in elem(P), m_p(e') = e\}$
- $reqMrk(m_{c_i}) = \{e \in mrkElem(G) \mid \exists e' \in elem(C_i), m_{c_i}(e') = e\}$
- $reqCrt(m_{c_i}) = \{e \in crtElem(G) \mid \exists e' \in elem(C_i), m_{c_i}(e') = e\}$

Based on the required elements for premise and conclusion patterns, a fourth constraint type can be added that guarantees that the rules that mark and create the elements of the pattern are chosen to be applied, such that the pattern match is valid. While rules can provide context for other rules in this way, they also provide context for premise and conclusion patterns, as specified in Def. 6.5.

Definition 6.5 (Constraint 4: Guarantee Context for Graph Constraints)**.**
Given a starting triple graph G_0*, a TGG* $(TG, \mathcal{R})$ *and a schema* $(TG, \mathcal{GC})$*, let* $D : G_0 \overset{*}{\Longrightarrow} G_n$ *be a derivation via operational rules with the underlying set* $\mathcal{D}$ *of direct derivations. For each premise match* $m_p \in \mathcal{P}$ *associated to* π *and each conclusion match* $m_{c_i} \in \mathcal{C}$ *associated to* γ*, the following constraints are defined:*

- $context(m_p) = \sum_{e \in reqMrk(m_p)} [mrkSum(e) - 1] + \sum_{d \in \mathcal{D}, [reqCrt(m_p) \cap crt(d) \neq \emptyset]} [\delta - 1] \leq \pi - 1$
- $context(m_{c_i}) = \bigwedge_{e \in reqMrk(m_{c_i})} [\gamma \leq mrkSum(e)] \wedge \bigwedge_{d \in \mathcal{D}, [reqCrt(m_{c_i}) \cap crt(d) \neq \emptyset]} [\gamma \leq \delta]$
- $context(G) = \bigwedge_{m_p \in \mathcal{P}} context(m_p) \wedge \bigwedge_{m_{c_i} \in \mathcal{C}} context(m_{c_i})$

context(m_p) is an implication of the form "m_p matches" $\implies \pi$. The two summands in the left part of the inequality are both 0 exactly when all required marked and created elements for m_p are present and are both negative otherwise. Demanding their sum to be $\leq \pi - 1$ forces the solver to set π to 1 whenever m_p matches. context(m_{c_i}) is analogous to context(d) (cf. Def. 5.9), i. e., the solver is not allowed to set any γ to 1 unless m_{c_i} matches. The reason for the structural differences between the formulae for context(m_p) and context(m_{c_i}) is the optimisation goal of marking as many elements as possible: Together with the encoding for satisfying graph constraints (cf. Def. 6.6), it can be guaranteed that a feasible solution of the ILP results in a triple that satisfies all given constraints. context(G), finally, just ranges over all matches for premises and conclusions.

For the running example, the context constraints for premise and conclusion patterns are listed in the middle column on the bottom of Fig. 6.3 ("context for premises", "context for conclusions"). In order to have a match for the negative constraint *NoTwoTargetStates* (p_8), each of the rule applications $d_3, \ldots, d_7$ must be chosen to have a complete match. For the implication constraints *TransitionHasSource* (p_9) and *TransitionHasTarget* (p_{10}), the choice of d_5 (rule TransitionToEvent) is sufficient for one premise match each. For

the latter, two matches for the conclusion (c_{10}, c_{11}) can be found as soon as the rules *StateToVariable* (d_3, d_4) and *TargetStateToEnterAction* (d_3, d_7) are applied.

The interdependencies between premises and conclusions are expressed by further type of linear constraints. This last constraint type encodes the semantics of premise and conclusions of graph constraints (cf. Def. 4.12), such that schema compliance can be guaranteed for the transformation result. According to constraint 4 (Def. 6.5) the solver has no reason to set any γ to 1; this will only be enforced with constraint 5 (Def. 6.6) that covers the relation between m_ps and m_{c_i}s. As the binary variables π (premise) and γ (conclusion) are set to 1 if the respective context is created or marked entirely, the potential matches can be considered as actual matches. The constraint can be formulated independent of the concrete rule applications, as their values influence the value assignment to all variables π and γ.

With the last constraint type, the additional variables for constraint pattern matches can be connected, such that the propositional logic of graph constraints is reflected in the set of ILP constraints (Def. 6.6). Linear constraints of this type are violated as soon as the variable associated to the premise match (π) is set to 1, but no match for a conclusion is found, for which the respective binary variable can be set to 1 as well, or the required context is not entirely created by the chosen rule applications (cf. Def. 6.5). For negative constraints, there are no conclusions, such that the respective linear constraint ($\pi \leq 0$) is immediately violated if π is set to 1. To satisfy $sat(G)$ as a whole, the previously described conditions must be met for all matches for premises that can be found for each graph constraint of the schema.

Definition 6.6 (Constraint 5: Satisfy Graph Constraints)**.**
Let $(TG, \mathcal{GC} = \{(P, \{c_i : P \to C_i \mid i \in I\})\})$ be a schema. Let $m_p \in \mathcal{P}$ denote a premise match associated to π, and let $m_{c_i} \in \mathcal{C}$ denote a conclusion match associated to γ. A linear constraint $sat(G)$ expressing that G fulfils all graph constraints of $\mathcal{GC}$ is defined as follows:

$$sat(G) = \bigwedge_{gc \in \mathcal{GC}} \bigwedge_{m_p \in \mathcal{P}_{gc}} [\pi \leq \sum_{m_{c_i} \in \mathcal{C}_{gc,m_p}, i \in I} \gamma]$$

The advantage of searching for potential matches for premise and conclusion patterns is that the decision whether rules are applicable or not, i. e., lead to constraint violations, is not made during the pattern matching step, but as part of the optimisation step. In cases where different rule applications can cause a constraint violation, it is possible to leave out the least important one from a global point of view, instead of blocking the one that is found last during pattern matching.

For the running example, the respective constraints are shown on the bottom right of Fig. 6.3 ("implications for graph constraints"). p_8 as a match for the negative constraint *NoTwoTargetStates* does not have a conclusion, such that the constraint is violated as soon as π_8 is set to 1. The same holds for p_9: Although this is a match for the premise of the implication constraint *TransitionHasSourceState*, it is immediately violated when π_9 is set to 1 because no match for a conclusion can be found, as no source state exists. To satisfy the third linear constraint corresponding to *TransitionHasTargetState*, it is sufficient to either set π_{10} to 0, or set either γ_{11} or γ_{12} to 1.

Finally, we refine the definition of the optimisation problem (Def. 5.12), such that the new constraint types are part of the ILP as well. The optimisation goal is still the same: A solution that entirely marks the input (and can therefore be completed to a schema compliant triple contained in the TGG's language) is preferred over a solution with less markings. As the solution must fulfil all ILP constraints, language membership and schema compli-

ance according to Def. 6.1 can be guaranteed in case of both consistent and inconsistent input models.

Definition 6.7 (Optimisation Problem (Refined))**.**
Given a starting triple graph G_0, a TGG (TG, $\mathcal{R}$) and a schema ($TG, \mathcal{GC}$), let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules. The ILP to be optimised is constructed as follows:

$$max. \sum_{d \in D} |mrk(d)| \cdot \delta \ s.t.$$
$$markedAtMostOnce(G_0) \wedge context(D) \wedge context(G_n) \wedge acyclic(D) \wedge sat(G_n)$$

The objective function to maximize the number of markings is depicted on the bottom line of Fig. 6.3)) All input model elements in the example instance could be marked setting $\delta_1, \ldots, \delta_7$ to 1, leading to an objective function value of 43 equal to the total number of elements in the triple graph. This marking would however violate the constraints *NoTwoTargetStates* and *TransitionHasSourceState* in the source model. We will now analyse how this graph constraint violation becomes apparent in the ILP: As d_5 was chosen, it creates the transition `t`, on which the premises p_9 and p_{10} of the constraints *TransitionHasSourceState* and *TransitionHasSourceState* can be matched, and as a result, π_9 and π_{10} must be set to 1 as well. While this is possible for π_{10} (γ_{11} and γ_{12} can be set to 1 in accordance with the context constraints for conclusions), the constraint $\pi_9 \leq 0$ is immediately violated. Similarly, the first context constraint for premises enforces that π_8 (*NoTwoTargetStates*) is set to 1, which leads to a further constraint violation for $\pi_8 \leq 0$. The optimal solution, representing the maximal consistent sub-triple, is achieved by setting δ_5, δ_6 and δ_7 to 0, such that π_8, π_9 and π_{10} can be set to 0 as well, resulting in an objective function value of 29. This means that leaving out `t`, the corresponding event `e` with its two actions `a1` and `a2` and all dependent arrows would lead to a consistent triple.

6.5 Correctness and Completeness

Based on the formal framework presented up to Sect. 6.4, we ar now able to show *correctness* and *completeness* properties for the hybrid approach, i. e., that the four previously introduced consistency management operations terminate with a consistent result if and only if it exists. We show that the ILP-based solution strategy - posing some assumptions on the TGG in use - always terminates, and yields a consistent solution with respect to Def. 6.2 iff such a solution exists. The formal proof follows the structures of Leblebici's proof for correctness and completeness of the correspondence creation operation (CC) in a setting without graph constraints [Leb18]. The main challenge here is to show that (i) the arguments from [Leb18] are transferable from CC to all other three operations and that (ii) schema compliance can still be guaranteed when further elements are continuously added during match collection. In the following, let $TGG = (G_\emptyset, \mathcal{R})$ and a schema $(TG, \mathcal{GC})$ be given for all definitions, lemmas and theorems.

As previously stated, the goal of the optimisation step is to determine a subset of rule applications that forms a derivation sequence from the starting triple graph to a transformation result. The input for the optimisation step is a "super model" G_n, which was constructed by applying all direct derivations of a set $\mathcal{D}$. We define a *proper subset* $\mathcal{D}$' of operational rule applications that can be arranged, such that a (possibly incomplete) derivation sequence D' is formed. The values assigned to the associated binary variables δ for each $d \in \mathcal{D}$ (1 for all $d \in \mathcal{D}'$, 0 for all $d \in \mathcal{D} \setminus \mathcal{D}$') form a feasible solution for the ILP, i. e., satisfy all defined constraints (Def. 5.8, 5.9, 5.11, 6.5 and 6.6).

Definition 6.8 (Proper Subset of Rule Applications)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$, and let $\mathcal{D}' \subseteq \mathcal{D}$ be a subset of direct derivations, such that $D' : G_0 \overset{*}{\Longrightarrow} G'$. We refer to $\mathcal{D}'$ as a proper subset of $\mathcal{D}$ iff $\mathcal{D}' \models markedAtMostOnce(G_0) \wedge context(D') \wedge acyclic(D') \wedge context(G') \wedge sat(G')$.

The first lemma (Lem. 6.1) states that such a proper subset exists if and only if there is a triple graph G' that is contained in the TGG's language, fulfils all constraints and does not have more elements than the starting triple graph in the given parts of the triple.

Lemma 6.1 (Consistent Portions of a Triple Graph)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$. $\exists$ proper subset $\mathcal{D}' \subseteq \mathcal{D}$ with $D' : G_0 \overset{*}{\Longrightarrow} G' \iff \exists\, G' \in \mathcal{L}(TGG) \cap \mathcal{L}(TG, \mathcal{GC})$ such that:

1. *$mrkElem(G') \subseteq mrkElem(G_0)$*
2. *$mrkElem(G') = \bigcup_{d' \in \mathcal{D}'} mrk(d')$,*
3. *$crtElem(G') = \bigcup_{d' \in \mathcal{D}'} crt(d')$.*

Proof (Sketch). First, it is shown that $\mathcal{D}$' is a proper subset iff $G' \in \mathcal{L}(TGG) \cap \mathcal{L}(TG, \mathcal{GC})$ holds. Our argumentation is based on the five constraint types that are fulfilled by G' and the derivation sequence D' according to Def. 6.8; With the first three constraint types, language membership can be ensured, whereas the last two constraint types guarantee schema compliance.

- *markedAtMostOnce(G_0)* ensures that elements of the input models are never marked twice, which would contradict the intention of simulating declarative rule applications with the hybrid approach. With declarative rules, each element can only be created once, which must be reflected by the marking strategy as well.
- *context(D')* guarantees that for the created part of the triple, all required context elements already exist, and for the marked part of the triple, the respective elements have been marked already. Hereby, we ensure that an operational rule is applicable iff the respective declarative rule is applicable in a comparable setting.
- With the constraint *acyclic(D')*, cyclic dependencies of rule applications can be excluded, such that the subset $\mathcal{D}$' can be sequenced over the $\triangleright$ relation.
- *context(G')* ensures that matches for premises and conclusions of graph constraints are found iff all elements which are part of the match are either created or marked by a chosen rule application.
- Finally, *sat(G')* represents the logic of the supported graph constraints, i. e., formalises the interrelations between premises and conclusions. With this constraint, it is guaranteed that G' fulfils all specified constraints.

Based on this equivalence, it remains to show that the three properties hold for a proper subset $\mathcal{D}$' and a resulting triple graph G':

1. As no new elements are added to the marked part of the input triple during the construction of $\mathcal{D}$, and only those elements can be marked in G' which are already present in G_0, mrkElem(G') $\subseteq$ mrkElem(G_0) holds.

2. The markings in G' result from the application of all operational rules $d' \in \mathcal{D}'$, such that this condition is fulfilled as well.

3. Likewise, the created part of G' is built up by applying all operational rules $d' \in \mathcal{D}'$.

□

The sequential application of rules of a proper subset leads to a transformation result, whose marked and created elements form a triple that is member of the TGG's language and that is schema compliant. In general, however, elements of the input can remain unmarked, which means that they were not (yet) consistently transformed. This does not necessarily mean that the input models are inconsistent, as adding further rule applications could form a proper subset that entirely marks the input models. Therefore, we denote proper subsets which involve a maximal number of markings as *maximal proper subsets*. As a proper subset fulfils all ILP constraints by definition, and maximising the number of marked elements is the optimisation goal, a maximal proper subset will be returned as a solution for the ILP.

Definition 6.9 (Maximal Proper Subset of Rule Applications)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$. A proper subset $\mathcal{D}'$ of $\mathcal{D}$ is maximal if there does not exist any other proper subset $\mathcal{D}''$ of $\mathcal{D}$ with a greater objective function value (cf. Def. 6.7).

Accordingly, we denote the triple graph that results from sequentially applying rules of the maximal proper subset on the starting triple graph as *maximally marked*.

Definition 6.10 (Maximally Marked Triple Graph)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$. Let $\mathcal{D}'$ be a maximal, proper subset of $\mathcal{D}$. The triple graph G' identified with $\mathcal{D}'$ according to Lemma 6.1 is denoted as a maximally marked triple graph with respect to D.

In Thm. 6.1, the correctness of the ILP-based transformation can be shown using the notion of a maximally marked triple graph, i. e., if it is possible to mark each element of the starting triple graph G_0 exactly once, the resulting maximally marked triple graph is a consistent solution according to Def. 6.2. Otherwise, the largest consistent sub-triple – according to the number of marked elements – is determined in the optimisation step, because the goal function maximises this number while respecting the previously described constraint types.

Theorem 6.1 (Correctness)**.**
Given a starting triple graph G_0 and a derivation $D : G_0 \overset{*}{\Longrightarrow} G_n$ with an underlying set $\mathcal{D}$ of direct derivations, let $\mathcal{D}' \subseteq \mathcal{D}$ be a maximal proper subset, such that $D' : G_0 \overset{*}{\Longrightarrow} G'$. It holds for a maximally marked triple graph G' with respect to $\mathcal{D}$:

$$\bigcup_{d' \in \mathcal{D}'} mrk(d') = mrkElem(G') = mrkElem(G_0) \implies G' \text{ is a consistent solution.}$$

Proof (Sketch). For G' being a consistent solution, (1) $G' \in \mathcal{L}(TGG)$ und (2) $G' \in \mathcal{L}(TG, \mathcal{GC})$ are required. The premise of the theorem, i. e., the marked part of G' and the non-empty part of G_0 are identical, states that it is possible to entirely mark G_0 by

sequentially applying all direct derivations $d' \in \mathcal{D}'$, whereby $\mathcal{D}'$ is a maximal, proper subset. By applying Lemma 6.1 with a maximal, proper subset, a triple graph G' can be produced that fulfils both conditions. Together with the decomposition and composition theorem for TGGs and operational rules [EEE+07], G' is a consistent solution according to Def. 6.2, while G_0 is a consistent input. □

For showing completeness, it remains to show the opposite direction, i.e., that the proposed strategy finds a consistent solution if one exists. The main challenge is to guarantee that the set $\mathcal{D}$ of rule application candidates is finite, such that the process always terminates, because ILP solving is known to be correct and complete as well. Therefore, we have to demand that the underlying TGG be *progressive*, which means that each operational rule has to mark at least one element. It is possible that some TGGs are progressive for only a few operations. In the running example, progressiveness is not fulfilled for the backward transformation operation, as the rule *AddRegion* only operates on the source model (cf. Fig. 4.2). For practical implementations, the problem can be overcome by applying heuristics, such as fixing an upper bound for applications of such rules.

Definition 6.11 (Progressive TGGs)**.**
A TGG is progressive for an operation op $\in \{$*CC, CO, FWD_OPT, BWD_OPT*$\}$ *if each of its operational rules for op has at least one marking element.*

Although progressiveness of a TGG ensures that the number of markings strictly increases when following a derivation sequence, it is still possible that new rule applications create new context elements which themselves enable further rule applications, such that termination cannot be immediately guaranteed as elements of the starting triple graph can be marked infinitely often (although only one of these markings can be finally chosen). However, as soon as rule applications overlap in their markings and one of them depends on the created context of another, the dependent rule application is superfluous as it implies and excludes the application of another rule at the same time. Therefore, only *essential* rule applications (cf. Def. 6.12) should be considered for forming a derivation sequence, meaning that (1) identical rule applications and (2) rule applications that are in conflict with their create dependencies as described above must be discarded. Note that in contrast to dependencies via marked context ($\triangleright^M$), create dependencies via $\triangleright^C$ are *actual* and not potential dependencies, as the rule application that creates an element is unique.

Definition 6.12 (Essential and Superfluous Rule Applications)**.**
Given a starting triple graph G_0 and a derivation $D : G_0 \overset{*}{\Longrightarrow} G_n$ with underlying set of direct derivations $\mathcal{D}$. Let $\triangleright^C_* \subseteq \mathcal{D} \times \mathcal{D}$ be the transitive closure of the $\triangleright^C$ relation in Def. 5.10. A rule application $d_{n+1} : G_n \overset{cr_{n+1}@cm_{n+1}}{\Longrightarrow} G_{n+1}$ with operational rule or_{n+1} is essential for D if:

1. $\nexists d_i \in \mathcal{D}, d_i : G_{i-1} \overset{or_i@om_i}{\Longrightarrow} G_i$ *such that* $or_{n+1} = or_i$ *and* $om_{n+1} = om_i$ *and*
2. $mrk(d_{n+1}) \cap \bigcup_{d' \in \mathcal{D}, d_{n+1} \triangleright^C_* d'} mrk(d') = \emptyset.$

Otherwise, d_{n+1} is superfluous for D.

Introducing some terminology for derivation sequences that purely consist of essential rule applications, such derivations are denoted as *final* according to Def. 6.13.

Definition 6.13 (Final Derivations with Operational Rules)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules with underlying set of direct derivations $\mathcal{D}$.
D is final if $\nexists d_{n+1} : d_{n+1} : G_n \overset{cr_{n+1}@cm_{n+1}}{\Longrightarrow} G_{n+1}$ such that d_{n+1} is essential for D.

In Lemma 6.2, we show that for every operation $op \in \{\text{CC, CO, FWD\_OPT, BWD\_OPT}\}$ and every starting triple graph G_0, there exists a final derivation, assuming that the underlying TGG is progressive, such that the process of gathering rule application candidates terminates.

Lemma 6.2 (Existence of a Final Derivation)**.**
Given a progressive TGG for an operation $op \in \{CC, CO, FWD\_OPT, BWD\_OPT\}$ and a starting triple graph G_0, a final derivation $D : G_0 \overset{*}{\Longrightarrow} G_n$ with operational rules for op exists for every starting triple graph G_0 for op.

Proof (Sketch). We show the set of essential rule applications according to Def. 6.12 is finite using two arguments: First, the number of possible derivation sequences of some fixed length l is finite for each $l \in \mathbb{N}$. Second, the length of a single derivation sequence consisting only of essential rule applications is also finite.

The first statement can be proven via induction over the length l of the derivation sequence. As all rule applications are essential, they can be sequenced over the $\triangleright^C$ relation by using Condition (2) in Def. 6.12, forming a derivation $d_i \triangleright^C \cdots \triangleright^C d_j$. For the induction step, let $d_i \triangleright^C \cdots \triangleright^C d_j$ be a derivation sequence of length l and let d_k be the next derivation to be added. d_k can only require context elements that are created by any rule application in $d_i \dots d_j$. According to the induction hypothesis, the number of such possible sequences is finite, so is the number of created context elements, as each rule application can create only a finite number of elements. As only distinct matches are considered (Condition (1) in Def. 6.12), the number of possible sequences $d_i \triangleright^C \cdots \triangleright^C d_j \triangleright^C d_k$ of length $l + 1$ must be finite.

To show that the second statement holds, we derive a contradiction from the assumption that derivation sequence of infinite length consisting solely of essential rule applications can be constructed for a (finite) starting triple graph. The set of essential rule applications can be partitioned into those with and without create dependencies; we will show that both sets are finite. For the former, it can be stated that both the starting triple graph G_0 and the set R of TGG rules are finite. Along with Condition (1) of Def. 6.12 (uniqueness of rule applications), it follows that the set of essential rule applications without create dependencies is finite as well. The TGG is required to be progressive (Def. 6.11), therefore each rule application must mark at least one element. Marking elements is only possible in the non-empty parts of G_0, to which no new elements can be added by applying operational rules. Each element of the starting triple graph G_0 can be marked infinitely often in theory, but Condition (2) in Def. 6.12 prevents an essential rule application d from marking elements that are also marked by a rule application d' if a context dependency $d \triangleright^C_* d'$ exists, stepwise and strictly reducing the set of elements that can be marked by d. As the total number of markable elements in the starting triple graph G_0 is finite, the constructed derivation sequence of essential rule applications must be of finite length.

Combining the arguments for the finiteness of the length of the derivation sequence on the one hand and of the number of sequences of a fixed length on the other hand, the number of essential rule applications must be finite as well, contradicting the assumption and proving the second statement. □

Lemma 6.2 can now be used to show completeness, i.e., that the existence of a consistent solution also implies that there is a final derivation of operational rules from the starting triple graph to this solution.

Theorem 6.2 (Completeness)**.**
Given a progressive TGG for an operation op $\in \{CC, CO, FWD\_OPT, BWD\_OPT\}$*, and a starting triple graph* G_0*, a final derivation* $D : G_0 \overset{*}{\Longrightarrow} G_n$ *for op, a maximal proper subset* $\mathcal{D}' \subseteq \mathcal{D}$ *and a maximally marked triple graph* G' *with respect to* D *exist such that:*

$$G' \text{ is a consistent solution} \iff \bigcup_{d' \in \mathcal{D}'} mrk(d') = elem(G_0)$$

Proof (Sketch). The implication in backward direction of the equivalence follows from Thm. 6.1, such that only the implication in forward direction is to be shown.

To derive a contradiction, we assume that G' is a consistent solution for the given input triple G_0, but G' contains unmarked elements. As G' is consistent, $G' \in \mathcal{L}(TGG)$ and $G' \in \mathcal{L}(TG, \mathcal{GC})$ hold. From $G' \in \mathcal{L}(TGG)$ and the decomposition and composition theorem for TGGs and operational rules [EEE+07, Leb18], it follows that there exists a derivation sequence $D' : G_0 \overset{*}{\Longrightarrow} G'$ via operational rules that entirely marks G_0. From Lemma 6.2, it follows that a final derivation D'' exists. As D' and D'' are not equal, and D'' is the final derivation for which the number of markings is maximised according to the optimisation objective (Def. 6.7), there must be at least one superfluous rule application in D'. This contradicts the assumption $G' \in \mathcal{L}(TGG)$ because superfluous rule applications lead to multiple markings on at least one element, i.e., G' cannot be produced using the respective set of declarative rule applications corresponding to D'. □

In summary, it can be stated that correctness and completeness of the hybrid approach can be guaranteed under a few assumptions: The TGG at hand must be progressive – a property that also depends on the concrete consistency management operation – and it must be ensured that only essential rule applications are collected during the pattern matching process. Especially for FWD_OPT and BWD_OPT, this is a practically relevant problem, because rules that do not mark any elements cannot be simply left out, as they possibly create context required by other rule applications. In order to still guarantee termination for practical implementations, a sufficiently large upper limit can be set for the number of rule applications. Better solution strategies might be possible, but are left to future work at this point. All main arguments from the formal proof for consistency checks [Leb18] could be generalised for the other operations by referring to the marked and created parts of the graph instead of the source, target and correspondence models. As no new elements are generated during the search for pattern matches for graph constraints, their integration into the formal framework does not require further restrictions or assumptions regarding the TGG at hand.

6.6 Evaluation

In an experimental evaluation, we analyse the impact of graph constraints on runtime performance using two example TGGs. The *JavatoDoc* TGG, compared to its original version [FKM+20, WFA20], was enriched with graph constraints. Furthermore, the BX benchmark example *FamiliesToPersons* was used, extended by additional negative and implication constraints. An overview of the metamodels, the TGG rules and constraints can be found in the appendix (Chap. A).

We investigate the scalability of the respective operations for growing model sizes with and without taking graph constraints into account with the following research questions:

RQ1 By which factor does the number of variables increase for the different operations when introducing graph constraints to the ILP?

RQ2 How does the runtime performance relate to the model size (number of nodes and edges) for consistency management operations with and without graph constraints?

RQ3 How is the runtime distributed between the different steps, i.e., pattern matching, rule application, ILP construction, and ILP solving?

RQ4 Do the operations show different scalability characteristics with and without graph constraints?

Setup: As inputs for the considered operations, synthetic test models were created with the model generation operation of eMoflon::Neo, which applies a given number of rules randomly, starting with an empty triple. The generated triples had overall model sizes from 720 to 36000 elements, i.e., nodes and edges. To create models of realistic shape, rule applications were composed with a fixed ratio as shown in Tab. 6.1 and 6.2. To keep the ratio equal for all model sizes, the number of rule applications was increased with a scaling factor. A factor of 10, for instance, was needed for triples with 720 elements, and a factor of 500 for 36000 elements.

JavaToDoc	# new elements	# rule applications	# created elements
ClazzToDoc	3	1	3
SubClazzToDoc	5	1	5
MethodToEntry	5	4	20
AddParameter	3	4	12
FieldToEntry	5	4	20
AddGlossary	1	0	0
LinkGlossaryEntry	1	4	4
AddGlossaryEntry	2	4	8

Table 6.1: Quota for generating JavaToDoc instances

FamiliesToPersons	# new elements	# rule applications	# created elements
FamiliesToPersons	3	1	3
MotherToFemale	7	1	7
- *w/o new Family*	5	2	10
FatherToMale	7	1	7
- *w/o new Family*	5	2	10
DaughterToFemale	7	1	7
- *w/o new Family*	5	2	10
SonToMale	7	1	7
- *w/o new Family*	5	2	10

Table 6.2: Quota for generating FamiliesToPersons instances

As a result, the models are guaranteed to be contained in the TGG, but likely violate each of the posed constraints several times. This is desirable, as the impact of taking graph constraints into account on the runtime performance is to be analysed, also for instances with many constraint violations at different points.

Each operation was run on these models (1) without graph constraints, (2) only with negative constraints, and (3) both with negative and implication constraints. Due to the nature of the operations, the entire triple was only given as input to the CO operation; the other operations received the respective parts of the triple. The JavaToDoc TGG is not progressive (cf. Def. 6.11) for FWD_OPT, such that the rules *AddGlossary*, *AddGlossaryEntry* and *LinkGlossaryEntry* were omitted for this operation. Similarly, the rule *AddParameter* was omitted for BWD_OPT. For the *FamiliesToPersons* TGG, no adaptions were necessary.

To reduce the effect of outliers, each configuration was repeated 5 times, and the median was taken as runtime result. Furthermore, the number of binary variables in the ILP

was tracked to get an indication for the problem size. The executions took place on a standard notebook with an Intel Core i7 (1.80 GHz), 16GB RAM, and Windows 10 64-bit as operating system. As a prerequisite for eMoflon::Neo, an installation of Eclipse IDE for Java and DSL Developers, version 2021-03 (4.19.0) with Java Development Kit (JDK) version 13 was used. 4GB RAM were allocated to the JVM running the tests, while 8GB were allocated to the graph database Neo4j (version 3.5.8). Gurobi 8.1.1 was used to solve the ILP.

Results: In the following, an overview of the runtime measurements is provided. While the results have already been summarised in previous work [WA21b], the complete dataset is available online[1].

The overall runtime needed for performing the operations on generated models of different sizes is depicted in Fig. 6.4, 6.6, 6.8, and 6.10, respectively. Note that both axes have a logarithmic scale, making it possible to depict the measurements for small and large model sizes in one diagram. It can be observed that the consumed runtime depends much more on the particular operation than on whether negative constraints are taken into consideration. For all operations, a slightly super-linear increase of runtime can be observed, independent of the consideration of negative constraints.

When executing CO and BWD_OPT for the *JavaToDoc* example, however, a substantial difference can be observed when taking implication constraints into account as well. An explanation could be that the CO operation is quite cheap in general as all relevant rule application candidates can be collected in parallel in a single step, as no new (context) elements are generated during the operation.

However, when the operation must handle implication constraints, the constructed ILP gets more complex, which can be observed when considering the increased share of the ILP solving step (Fig. B.6). As `Entry` nodes of the documentation model can be transformed to either `Method` or `Field` nodes in the Java model, various possibilities exist for BWD_OPT to transform the target model. This could result in an exploding number of premise matches for the *SameNameSameGlossaryEntry* constraint (cf. Fig. 6.2), leading to a large runtime consumption of the operation.

The number of binary variables moderately correlates with the runtime consumption for both TGGs all operations (cf. Fig. 6.5, 6.7, 6.9, and 6.11), reflecting the additional time effort for pattern matching and ILP solving. Larger differences can only be observed for the *FamiliesToPersons* example for the operations CC and FWD_OPT with implication constraints, where the increase in the number of binary variables does not affect the runtime consumption to the same extent.

Figures B.1 - B.24, which were moved to the appendix for better readability, depict the runtime consumptions of the different phases, namely pattern matching, rule application, ILP construction and ILP solving, for both example TGGs and each of the four consistency management operations. Gurobi's logger messages indicated that it was possible for the solver to substantially reduce the optimisation problem at an early stage, such that pattern matching remained the most costly step for CC (cf. Fig. B.11) and still played an important role for the overall runtime performance of FWD_OPT (cf. Fig. B.17).

For all operations, ILP solving has a major impact on the runtime performance when handling implication constraints, whereas pattern matching appears to be the most time-consuming step otherwise. Applying the rule candidates to the model triple is expensive for all operations except CO, as for this operation, all parts of the triple are given as input and elements need not be created. The time needed for ILP construction is almost negligible, while showing similar scalability characteristics as the solving step.

[1] https://bit.ly/35RAeaz, https://bit.ly/3u5oXgs

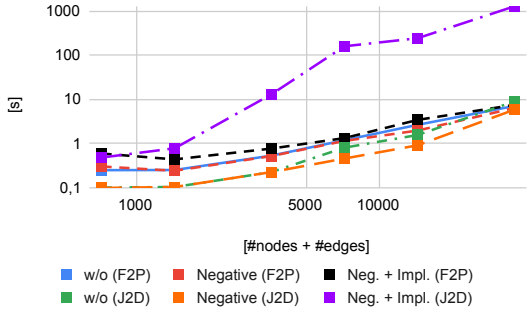


Figure 6.4: Runtime: CO

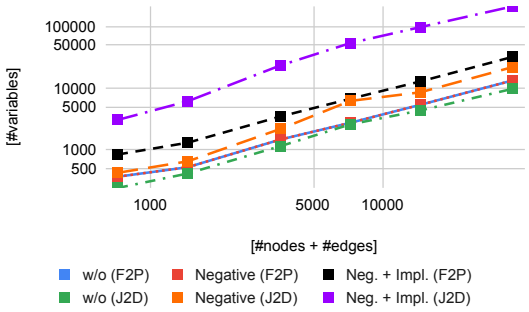


Figure 6.5: Number of variables: CO

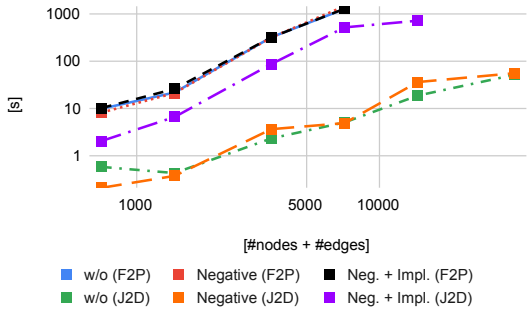


Figure 6.6: Runtime: CC

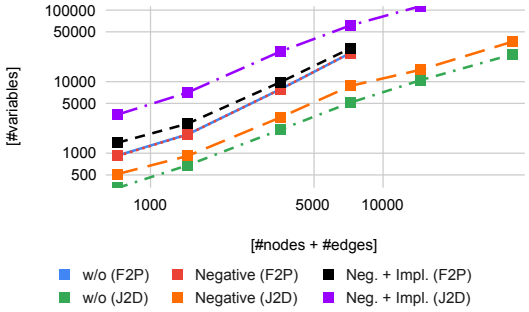


Figure 6.7: Number of variables: CC

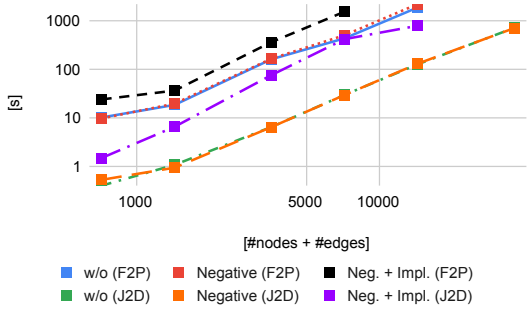


Figure 6.8: Runtime: FWD_OPT

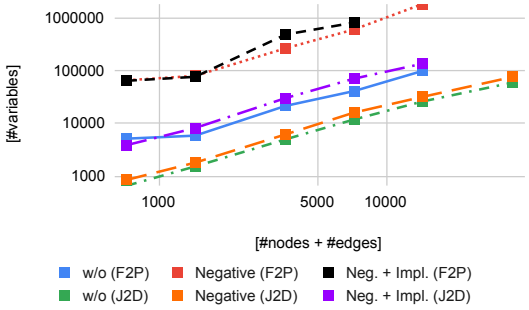


Figure 6.9: Number of var.: FWD_OPT

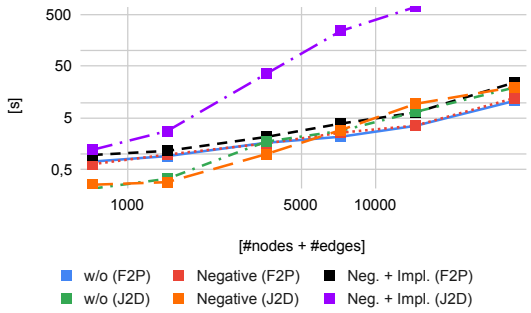


Figure 6.10: Runtime: BWD_OPT

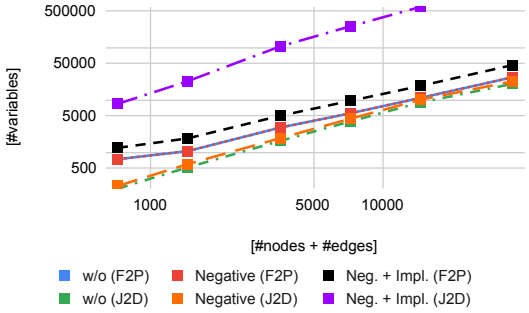


Figure 6.11: Number of var.: BWD_OPT

In total, for negative graph constraints, both the number of variables and the runtime consumption increase by a small factor that remains roughly the same for all model sizes This can be explained with the additional variables as described in Def. 6.3. For implication constraints, both measures substantially increase for most operations and both examples. The root cause is probably the additional complexity which is induced by handling premise and conclusion patterns separately, adding more complexity to the constructed ILP problem.

Summary: Revisiting our research questions, the number of additional binary variables increases moderately for all consistency management operations when adding negative constraints, whereas implication constraints can have a large impact on this measure (RQ1). The time required for each of the steps increases slightly super-linear, whereby a noticeably larger increase was observable for CO and BWD_OPT for the *JavaToDoc* example after adding implication constraints. Depending on the TGG and operation, this can become problematic for large model sizes (RQ2). The runtime consumption is dominated by the pattern matching and rule application steps in settings without implication constraints, whereas ILP solving is of major importance otherwise. Efforts for the ILP construction step can almost be neglected, though (RQ3). The runtime performance is much more dependent on the concrete operation than on whether negative constraints are considered, which might also differ depending on the characteristics of the TGG in use. Implication constraints, however, add more complexity to the ILP, whereby the extent again differs between the operations and TGG examples at hand. (RQ4).

Threats to validity: The runtime measurements were conducted only for two TGGs, with eight and nine comparably small rules. Also the used graph constraints are composed of rather small patterns. The example models were small- or medium-sized and generated randomly, such that they are not necessarily comparable to realistic use cases. The observed remarkably different runtime measurements for the two TGGs and the four operations indicate that the performance depends on the used TGG, making further tests with other benchmark examples crucial. Our results only hold for Neo4j as graph pattern matcher, and Gurobi as ILP solver. Previous experiments [WA20] have indicated, however, that these two solvers are very well-suited for their respective tasks.

6.7 Summary and Discussion

In this chapter, the handling of graph constraints was integrated into the hybrid consistency management framework, which enables the user to express domain constraints, such as upper and lower bounds for multiplicities in UML class diagrams. As there is a bidirectional transformation algorithm between graph constraints and application conditions, the extension substantially increases the expressive power of the approach, as discussed in Sect. 4.5. For different consistency management operations, graph constraints are encoded as additional linear constraints which are added to the ILP in order to guarantee schema compliance, in addition to consistency with respect to the underlying TGG.

Following the structure of the correctness and completeness proof for consistency checks, we have shown that for all four operations, the hybrid approach terminates with a consistent solution, if and only if one exists, and returns the largest consistent sub-triple otherwise. This holds both for settings with and without graph constraints.

Regarding the advanced language features presented in Chap. 4, the approach now covers attribute conditions, as they remain directly connected to TGG rules, and graph constraints as a substitute for application conditions. The integration of multi-amalgamated rules is left to future work, leaving room for increasing the expressive power even further.

An interesting conceptual feature would be fault-tolerance towards constraint violations in the output models: For practical use, it might be more helpful to provide the user with a solution that violates a single constraint than to leave larger parts of the input models untranslated, besides returning the largest (fully) consistent sub-triple. In this case, however, the question would arise whether one should also permit inconsistent results with respect to the TGG, which makes it possible to return a (schema compliant) triple that is not fully inter-model consistent.

The approach was implemented in the TGG tool eMoflon::Neo for all operations. While the formal framework is sufficiently mature and general enough for our purposes, the evaluation clearly suggests that further research is needed on the practical implementation. An experimental evaluation indicates that the introduction of negative constraints does not have a severe impact on the runtime performance, such that small and medium-sized models can be sufficiently well handled in real-world applications. For implication constraints, the scalability results show that further improvements are necessary for practical use, though.

Although extensive performance tests were conducted using two example TGGs, further experiments with benchmark examples and other (industrial) use cases can give insights as to whether the test results are generalisable, and which effect the size of the metamodels, the number and size of TGG rules and other characteristics have on the runtime performance. Finally, we plan to generate graph constraints directly from metamodels, such that the handling of, e. g., multiplicity constraints can be fully automated in the practical framework.

While in this chapter, the expressive power of the hybrid approach was increased, the upcoming Chap. 7 and 8 will add a fifth operation, namely *concurrent model synchronisation*, to the framework. With this operation, it is not only possible to transform entire models into another domain, but propagate changes *incrementally*, such that two domain experts can work in parallel on semantically interrelated models.

7 A Fault-Tolerant Approach to Concurrent Model Synchronisation

In collaboration scenarios, multiple (teams of) domain experts work concurrently on semantically interrelated models. As such models describe a software system from different perspectives, maintaining and restoring consistency between these models is a crucial task. Concurrent model synchronisation denotes the task of keeping these models consistent by propagating changes between them. This is challenging as changes can contradict each other and thus be in conflict. In contrast to tasks such as (unidirectional) transformations or consistency checks, just a few approaches exist that address this problem even for consistent input models. For the faulty inputs, to the best of our knowledge, no solution has been proposed yet.

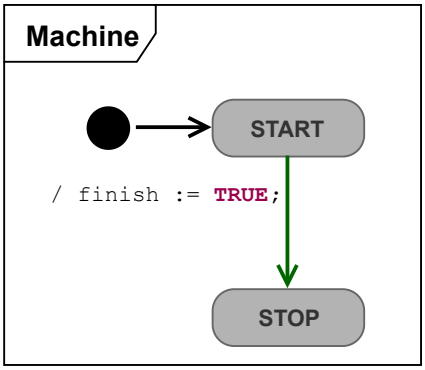
We extend the hybrid approach based on TGGs and ILP to overcome these issues: Besides the current state of the models (which usually involves all three parts of the triple), information about recent changes, i. e., modifications since the last synchronisation took place, are given as input to the operation. This information is used to refine the objective function in order to balance different optimisation goals. The implementation of the hybrid approach is extended by this fifth operation and evaluated regarding scalability for growing model sizes and an increasing number of changes.

The chapter is structured as follows: After a brief introduction (Sect. 7.1), Sect. 7.2 compares the proposed concurrent synchronisation strategy to related approaches. An overview of the necessary adaptions of the formal framework is given in Sect. 7.3. The notion of operational rules is generalised and extended towards so-called rule variants in Sect. 7.4, which are required to reasonably support synchronisation scenarios. The formal framework for fault-tolerant consistency management is extended by a parametrised and configurable rating function in Sect. 7.5. The actual ILP construction and the resulting solution for an example instance is shown in Sect. 7.6. The results of an experimental evaluation are presented in Sect. 7.7, before Sect. 7.8 summarises the findings and assesses their contribution to a fault-tolerant consistency management framework.

7.1 Motivation

In a collaborative setting, restoring consistency after concurrent updates involves more challenges than forward or backward transformation of given models. Consider the small example in Fig. 7.1, which revives the introductory example of Fig. 1.3: Both the SysML engineer an the Event-B expert remove the fault which was present in Fig. 1.2, i. e., the dangling edge going out of the `START` state. While both experts agree on assigning the variable `finish` the value `TRUE` via this transition, the transition's target is differs for the two models: The SysML engineer defines that the `STOP` state should be the target, whereas the Event-B expert makes the transition a self-loop on the `START` state, simultaneously removing the unreachable `STOP` state. The latter change produces a *conflict*: It is not possible to determine a solution in which both changes can be considered.

Propagating conflicting changes can leave the models in an inconsistent state, which in general is not a desired result. In the scope of this thesis, the handling of inconsistent



(a) SysML model

```
MACHINE Machine
VARIABLES START ~~STOP~~ finish
INVARIANTS ...
EVENTS
  INITIALISATION ≙
    ...
  COMPLETION ≙
    WHEN
      isin_START : START = TRUE
    THEN
      ~~leave_START : START := FALSE~~;
      act1 : finish := TRUE;
    END
END
```

(b) Event-B code

Figure 7.1: Concurrent changes on both models result in a conflict

inputs should be possible (which is required here because the input models cannot be completed to a triple of the example TGG's language). Bringing the models closer to a consistent state while modifying them to the smallest possible extent is still a goal to pursue. A trivial solution to this problem would be to drop all conflicting changes. This, however, is likewise undesirable as it ignores the intentions of all users who made these changes. Furthermore, this strategy only succeeds on the assumption that the models have been in a consistent state before, which does not hold in the context of this thesis. Hence, we have to find a solution that finds a reasonable compromise between changes, while offering a consistent synchronised solution to the users, together with a set of remaining elements that were left over when computing a synchronised solution automatically.

A few strategies to restore consistency for concurrent modifications with different limitations were proposed in existing work. Frequently, a substantial amount of user interaction is required to finally reach a consistent state in case of conflicting changes [Egy07a, KPP08b, TA17], or the retrieved solution might still contain inconsistencies [Tra08], which should rather be considered as a limitation than a feature of the approach. Other approaches pose severe restrictions on the modelling language in use [PSG03, GW09], or on the set of allowed changes and conflicts which can be resolved [XHZ+08, XSHT09, XSHT13, HLR08]. These restrictions can be overcome by building up a search space of all possible solutions from which a range of adequate results can be retrieved [OPN20]. This search space grows rapidly with model size and number of changes, however, such that scalability becomes a severe problem. To the best of our knowledge, none of the proposed strategies supports arbitrary edits on the input models, which means that models are expected to be at least well-formed, i. e., intra-model consistency must be guaranteed.

In Chap. 5, a hybrid approach based on TGGs and ILP has been successfully applied to basic consistency management operations, such as forward and backward transformation and consistency checking, and was extended to the handling of graph constraints in Chap. 6. Guarantees for correctness and completeness could be provided in case of consistent input models, whereas a consistent sub-graph of maximum size can be determined for inconsistent inputs.

To address open issues in concurrent model synchronisation, we extend this hybrid, synergistic approach towards such scenarios: A novel operation constructs a search space of possibilities for propagating user edits, and computes an optimal solution with respect to a

rating function. Configurable parameters allow us to influence the importance of creations and deletions for the final result. In this manner, the line of fault-tolerant consistency management operations is continued: The concurrent synchronisation operation can cope with arbitrary graph edits on an input model, which itself is not required to be well-formed or consistent with the underlying TGG at any point of time.

7.2 Related Work

In general, concurrent model synchronisation approaches can be divided into constraint-based, search-based and propagation-based approaches, which come each with their own advantages and limitations.

Constraint-based approaches often use a relational specification language, which describes consistency relations over models. Using optimization techniques such as, e. g., ILP solving or SAT solving, they encode the search space of all feasible solutions and use a solver to find an (optimal) solution satisfying all constraints. Macedo et al. [MC13] find the closest consistent model that is again consistent w.r.t. all constraints and where user-preferences can be expressed by implementing a custom distance function for model changes. Yet, controlling the synchronisation process by designing such a distance function can be very hard and requires extensive domain knowledge. Furthermore, the scalability of such approaches is problematic as a large amount of constraints is necessary to encode graph properties, whereas our approach operates on rule-application level.

Search-based approaches explore the space of all solutions in order to find an optimal solution or present a rich set from which a user can pick the preferred one. As such, they are highly related to constraint-based approaches that also encode the search space of all possible solutions, however, without explicitly calculating it. Cicchetti et al. [CRE+10] calculate all closest sub-models that are consistent w.r.t. their consistency specification but this means that a lot of information may be dropped, e.g., user-changes, if they are considered to be inconsistent. Since they employ optimisation techniques to find constraint preserving solutions, their approach can also be considered as constraint-based. Orejas et al. [OPN20] recently presented a TGG-based approach, where in a first step, the grammar rules are used to find all possible parsings for the models. In a second step, these parsing are used to annotate the model w.r.t. whether these elements have been changed or need to be preserved. Given these annotations, contradicting changes can be detected for which all possible synchronisation solutions are generated using back-tracking. These solutions are then presented to the user to choose from. However, exploring all solutions may be very expensive and relying on the user to choose one might be very overwhelming.

Propagation-based approaches are probably the best investigated technique so far since it reuses sequential synchronisers to propagate changes round-trip-wise between models in order to restore consistency for concurrent changes. However, these approaches tend to suffer from several limitations. Some works [BG16, Egy07a, XHZ+08, KPP08b, PSG03, Tra08, XHZ+08, XSHT09] do not consider that changes can be in conflict and, thus, may not guarantee which change is propagated and dropped or assume that changes are a priori not in conflict. The approach of Buchmann et al. [BG16] employs hand-crafted transformations and Kolovos et al. [KPP08b] delegate the responsibility of implementing conflict detection and resolution to the user. Hence, both approaches cannot guarantee correctness. Other approaches pose restrictions to the structure of the consistency relation, e. g., that it is bijective [GW09] or that one model has to be an abstraction of the other and both have to be tree-like hierarchies [PSG03].

More advanced works [TA17, GW09, XSHT13, HLR08, HEEO12, GHN+13, KPP08b] in this area, implement a conflict detection and resolution framework. This is done by analysing whether the changes made during a propagation step contradict the user edits in the same model. However, Orejas et al. [OBE+13] showed that this kind of conflict detection is not deterministic and that the detection rate can depend on the order in which the propagation takes place. A recent TGG-based approach by Fritsche et al. [FKM+20] separates the phases of conflict detection and resolution, such that the result is deterministically computed. The approach enables the user to configure the conflict resolution strategy according to their preferences. As part of this chapter's evaluation, our hybrid approach will be compared performance-wise to the implementation of Fritsche et al. in eMoflon::IBeX.

Following similar arguments as for consistency checks and (unidirectional) transformations (Sect. 5.2), a gap in research can be identified for concurrent synchronisation as well: Constraint-based and search-based approaches provide enough flexibility to be used for handling faulty input models, scalability remains a severe issue. Propagation-based approaches, in contrast, scale well for large models and numerous conflicting changes, but are not able to handle inconsistent input models. We will therefore continue with the extension of the hybrid approach to develop a fault-tolerant synchronisation strategy for concurrent updates.

7.3 Solution Overview

Compared to the operations introduced in Chap. 5, several similarities and differences can be observed. The concurrent synchronisation operation receives a complete triple as input, which is similar to the CO operation at the first glance. The input triple is, however, not homogeneous: Along with the model elements, the information whether the respective nodes and edges were *created*, *deleted*, or remained *unchanged* since the last synchronisation took place, is of major importance. The respective sets of elements are denoted as *delta structures* in the following. With these additional structures, changes over time are made known to the operation, opposed to the previously introduced operations, which can be considered as "stateless".

The fact that the input models contain both recently created and already existing elements makes it necessary to handle these elements in different ways, leading to a new notion of operational rules. For tasks such as forward and backward transformation or consistency checking, the *domain* (i. e., source, target or correspondence model) prescribes which elements should be marked and created by an operational rule. For synchronisation tasks, this is not possible because the operation receives a complete triple and change information as input. Instead, it makes sense to check the unchanged part of the triple for consistency and translate the created elements to the respective other domains. This makes it necessary to search for matches of multiple operational rules for a single declarative rule, which also have interdependencies that guarantee the language membership of the produced result. As we will see in Sect. 7.4, there is a need for further forms of operational rules besides those for CO, CC, FWD_OPT and BWD_OPT, leading to a generalised notion of operational rules. A last difference between concurrent synchronisation and the other operations is that depending on whether the elements belong to one of the delta structures, their importance for the final solution is different: While it is very desirable to involve all recently created elements in the synchronised solution, recent deletions should be preserved, if possible. These requirements are reflected in a refined objective function, which balances these interests via configuration parameters.

To address these challenges of the concurrent synchronisation operation, the hybrid approach is enhanced with:

1. a generalised notion of operational rules (Sect. 7.4)
2. delta-structures, and the derived rating of rule applications (Sect. 7.5)
3. a refined and parametrised objective function (Sect. 7.6)

The work-flow for concurrent synchronisation is depicted in Fig. 7.2 as an adaption of the work-flow of Fig. 6.1. In a first step (A), operational rules are generated from the declarative TGG rules. In contrast to the operational rule generation for CO, CC, FWD_OPT and BWD_OPT, the mapping between declarative rules and operational rules is not 1:1. For each declarative rule, multiple *rule variants* are generated which include rules for consistency checks, forward and backward transformation, but also other forms of operational rules. The previously mentioned delta structures are an additional input for the rule pattern matching step (B1). Another difference concerns the optimisation step (D): While for the other operations, the coefficient for each rule application candidate could be directly derived from the number of marked elements, for concurrent synchronisation, a *rating* is computed for each rule application that serves as coefficient in the objective function. Configuration parameters are provided as additional input to the operation. They describe the relative importance of preserving deletions, creations, and unchanged structure, and thereby influence the rule applications' rating.

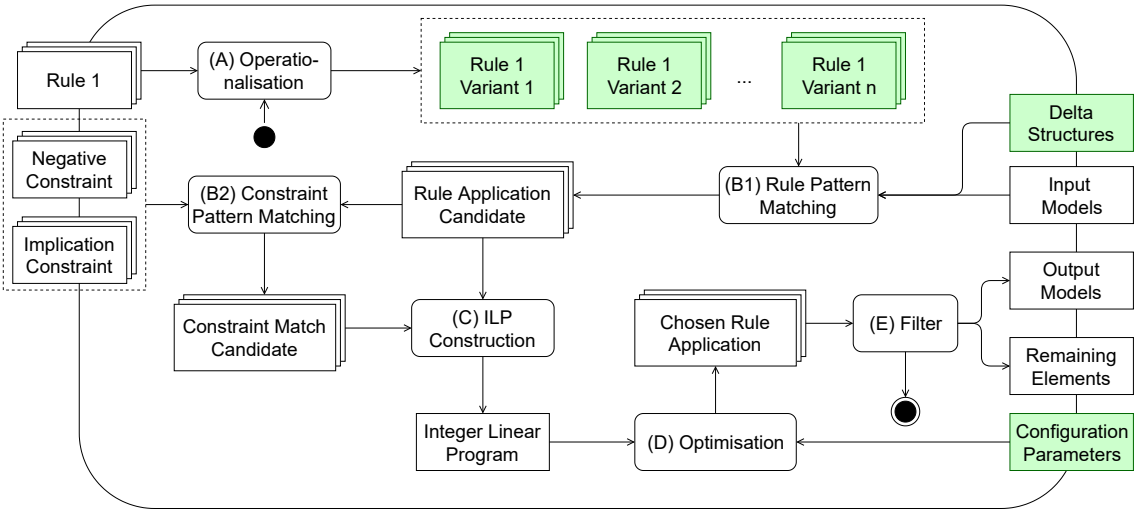


Figure 7.2: Work-flow for fault-tolerant concurrent model synchronisation

7.4 Operational Rules

A key feature of the TGG approach is that various consistency management operations can be derived from the descriptive rules, as discussed in Chap. 5. For the concurrent synchronisation operation, we generalise the definition of operational rules (cf. Def. 5.4), such that the domain is not the only decisive factor for which elements are to be marked or created any more: All possible subsets of green elements (with the exception of the empty set) can form the set of marked elements, whereas the remaining elements are created by applying the operational rule. This generalisation is necessary because there is often no

1:1-mapping of model changes and TGG rule applications, which will be demonstrated concretely at the end of this section. To overcome this problem, different strategies are proposed in existing work. The approach of Fritsche et al. [FKM+20] makes use of so-called short-cut rules [FKST18], which combine revoking and applying TGG rules in a single step. This mechanism, however, assumes that elements can be deleted via a short-cut rule application, which is not compatible with our hybrid approach.

Formally, a subset of the created elements in r is added to the left-hand-side OL of the operational rule *or*, receiving a marker after rule application instead of being (re-)created (cf. Fig. 7.3). Note that the subset must be non-empty, so the operational rule is not equal to the declarative rule.

Definition 7.1 (Operational Rule and Marking Elements)**.**
Given a triple rule $r : L \to R$, an ***operational rule*** *$or : OL \to OR$ for r is constructed as depicted in Fig. 7.3. An element $e \in elem(R)$ is a* ***marking element*** *of or iff ($\exists e' \in elem(OL)$ with $or_S(e') = e \vee or_C(e') = e \vee or_T(e') = e) \wedge (\nexists e'' \in elem(L)$ with $r_S(e'') = e \vee r_C(e'') = e \vee r_T(e'') = e)$.*

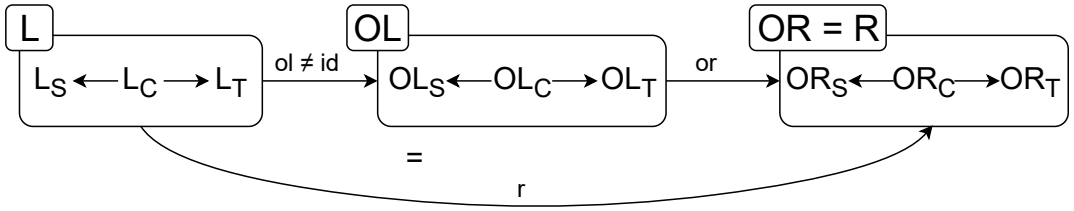


Figure 7.3: Construction of operational rules

The operational rules of Def. 5.4 can be regarded as special cases of generalised operational rules, as stated in Def. 7.2. The respective operational rules resemble FWD_OPT (BWD_OPT) rules, if the set of marking elements exclusively contains all elements of the source (target) side. If, in contrast, all created elements of the declarative rule are taken into the subset, the operational rule can be denoted as a CO rule.

Definition 7.2 (Special Rules)**.**
For some operational rules, intuitive names can be found, if OL is composed of graphs of L and R according to the this table:

Operation	OL_S	OL_C	OL_T
CC	R_S	L_C	R_T
CO	R_S	R_C	R_T
FWD_OPT	R_S	L_C	L_T
BWD_OPT	L_T	L_C	R_T

With the aforementioned operational rules, it is possible to mark elements that can be matched to a full TGG rule application (CO), to transform created elements into the other domain (FWD_OPT, BWD_OPT), and to align elements which are new on both sides (CC). However, there are cases in which other *rule variants* in between these operations are beneficial to reflect the users' edit operations properly. Provided that further elements can be matched that fit into the scope of the rule, it is better to reuse these instead of creating them anew (such rules were proposed in a similar fashion by Orejas et al. [OPN20]).

Figure 7.4 depicts two variants of the *StateToVariable* rule that do not match any of the already known rule variants, as they both mark and create elements of the same

domain. The left rule variant is useful in case a state `s` was moved to a new region `r` in the SysML model, such that new connections can be drawn in the Event-B model. This rule variant assumes that the variable `v` and the invariant `i` of the Event-B model, as well as the respective correspondence links, already exist, such that only two new edges must be created. With the FWD_OPT rule variant, in contrast, these elements would be re-created, which is unnecessary in this case. Furthermore, the rule variant allows to reuse existing elements of the Event-B model that are connected to `v` and `i`, which would not be possible otherwise. The right rule variant, in turn, depicts a useful variant for situations where a variable and an invariant were moved to another Event-B machine, re-locating the affected SysML state.

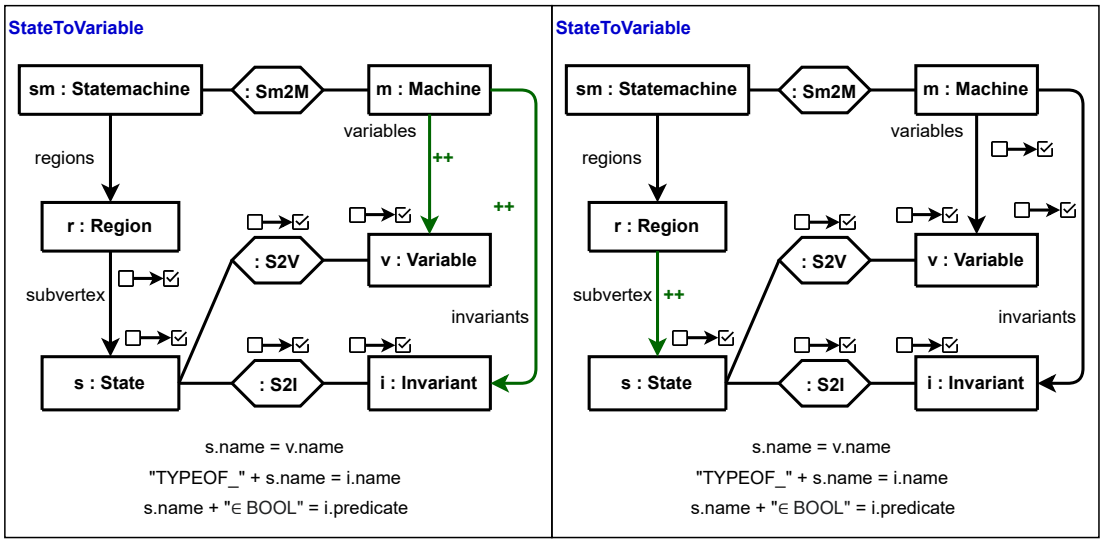


Figure 7.4: Rule variants for StateToVariable

To find a solution for the concurrent synchronisation operation, we determine matches for different operational rules simultaneously. In this way, it is possible, e. g., to cover the unchanged parts of the triple with CO matches. We cannot only rely on the CO operation, though, as it only matches already existing elements. Hence, especially when nodes and edges of the create delta need to be translated, also matches for other operational rules need to be collected in order to propagate changes. As the handling of delta structure elements was only discussed in an intuitive form up to here, a precise specification follows in the upcoming Sect. 7.5.

7.5 Rating of Rule Applications

This section presents how elements of the input models can be partitioned into different sets, such that recent changes, i. e., creations and deletions, can be formally represented. Based on these sets and multiple configuration parameters that describe the importance of preserving creations and deletions, a *rating* for each rule application candidate can be computed that quantifies the potential contribution of the candidate in the synchronised solution. The ratings will be used to form the refined objective function of the optimisation problem (cf. Sect. 7.6).

Similar to the basic operations of Chap. 5, the concurrent synchronisation operation receives a triple as input, which is denoted as *starting triple graph* (cf. Def. 5.1), and for which consistency should be restored. Elements of this triple graph are to be marked dur-

ing the synchronisation process, whereas the operation can add further elements as well. To track the dependencies between operational rule applications, we distinguish between required, marked and created elements for each rule application (cf. Def. 5.3). For indicating those nodes and edges which were created or deleted since the last synchronisation took place, parts of the triple graph are labelled as such, whereas the remaining elements are considered as unchanged. The labelled elements form a *delete delta* and a *create delta*, respectively. The third delta structure is denoted *induced delta*, and consists of all elements added by the operation. Note that only elements of the starting triple graph can be marked, as marking an element which was created during the operation would mean that declarative rules would create it twice, which is impossible.

Definition 7.3 (Delta Structures and Unchanged Elements)**.**
*Let $G_0 = G_{0_S} \leftarrow G_{0_C} \rightarrow G_{0_T}$ be a triple graph given as input to the operation, denoted as **starting triple graph**. Let $G^- = G_S^- \leftarrow G_C^- \rightarrow G_T^-$ and $\overline{G}_S \leftarrow \overline{G}_C \rightarrow \overline{G}_T$ be triple graphs, and let $\delta^- : G^- \rightarrow \overline{G}$ and $\delta^+ : \overline{G} \rightarrow G_0$ be triple graph morphisms, such that δ^- adds elements labelled as deleted to G^-, and δ^+ adds elements labelled as created to $\overline{G}$. The following sets are defined:*

- $unchanged(G_0) := elem(G^-)$
- $dltDelta(G_0) := elem(\overline{G}) \setminus elem(G^-)$
- $crtDelta(G_0) := elem(G^+) \setminus elem(\overline{G})$
- $indDelta(G) := \bigcup_{d \in \mathcal{D}} crt(d) = elem(G) \setminus elem(G_0)$

In order to assess the contribution of a rule application candidate to a desirable consistent solution, a coefficient is computed for each rule application; the number of marked and created elements is considered, as well as if these belong to one of the three deltas. To model the effect of user edits, we weight elements of create, delete and induced delta, and therefore determine an individual *rating* with configurable parameters for each rule application, which is described in the remainder of this section. Positive values suggest to use this rule application for the synchronised result, whereas negative values recommend to avoid it, if possible.

Unchanged elements should usually remain in the final solution because no recent change indicates something different, and are therefore assigned a normalised weighting factor of 1. The motivating example of Fig. 7.1 contains several unchanged elements, which are coloured black in the visual notation of a triple graph instance (cf. Fig. 7.5). For existing structure, CO matches can be determined, as all necessary elements for such a match are present in the input models. In the concrete example, this holds for matches for *StatemachineToMachine* (d_1), *PortToVariable* (d_2), *StateToVariable* (d_3), *PseudostateToActions* (d_4), *TransitionToEvent* (d_5, d_6), and *TargetStateToEnterAction* (d_7). Note that d_4 and d_5 overlap in their markings: For marking the initialisation transition `t`, matches for two different rules can be found. d_5, however also requires markings of d_4, such that we have an example for a superfluous rule application according to Def. 6.12 here.

Deleted nodes and edges, which are depicted in red and with a −− mark-up, were recently removed by a user and therefore should not be part of the final solution. However, they might still be necessary context for other rule applications which in turn cannot be chosen otherwise (which is the main argument for not deleting them right away in this approach). Therefore, marking deleted elements should be penalised by assigning them a negative factor $\alpha < 0$. The CO rule applications for *StateToVariable* (d_8) is shown in Fig. 7.6.

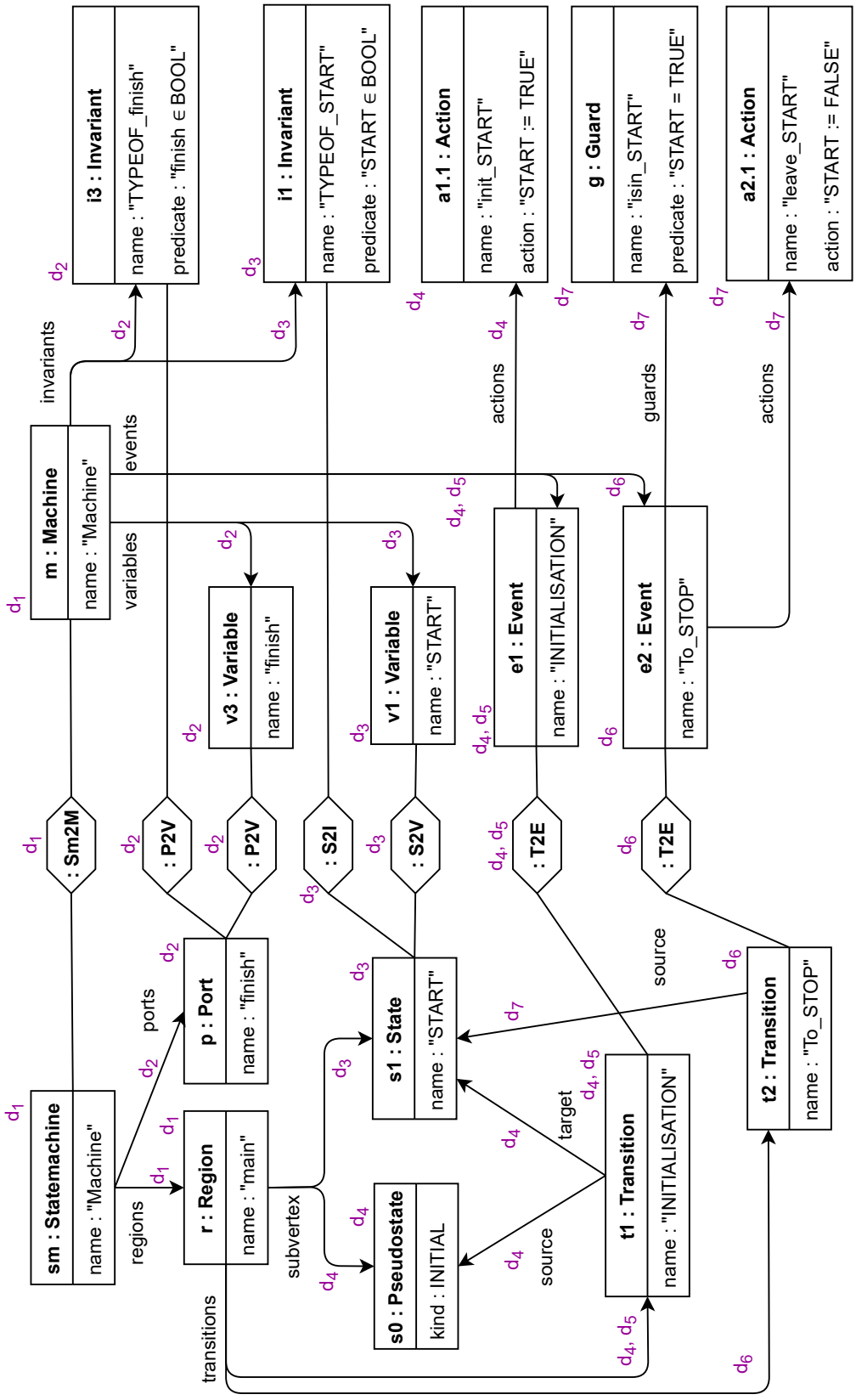


Figure 7.5: Unchanged elements

Besides deleted elements in the target model, also unchanged source and correspondence elements are concerned, such that a rating of $4 + 4\alpha$ can be computed. It clearly depends on the rating of the dependent rule applications and on the choice of α, whether these elements are part of the solution. Setting α to -1 or less results in a negative rating for d_8, which means that d_8 is only chosen if dependent rule applications receive a positive rating that can balance this negative value.

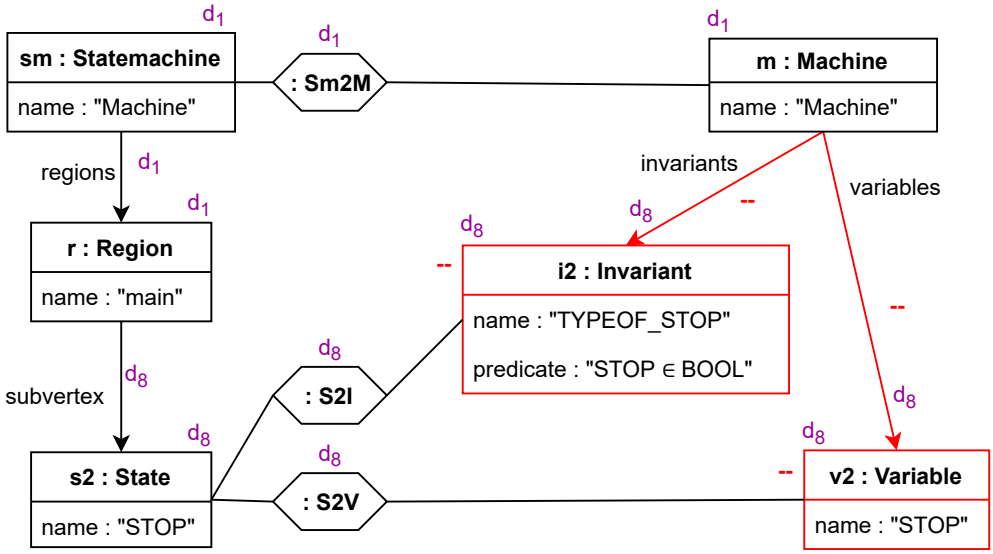


Figure 7.6: Delete delta

An example for a rule application that depends on a CO rule application involving deleted elements is d_9 (*TargetStateToEnterAction*) as presented in Fig. 7.7. The translation of *create delta* elements is especially important, as ignoring them would lead to an unnecessary loss of information about recent user edits. In triple graph instances, create delta elements can be identified via their green colour and their ++ markup[1]. To increase the probability for them to be contained in the synchronised result, we weight these elements with a factor $\beta > 1$. In the general case, edits in one model must be propagated to the other model to form a consistent triple, which is not manageable only with matches of CO rule applications. To properly translate elements of the create delta, FWD_OPT (d_9) and BWD_OPT (d_{10}) rule applications are necessary. In this example, both d_9 and d_{10} are applications of *TargetStateToEnterAction*, that shall propagate user edits on the source and the target model to the respective other model. In order to do so, further elements must be created, which are part of the induced delta and marked up with `+?`. On an intuitive level, we have already seen that these two edits are conflicting. In the course of this chapter, it will be demonstrated how this conflict becomes evident by constructing the optimisation problem.

Besides the weighting factor for elements of the create delta, the handling of the *induced delta* is essential for computing the rating of d_9 and d_{10}. Nodes and edges of the induced delta are created as part of the synchronisation process, such that their presence is neither desired nor unwanted. It seems reasonable, though, to prefer (re-)using existing elements over creating duplicates, such that induced delta elements are weighted with a negative factor $\gamma < 0$. For d_9, this results in a rating of $\beta+2\gamma$, and in a rating of $2\beta+\gamma$ for d_{10}. As it is usually necessary to add such elements in order to translate user edits, this "drawback"

[1]Note that this visual variable has a different meaning for TGG rules

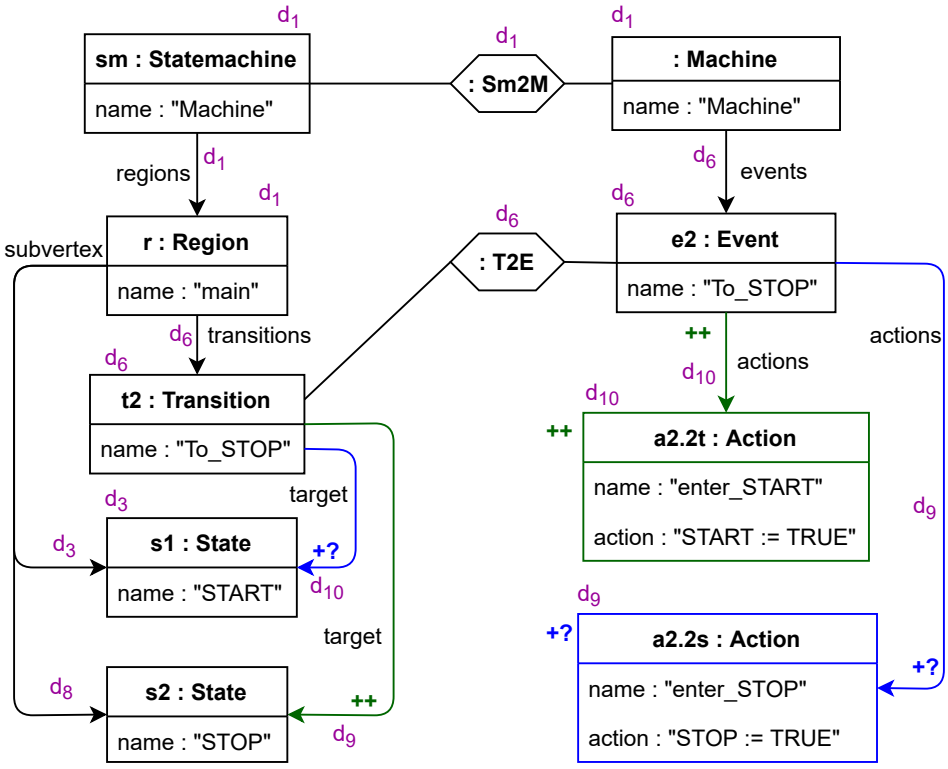


Figure 7.7: Create delta

must be compensated by a suitable choice of β for the create delta. Independent of the concrete parameter choice, the rating of d_{10} is better due to the value range for β and γ, suggesting to prefer the target model change over the source model change. The decision is, however, a result of the global optimisation process.

For the previously introduced operations CO, CC, FWD_OPT and BWD_OPT, the induced delta (which we simply denoted as *created elements* in Def. 5.3, i. e., elements that are created during the consistency management process) does not have any influence on the objective function: The optimisation goal is to maximise the number of marked elements, independent from the size of the resulting triple. To argue why the situation is different for the concurrent synchronisation process, the last user edits on the example instance shall be considered (cf. Fig. 7.8). Both users express that when the transition `t2` fires, the value of `finish` shall be `TRUE`.

To process these edits, one could translate them separately by applying FWD_OPT (d_{11}) and BWD_OPT (d_{12}) rules for *EffectToAction*, which would lead to two identical effects and actions each. However, it would be also possible to just create a correspondence between these elements (provided that the attribute condition is fulfilled) with a CC rule application (d_{13}). As a result, ratings of $2\beta + 3\gamma$ for both d_{11} and d_{12} can be determined, whereas d_{13} receives a rating of $4\beta + \gamma$. As γ is negative, the sum of the two values for d_{11} and d_{12}, $4\beta + 6\gamma$, is less than the coefficient for d_{13} ($4\beta + \gamma$), such that the latter will be preferred for the final solution.

For the example instance - due to its simplicity - other rule variants than the special rules (CO, CC, FWD_OPT, BWD_OPT) are not necessary. To demonstrate the application of other rule variants on an example instance, Fig. 7.9 shall be considered. Here, the SysML engineer has moved the state `s2` from the region `r1` of the statemachine `sm1` to the region `r2` of the statemachine `sm2`. The rule application candidates d_a, d_b and d_c (indices

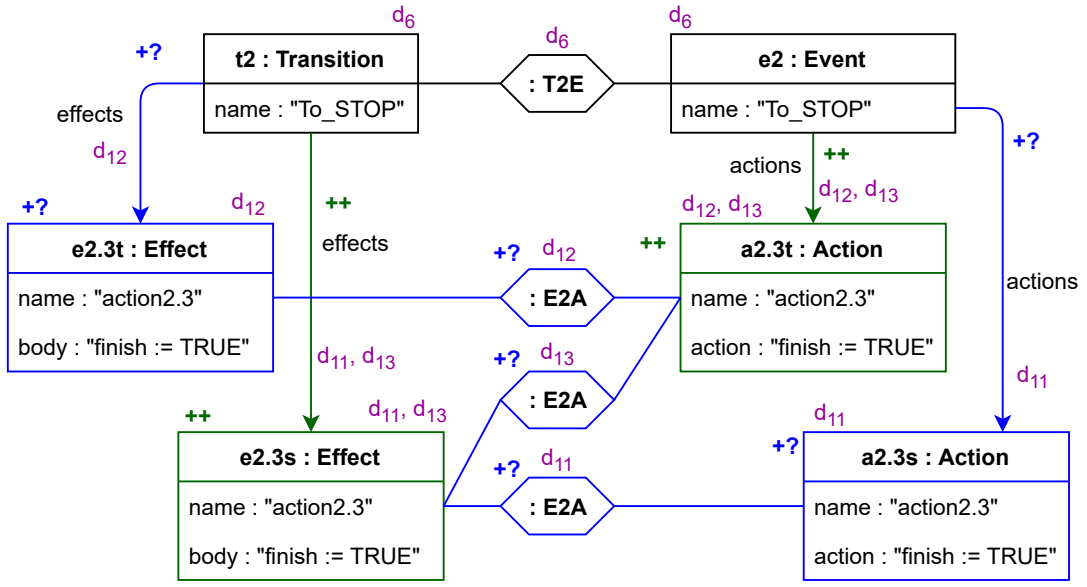


Figure 7.8: Induced delta

are letters to emphasize that the running example of this chapter is left) are CO rule applications, whereas d_d is an application candidate for the left rule variant of Fig. 7.4. The intended change on the SysML model can be incorporated by choosing d_a, d_b and d_d, such that the deleted edge in the SysML model and the outgoing edges of the machine `m1` in the Event-B model remain unmarked. The possibility of reusing an element is especially beneficial if it (indirectly) provides context for various other rule applications, such deletions and recreations of the dependent elements can be avoided [FKST18].

Based on the declarative TGG rule, such rule variants can be systematically generated by enumerating all combinations of creating and marking the green elements of the declarative rule, as long as it does not violate the dangling edge condition [KLKS10] (i.e., if an edge is required as context, its source and target nodes must be context as well). This however means that the number of variants for a descriptive TGG rule grows exponentially with its number of green (created) elements. To implement the approach efficiently, a subset of promising rule variants must be chosen, which is described at the end of Sect. 7.6.

In the presented form, the ratings are only a local assessment for single rule application, while the determination of an optimum synchronisation solution is still an open issue. The rating of entire solution alternatives is done via transferring the ratings of rule applications to coefficients in the objective function of an ILP. Based on the outcome of the optimisation process, the final solution can be constructed, which will be explained in detail in Sect. 7.6.

7.6 Constructing the Optimisation Problem

To find an optimum solution for the concurrent synchronisation problem, an ILP is constructed in a similar fashion as presented in Chap. 6. The constraint specification is done in the same way, whereas the objective function set-up is extended towards individual ratings. For consistency checking and forward and backward transformation, the number of marked elements is maximised (Def. 6.7), such that each rule application candidate is weighted with the respective number of marked elements. For concurrent synchronisation, in contrast, each rule application candidate is weighted with its rating as discussed in

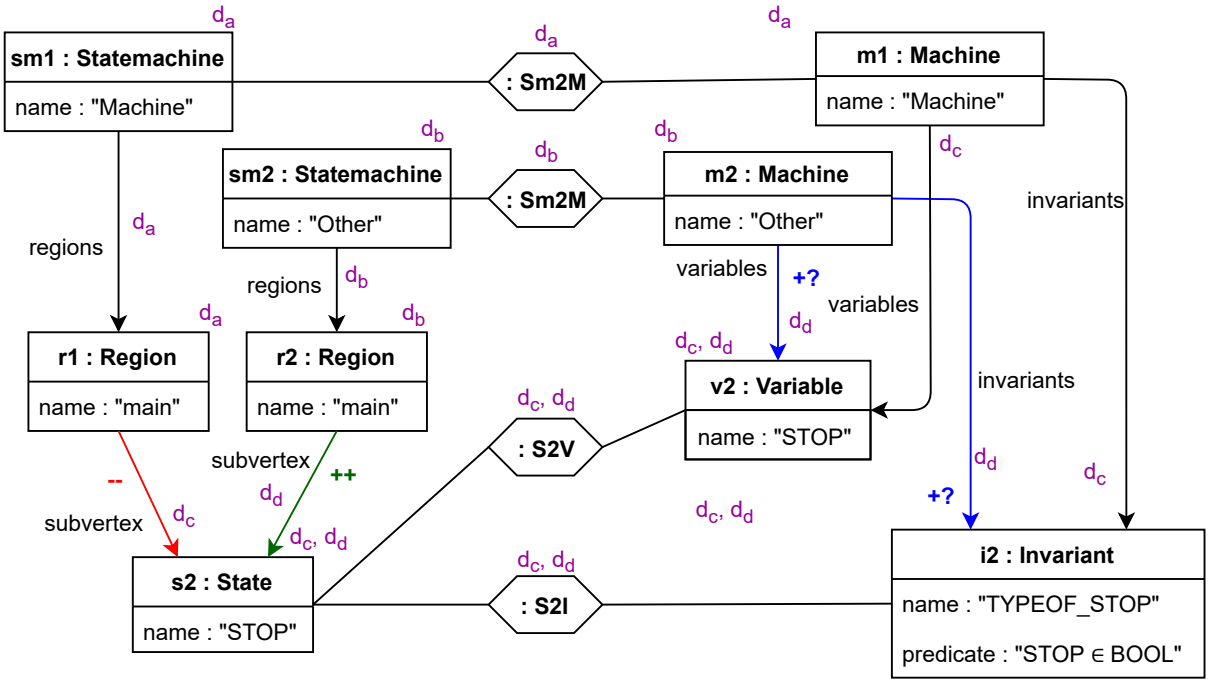


Figure 7.9: Rule variant application

Sect. 7.5. In this way, rule applications which contribute to a good synchronisation solution (i. e., have a positive rating) are more likely to be chosen than rule applications with a negative rating. At the same time, it must be guaranteed that the result is consistent with respect to the TGG at hand and complies to the defined schema. Therefore, the decision whether a particular rule application is chosen is still the result of a global optimisation process, such that rule applications with a negative rating can be part of the final solution if they, e. g., provide context for other rule applications with a positive rating.

The utility of each rule application is assessed based on the number of elements it marks or creates for the different deltas. Marking unchanged elements or elements of the create delta increases the coefficient for the rule application in the objective function, whereas marking deleted elements or creating further elements during the operation decreases it, which is specified by user-defined parameters α, β and γ. The optimum solution, i. e., the solution which satisfies all constraints and maximises the objective function, determines the choice of rule applications for the final solution. The objective function of the optimisation problem (Def. 6.7) is therefore refined to a weighted sum of marked and created elements.

Definition 7.4 (Optimisation Problem (Refined))**.**
Let $\alpha \in (-\infty; -1), \beta \in (1; \infty), \gamma \in [-1; 0)$ be configuration parameters for the operation. Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules. The ILP to be optimised is constructed as follows:

$$
\begin{aligned}
max.\ \Big(\sum_{d \in D} \quad & |mrk(d) \cap unchanged(G_n)| \cdot \delta \\
& + \alpha \cdot |mrk(d) \cap dltDelta(G_n)| \cdot \delta \\
& + \beta \cdot |mrk(d) \cap crtDelta(G_n)| \cdot \delta \\
& + \gamma \cdot |mrk(d) \cap indDelta(G_n)| \cdot \delta \ \Big)\ s.t. \\
markedAtMostOnce(G_0) \wedge\ & context(D) \wedge context(G_n) \wedge acyclic(D) \wedge sat(G)
\end{aligned}
$$

When comparing Def. 6.7 and Def. 7.4, it becomes apparent that the concurrent synchronisation operation can be regarded as a generalisation of the other four operations: Assuming that the create and delete deltas are empty, and setting $\gamma := 0$, the problem specification is equal for all five operations. Regardless of the parameter values, the synchronisation of updates on a single model can be considered as a special case of concurrent synchronisation as well. While a considerable amount of research exists for this consistency management task already, we propose a fault-tolerant strategy to solve the synchronisation problem.

The last term of the optimisation problem specification, *sat(G)*, indicates that the concurrent synchronisation operation supports graph constraints as introduced in Chap. 6. To understand why this feature is necessary to produce a suitable solution of the example instance, we consider a part of the SysML model as depicted in Fig. 7.10. Each transition should have exactly one source and target state, which is guaranteed by using the graph constraints from Fig. 6.2. Matches for premise patterns (p_i) are annotated in blue, whereas matches for conclusion patterns (c_i) are annotated in red. p_{14} and p_{16} are premise matches for the constraint *TransitionHasSourceState*, and c_{19} and c_{21} are matches for the respective conclusions. Similarly, p_{15} and p_{17} are premise matches and c_{20}, c_{22} and c_{23} are conclusion matches for the constraint *TransitionHasTargetState*. p_{18}, finally, is a match for the negative constraint *NoTwoTargetStates*. With these constraints, it can be guaranteed that d_4, d_7, and either d_9 or d_{10} are chosen.

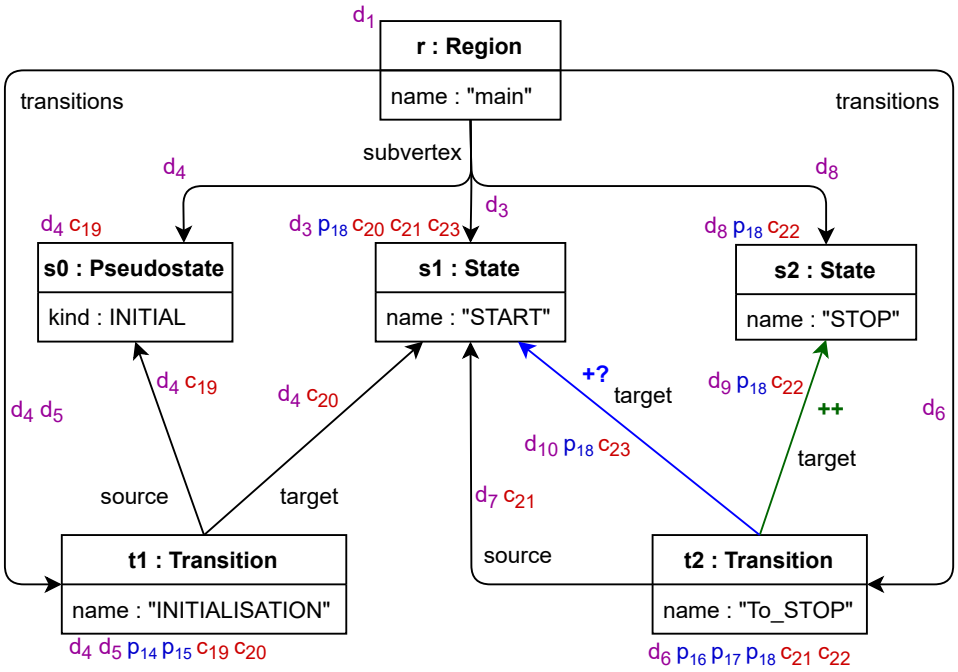


Figure 7.10: Source model constraints

Before all bits and pieces are put together to construct the optimisation problem for the running example, the fault-tolerance of the presented approach is briefly demonstrated. In Fig. 7.11, faulty user-edits on the example instance are depicted which violate the source model's well-formedness. The state `s1` is moved from the region `r1` to `r2`, leaving the former disconnected from the rest of the source model. Furthermore, a `subvertex` edge connects the statemachine `sm` and the state `s1` directly, which is not possible with respect to the metamodel. The fault-tolerant synchronisation operation would accept this input

and drop both the isolated region `r1` and the misplaced `subvertex` edge, producing a consistent result. Although other (potentially better) solutions for error-handling are possible, the presented concepts can be applied to such input models as well.

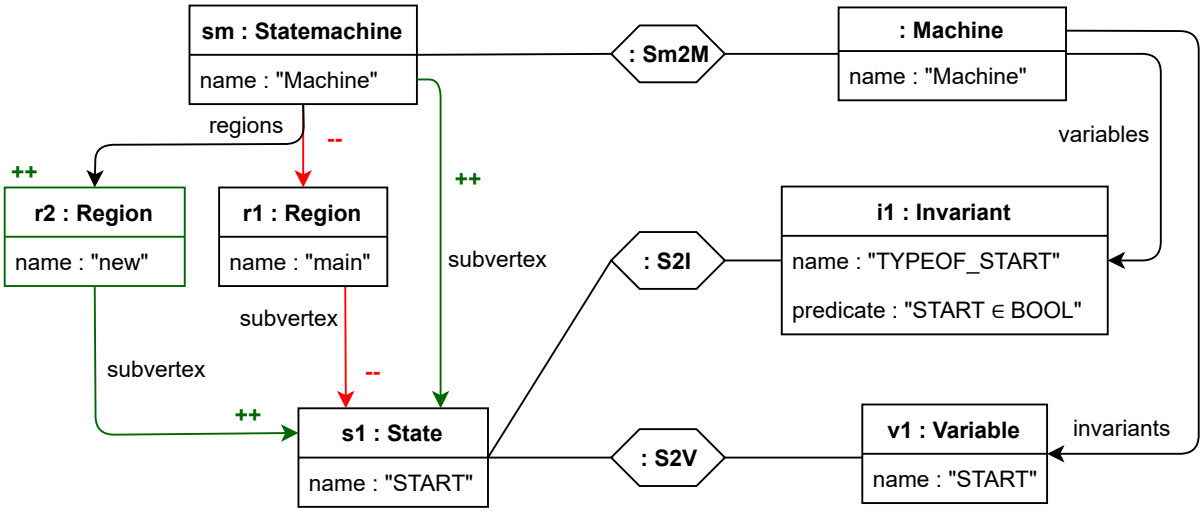


Figure 7.11: Faulty input model

By merging Fig. 7.5, 7.6, 7.7, 7.8, and 7.10, the input triple for the optimisation process can be formed. A tabular overview of all rule application candidates including their rating and all possible constraint pattern matches is provided in Tab. 7.1. The "Cand." column contains the rule applications and constraint pattern matches, in bold font for chosen candidates (the associated binary variable δ_i, π_i, or γ_i is set to 1) and in parentheses and normal font for others. The "Coeffic." column contains the coefficients in the objective function depending on the parameters α, β and γ, such that the objective function is the sum of $\delta_1 \dots \delta_{13}$ weighted with the coefficients of this column.

After setting up the objective function and adding all necessary constraints, the ILP can be solved to find a subset of rule applications which maximises the objective function, and thereby a solution for the concurrent synchronisation problem which is optimal according to our metrics. In contrast to the four other operations, marking the whole triple is a rare exception (or even not desirable in case of deletions) for the concurrent synchronisation operation. Therefore, the returned sub-triple can be regarded as the optimum solution according to our metrics, independent of how many elements of the input models are marked. The result of this process clearly depends on the choice of the parameters: Smaller values for α will decrease the likelihood of keeping elements that were marked as deleted, while increasing β increases the probability of taking over created elements to the final solution. An example solution for $\alpha = -5, \beta = 5$ and $\gamma = -1$ is shown in Fig. 7.12.

Apparently, we were able to avoid choosing rule applications with a negative rating, which means that all deletions could be propagated. For the new effect `e2.3s` and the new action `a2.3t`, we were able to create a correspondence and correlate them instead of translating each of them separately. Regarding the decision whether the target of the transition `t2` should be the `START` state `s2` (as suggested by the SysML engineer) or the `STOP` state `s1` (in accordance with the Event-B expert), the latter solution is preferred. Only one edge from `t2` to `s1` must be added by the operation, while all creations and deletions can be preserved. This in some way reveals a weakness of the approach: Independent of the choice of the configuration parameters, this solution is preferred due to the focus on maximising the rewards per element, although the other solution seems to be

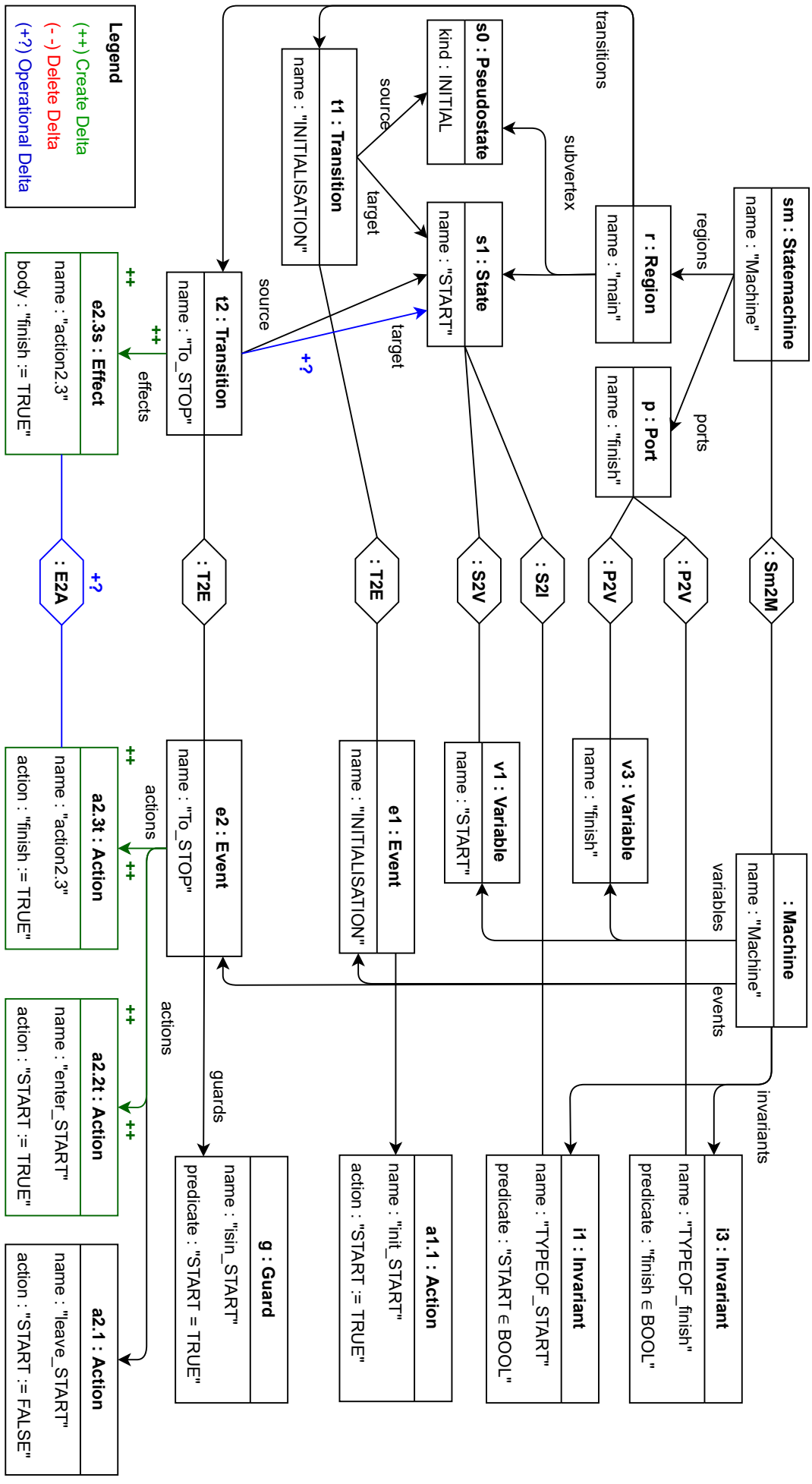


Figure 7.12: Final solution

Cand.	Rule / Constraint	Type	Constraints			Coeffic.
$\boldsymbol{d_1}$	StatemachineToMachine	CO				5
$\boldsymbol{d_2}$	PortToVariable	CO	$\delta_2 \leq \delta_1$			8
$\boldsymbol{d_3}$	StateToVariable	CO	$\delta_3 \leq \delta_1$			8
$\boldsymbol{d_4}$	PseudostateToActions	CO	$\delta_4 \leq \delta_1$	$\delta_4 \leq \delta_3$	$\delta_4 + \delta_5 \leq 1$	11
(d_5)	TransitionToEvent	CO	$\delta_5 \leq \delta_1$	$\delta_5 \leq \delta_4$		5
$\boldsymbol{d_6}$	TransitionToEvent	CO	$\delta_6 \leq \delta_1$			5
$\boldsymbol{d_7}$	SourceStateTolLeaveAction	CO	$\delta_7 \leq \delta_1$	$\delta_7 \leq \delta_3$	$\delta_7 \leq \delta_6$	5
(d_8)	StateToVariable	CO	$\delta_8 \leq \delta_1$			$4 + 4\alpha$
(d_9)	TargetStateToEnterAction	FWD_OPT	$\delta_9 \leq \delta_1$	$\delta_9 \leq \delta_6$	$\delta_9 \leq \delta_8$	$\beta + 2\gamma$
$\boldsymbol{d_{10}}$	TargetStateToEnterAction	BWD_OPT	$\delta_{10} \leq \delta_1$	$\delta_{10} \leq \delta_6$	$\delta_{10} \leq \delta_8$	$2\beta + \gamma$
(d_{11})	EffectToAction	FWD_OPT	$\delta_{11} \leq \delta_6$		$\delta_{11} + \delta_{13} \leq 1$	$2\beta + 3\gamma$
(d_{12})	EffectToAction	BWD_OPT	$\delta_{12} \leq \delta_6$		$\delta_{12} + \delta_{13} \leq 1$	$2\beta + 3\gamma$
$\boldsymbol{d_{13}}$	EffectToAction	CC	$\delta_{13} \leq \delta_6$			$4\beta + \gamma$
$\boldsymbol{p_{14}}$	TransitionHasSourceState	Premise	$\delta_4 \leq \pi_{14}$	$\delta_5 \leq \pi_{14}$	$\pi_{14} \leq \gamma_{19}$	
$\boldsymbol{p_{15}}$	TransitionHasTargetState	Premise	$\delta_4 \leq \pi_{15}$	$\delta_5 \leq \pi_{15}$	$\pi_{15} \leq \gamma_{20}$	
$\boldsymbol{p_{16}}$	TransitionHasSourceState	Premise	$\delta_6 \leq \pi_{16}$	$\pi_{16} \leq \gamma_{21}$		
$\boldsymbol{p_{17}}$	TransitionHasTargetState	Premise	$\delta_6 \leq \pi_{17}$		$\pi_{17} \leq \gamma_{22} + \gamma_{23}$	
(p_{18})	NoTwoTargetStates	Negative Constraint	$\pi_{18} \leq 0$	$\delta_3 + \delta_6 + \delta_8 + \delta_9 + \delta_{10} \leq \pi_{18} + 4$		
$\boldsymbol{c_{19}}$	TransitionHasSourceState	Conclusion	$\gamma_{19} \leq \delta_4$		$\gamma_{19} \leq \delta_4 + \delta_5$	
$\boldsymbol{c_{20}}$	TransitionHasTargetState	Conclusion	$\gamma_{20} \leq \delta_3$	$\gamma_{20} \leq \delta_4$	$\gamma_{20} \leq \delta_4 + \delta_5$	
$\boldsymbol{c_{21}}$	TransitionHasSourceState	Conclusion	$\gamma_{21} \leq \delta_3$	$\gamma_{21} \leq \delta_6$	$\gamma_{21} \leq \delta_7$	
(c_{22})	TransitionHasTargetState	Conclusion	$\gamma_{22} \leq \delta_6$	$\gamma_{22} \leq \delta_8$	$\gamma_{22} \leq \delta_9$	
$\boldsymbol{c_{23}}$	TransitionHasTargetState	Conclusion	$\gamma_{23} \leq \delta_6$	$\gamma_{23} \leq \delta_8$	$\gamma_{23} \leq \delta_{10}$	

Table 7.1: Optimisation problem

at least equally suitable from the user perspective. This aspect motivates to strengthen the user involvement in concurrent synchronisation processes, which will be studied in more detail in Chap. 10.

For an efficient implementation, the set of operational rules has to be restricted to keep the number of matches manageable. On the one hand, we decided to restrict the set of generated rule variants (cf. Sect. 7.4) to a (according to our experience) practically relevant subset. In principle, all rule variants can also be matched on elements that are not contained in any delta. Yet, this increases the search space heavily and is in general not reasonable as we assume that elements that have not been altered are still consistent, i.e., the same elements should still correspond to one another. Therefore, we only apply CO rules on entirely unchanged structure. This restriction was applied already to solve the running example, otherwise far more rule applications would have to be considered.

Although a treatment of formal properties for the operation is out of scope for this operation, we can argue for them: In a straightforward manner, one can show the *correctness* of the operation: Only elements which are marked or created by a rule application that is chosen by means of ILP will be taken over into the final result. Thereby, constraints ensure that elements are marked at most once, that a rule is provided with its necessary context, and that the chosen rule applications can be sequenced, such that the generation of a consistent model with declarative rules is simulated. As the output triple only consists of elements which are marked or created by such rule applications, it is contained in the language of the TGG. Furthermore, the constraints which belong to $sat(G)$ (Def. 7.4) ensure schema-compliance. The arguments for *completeness* are not directly transferable

to concurrent synchronisation, though, because it is unclear what exactly this property means for this operation.

There is still some work to be done for proving the approach's termination due to the match collection phase, but as we succeeded in showing that it is possible to identify superfluous matches and stop the collection process for the other four operations, we are confident that termination can also be guaranteed for concurrent synchronisation. The main challenge will be to identify a restricted set of rule variants, for which termination can be proven. Completeness has to be re-interpreted for concurrent synchronisation, as conflicting changes make it impossible to propagate all changes to the respective other model and to reach a consistent state at the same time. As not all rule variants are taken into consideration, and we do not try to also translate existing elements, completeness cannot be guaranteed for the implementation. For concurrent synchronisation scenarios, Orejas et al. [OBE+13] defined further desirable properties, such as maximal user edit preservation. Although the objective function of our approach follows a very similar goal, it is up to future work to decide which properties are actually fulfilled. In order to compare our concepts to other approaches to concurrent model synchronisation, a benchmark should be established that helps to assess how limiting the heuristic choices made for the implementation are in practice. Our experience up until now indicates that good results are still obtained.

7.7 Evaluation

To assess the applicability of the approach in real-world scenarios, scalability is an important criterion. Both the sizes of the synchronised models and the number of conflicting changes can influence the runtime performance, whereby splitting up the runtime measurements according to the phases depicted in Fig. 7.2 helps to identify room for improvement. In particular, the following research questions shall be answered:

RQ1 How does the approach scale for increasing model sizes and an increasing number of conflicting changes?

RQ2 How are the time requirements for pattern matching, ILP construction and ILP solving related? Does the relation change for increasing model sizes or an increasing number of conflicts?

Setup: We implemented the concurrent synchronisation approach in eMoflon::Neo (Sect. 9.5). As a test example, we took the *JavaToDoc* TGG (Sect. A.2) with all rules depicted in Fig. A.5 and parameter values chosen as in Sect. 7.6, but without the constraints of Fig. A.6.

To obtain synthetic models, we used the model generator of eMoflon::Neo to produce random models with linear increasing sizes from 1178 to 13,323 elements, whereby the ratio of rule applications for *ClassToDoc* and *SubClassToDoc* to the remaining rules was 1 : 4. Conflicting changes of the two types presented in the running example were randomly inserted to simulate user edits on the model. Runtime measurements were taken for increasing model sizes and a fixed number of 100 conflicts (Fig. 7.13), and for an increasing number of conflicts and a fixed model size of 6446 elements (Fig. 7.14). For each configuration, the time needed for pattern matching, ILP construction, and ILP solving was measured for 7 repeated runs. As final values, the medians of the 7 test runs were taken to minimize the bias introduced by outliers. All performance tests were executed on a standard notebook with an Intel Core i7 (1.80 GHz), 16GB RAM, and Windows 10

64-bit as operating system. An installation of Eclipse IDE for Java and DSL Developers, version 2019-09 with JDK version 13 was used. The JVM running the tests was allocated a maximum of 4GB memory, and 4GB were allocated to the graph database Neo4j.

Results: The median runtime measurements, of which an overview was already given in prior work [WFA20], are plotted in Fig. 7.13 and 7.14, respectively, while a full overview of the measured data is provided online[2]. In Fig. 7.13, a slightly disproportionate growth of the runtime for increasing model sizes is observable (note that a logarithmic scaling is used to properly illustrate the growths of small and large values in one diagram).

As can be seen, the time consumptions differs noticeably between the three phases: While the time required for the construction of the ILP is almost negligible, more than 90% of the time is used for the ILP solving step, whereas the pattern matching step is much less time consuming. This is at first glance surprising as the previously presented scalability results for the hybrid approach (Sect. 5.8 and 6.6) indicate that pattern matching involves more efforts, but can be explained by the complexity of interdependencies between rule applications in a concurrent synchronisation scenario. For larger models, this problem appears to be more severe than for smaller instances. According to the results of Sect. 6.6, scalability could be even more critical when adding implication constraints.

In contrast to that, the number of conflicting changes has only a small effect on the runtime (cf. Fig. 7.14). The time for the pattern matching and ILP construction step remains almost constant, and only a slight increase can be noticed for solving the ILP. This reflects the concept of a global view of the synchronisation problem: The approach is not based on change propagation, but represents changes as an additional label on graph structures.

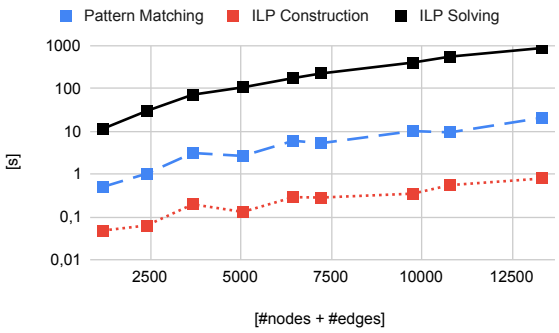


Figure 7.13: Runtime measurements for increasing model sizes

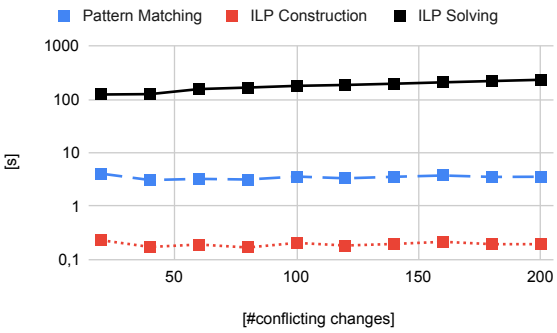


Figure 7.14: Runtime measurements for increasing number of conflicts

The experiment shows that the hybrid approach to concurrent model synchronisation struggles with medium-sized models already. Furthermore, the measured runtime seems to depend on the model size rather than on the number of conflicting changes. The limited scalability is to be expected, though, for several reasons: First, as part of a fault-tolerant consistency management framework, the operation must be able to process inconsistent input models, as well as inconsistent changes, which enlarges the set of possible solutions by far. Second, the operation is inherently non-incremental: Also for the unchanged part of the input models – which should make up the major part of elements for realistic examples – pattern matches must be collected and interwoven with the global optimisation problem. For synchronising multiple models, however, incremental mechanisms provide substantial gains in performance [GW06, LAF+17].

[2] https://docs.google.com/spreadsheets/d/1BtB8EV2lGLAH3r5jigWjkRUQrCwS9wMrxdOCYb3knUA

With a second experiment, we determine the "price of fault-tolerance" by comparing the two implementations for concurrent model synchronisation in eMoflon: The approach of Fritsche et al. [FKM+20] is implemented in eMoflon::IBeX (Sect. 9.4), and can be regarded as a concurrent synchronisation approach with focus on efficiency instead of fault-tolerance. With a common example TGG, the runtime characteristics of the two implementations are investigated. The second experiment shall be guided by the following two research questions:

RQ3 How scalable is the hybrid approach to concurrent model synchronisation compared to greedy and fault-intolerant strategies?

RQ4 Should the comparably weak scalability results of the previous experiment be rather attributed to the handling of inconsistent inputs, or to the inherent complexity of the consistency management operation?

Setup: For comparing the two operations, an extended version of the *JavaToDoc* TGG with a larger metamodel and an extended rule set was used (cf. [Fri21]). Input models were created with the model generator of eMoflon::IBeX and imported into eMoflon::Neo, such that both operations work with the same example instances. The experiment was run on a workstation with an AMD Ryzen 9 3900X processor (4.6 GHz), 64 GB main memory, and Windows 10 64-bit as operating system. Similar to the first experiment, measurements were taken for increasing models sizes and an increasing number of conflicting changes, whereby the respective other measure was kept constant. To minimise the effect of outliers, the median of 5 repeated test runs was taken for each configuration. Installations of Eclipse Modelling Tools (for IBeX) and Eclipse IDE for Java and DSL Developers (for Neo) were used, both in version 2021-09 with JDK version 15.

Results: In the following, a summary of the results reported by Fritsche [Fri21] is presented, focussing on the concurrent synchronisation operation. A more detailed overview of the measurements is available online[3]. Figure 7.15 shows the runtime measurements for increasing model sizes and both operations. It becomes apparent that the runtime required by the fault-tolerant operation grows super-linear with the model size (in accordance with the results of the first experiment shown in Fig. 7.13), whereas for the greedy operation, a linear growth can be observed. The steep increase towards the last value can be explained with an observed main memory shortage. Apparently, the absolute differences with respect to scalability are enormous and cannot be explained solely with the underlying technology (cf. Sect. 9.6).

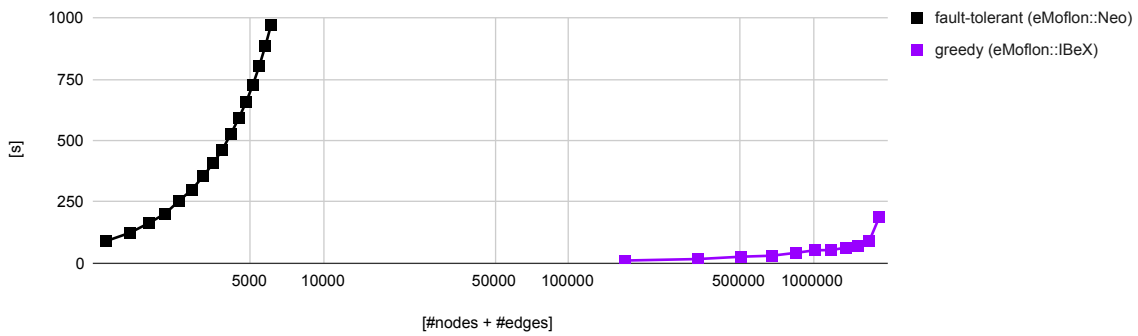


Figure 7.15: Tool comparison for increasing model sizes

[3] https://docs.google.com/spreadsheets/d/1-I1b9B6x5bNlnPDokKxemCPMskQbLOIsjbVn_z1JPZk

In Fig. 7.16, the runtime measurements for increasing numbers of conflicting changes are depicted, whereby the model size was kept constant. Similar to the results shown in Fig. 7.15, large absolute differences can be observed for the two implementations. The fault-tolerant operation seems to scale better with an increasing density of changes, though: In accordance with the first experiment (cf. Fig. 7.14), the required runtime is almost independent of the number of changes, whereas again a linear growth is observed for the greedy operation.

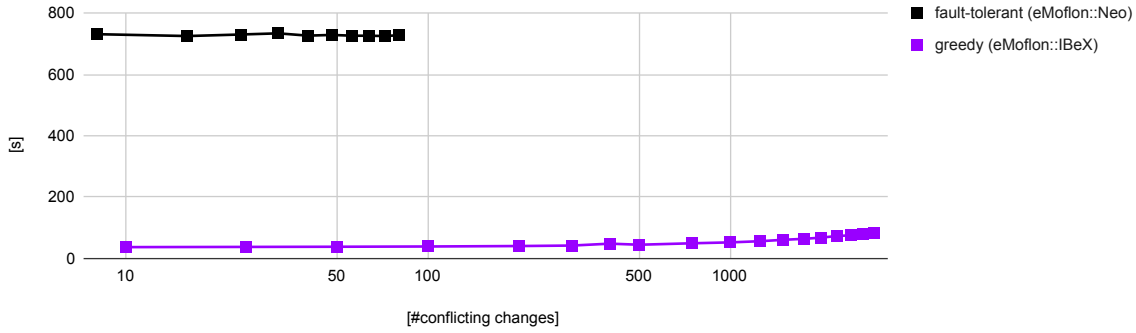


Figure 7.16: Tool comparison for increasing number of conflicts

Summary: In total, the evaluation has shown that the time requirements depend on the model sizes rather than on the number of conflicting changes [**RQ1**]. By far, most time is needed for the ILP solving step, whereas pattern matching and especially ILP construction have only minor effects on the performance [**RQ2**]. Compared to an implementation of a greedy and fault-intolerant synchroniser, large differences with respect to the measured absolute runtime can be observed. In contrast to the fault-tolerant approach, a linear growth can be observed both for increasing model sizes and a growing number of conflicts [**RQ3**]. From the second experiment, we can conclude that both the handling of inconsistent inputs and the lack of incrementality have a substantial influence on the scalability of the hybrid approach to concurrent model synchronisation [**RQ4**].

To address these issues, the runtime performance can be possibly improved by using (meta-)heuristics instead of ILP to determine the solution. For the operations presented in Chap. 5 and 6, it was important to always get an optimal result in order to guarantee completeness (cf. Sect. 6.5). Due to other necessary restrictions for the implementation, this requirement has been dropped for the concurrent synchronisation operation already, such that a speed-up at the expense of minor quality losses seems reasonable. The idea to speed-up the process by using a different search strategy is pursued in Chap. 8.

Threats to Validity: Although the example TGG arguably bears some resemblance to a practical application scenario, both models and changes are randomly generated and therefore not directly comparable to realistic synchronisation problems. As only two TGGs were considered, which are structurally similar and have a rather small rule set, further tests with TGGs of realistic sizes are necessary to underpin the results. Only a few selected modifications were performed on the models, whereas violations of domain constraints or the loss of metamodel conformance were not taken into account.

Also, only one fixed combination of configuration parameter values for rating rule applications was used. It is unlikely, though, that the runtime performance is affected by this choice, as only the objective function is concerned. While the approach is claimed to be tolerant towards faulty input models, this aspect was not investigated by this evaluation, as the generated models had been consistent before conflicting changes were introduced.

We observed a high variance for time measurements in the pattern matching step, such that even more repetitions per test case might be necessary to further reduce the effect of outliers. In the second experiment addressing RQ3 and RQ4, two implementations in two separate tools were used for the comparison of fault-tolerant an greedy strategies. The results could therefore be biased by technological differences, which should not be the only explanation for the measurement results, as further experiments will show in the course of this thesis (cf. Sect. 9.6). Finally, the evaluation is restricted to runtime performance, whereas the quality of the solution plays also an important role in practice. It is unclear, though, which measure could be used to compare the solution quality of the two implementations.

7.8 Summary and Discussion

In this chapter, we extended the hybrid approach based on TGGs and ILP to concurrent model synchronisation. By representing user edits as labelled sub-structure of the input models, the approach is tolerant towards inconsistencies in both the input models and the user edits. Users can adapt the synchronisation strategy by assigning case-specific values to configuration parameters. The synchronisation of updates on a single model, which is a well-researched consistency management problem already, can further be regarded as a special case of concurrent synchronisation, for which we offer a fault-tolerant solution strategy.

Although performance tests have shown that concurrent changes on small and medium-sized models can be synchronised, the runtime performance clearly leaves room for improvement. It seems to be promising to use heuristics to solve the optimisation problem, as ILP as an exact method turned out to be the performance bottleneck. Using multi-objective optimisation algorithms would simultaneously circumvent the problem of finding suitable configuration parameters, as separate optimisation goals can be specified for the unchanged elements and for the create, delete, and induced delta. When sticking to the single-objective problem specification, the evaluation of suitable parameter values is also an open issue. All these points are addressed in the upcoming Chap. 8.

Furthermore, a proper visualisation of delta structures and of rejected changes could be necessary to increase the tool's trustworthiness, for which a tool implementation is presented as part of the VICToRy debugger in Chap. 10.

8 Concurrent Model Synchronisation with Multiple Objectives

To overcome the scalability problems which are induced by using exact algorithms for concurrent model synchronisation, we apply meta-heuristics instead of ILP for the optimisation step in this chapter. As the requirement of optimality is relatively weak for this operation, a slight decrease of the solution quality seems to be acceptable to improve the efficiency of the implementation. Furthermore, for determining suitable configuration parameter values, two alternatives are proposed: Via an empirical case study, a method for parameter determining is proposed for a specific use case, which can be transferred to other application scenarios as well. The problem can also be circumvented by using multi-objective optimisation strategies, resulting in a wide range of optimal solutions, though. Runtime performance tests have indeed shown that - due to the small portion of feasible solutions in the search space - ILP still outperforms all other heuristics in this scenario.

This chapter is structured as follows: The use of heuristics for concurrent model synchronisation is briefly motivated in Sect. 8.1. After providing an overview of related work (Sect. 8.2), Sect. 8.3 shows how the optimisation step can be performed using heuristic search algorithms. The adaption of their operators to the problem domain is presented in Sect. 8.4. The results of applying our approach to a case study is described in Sect. 8.5, as well as an experimental evaluation that compares its runtime performance and result quality to the baseline of ILP solving. A summary and ideas for subsequent research are sketched in Sect. 8.6. Finally, a the hybrid approach to consistency management as a whole is wrapped up in Sect. 8.7.

8.1 Motivation

In Chap. 7, we have seen that the hybrid approach combining TGGs and ILP is indeed extensible to concurrent synchronisation scenarios. The scalability analysis, however, has shown that for concurrent synchronisation - in contrast to the previously introduced operations - the ILP solving step can be performance-critical as well. Another difference is that optimality is not absolutely necessary for concurrent synchronisation: While for the other operations, the optimal result is a precondition for deciding whether the input models are consistent, this aspect is rather unimportant for concurrent synchronisation.

Furthermore, the determination of suitable configuration parameters is still an open issue, as long as the optimisation is based on a single function. The objective function according to Def. 7.4 is composed of four parts which balance conflicting goals, such as preserving deletions and creations, keeping unchanged model parts as they are, and reducing the amount of automated changes. The four goal functions are weighted with configuration parameters, such that the optimality of a solution heavily depends on their choice. The values were arbitrarily chosen for demonstration purposes, such that the determination of weighting parameters that lead to a satisfactory synchronisation solution is still an unsolved problem.

It seems reasonable to replace the exact method of ILP solving to improve the scalability of the approach by using a heuristic that only finds a near-optimal solution but shows a

much better runtime performance. As an alternative, we propose a heuristic approach to concurrent model synchronisation that alters the hybrid strategy of Chap. 7 by using meta-heuristics that were successfully applied to model merging[1] already [KWLW13, DRV+16, BFT+19].

Several algorithms enable the definition of multiple objective functions. This can be beneficial to model the concurrent synchronisation problem: To balance the previously mentioned optimisation goals, four objective functions can be defined. The set of pareto-optimal solutions is presented to the user, from which he or she can pick a single solution that satisfies their expectations best.

It is to be expected, however, that the set of solutions will be too large even for medium-sized models to let a user choose the final solution from it. Therefore, suitable weighting parameters are determined in an empirical case study to form a scalarised single objective. Alternatively, with a sufficiently large amount of training data, machine learning techniques could be applied to determine these parameters, which is not the case here, though. For single-objective optimisation heuristics, the problem is specified just as for the ILP-based approach, but the solution strategy is fundamentally different.

Finally, these insights are used to recommend a strategy for determining solutions of satisfying quality within an acceptable amount of time. We compare the runtime performance and the achieved quality to the ILP-based approach using models of different sizes.

8.2 Related Work

Several heuristics were recently applied to address model management problems. From the group of evolutionary algorithms, GA [Hol92] and NSGA-II [DAPM02] are most commonly used for single- and multi-objective optimisation, respectively. Kessentini et al. [KWLW13] proposed an approach for merging different model versions based on GA, maximising the number of edit operations to create the merged model. Similarly, Assuncao et al. [AVLH17] used GA to merge UML models with the goal of minimising the difference between the input models and the merged model. Taking the importance of edit operations into account as a second objective, Mansoor et al. [MKL+15] performed model merging using NSGA-II. When analysing the results of a model transformation, the applied steps are of particular interest. Based on a list of possible steps, which can be fine-grained (e.g., graph edits) or high-level (e.g., refactorings), a search space of application sequences can be constructed, which is usually too large for performing an exhaustive search. Both GA [bFKLW12], NSGA-II [SHNS13] and its predecessor NSGA [AVS+14] were used as meta-heuristics for efficiently exploring this search space.

Local search strategies start with a (random) initial solution and iteratively improve it until some termination condition is reached. Debreceni et al. [DRV+16] use guided local search for merging different model versions, while conflicts emerge as constraint violations. Besides this, DSE was used to determine optimal merging results, which was also applied to other problem domains [HKC+14, DDGV16]. Good runtime performance results were received on benchmark instances of 50,000 model elements. Dam et al. [DEW+16, DRE14] propose a local search algorithm for merging uncontroversial changes in different model versions and detect conflicts, which are subsequently presented to the user. SA [KJV83] is a local search algorithm that accepts worse intermediate solutions with some probability to

[1] Model merging denotes the task of synchronising concurrent changes on the *same* model. Similar to text-based synchronisers such as GIT, model mergers create a single solution that incorporates as many edits as possible.

escape local optima. This strategy is used by Kessentini et al. [KSB08,KBSB10,KSBB12] in combination with particle swarm optimisation for "model transformation by example", showing promising results for small model instances.

In summary, both evolutionary algorithms and different local search strategies were successfully applied in model merging and model transformation scenarios. In a comparative study, Bill et al. [BFT+19] applied different meta-heuristics to model transformation, concluding that SA outperforms other local search algorithms for larger models. Compared to GA, SA shows better results for search spaces with a large infeasible region, whereas the opposite is the case when most solutions of the search space are valid. As the hybrid approach of combining TGGs and optimisation techniques provides the potential of exchanging the optimisation algorithm without larger efforts, we will continue with this idea to analyse the potential of GA, NSGA-II and SA for concurrent model synchronisation.

8.3 Solution Overview

In this section, the solution approach, i. e., the modification of the problem definition and the adaption of meta-heuristics for efficiently retrieving appropriate solutions is discussed in more detail. First, the foundations of bio-inspired meta-heuristics are briefly introduced, such that the adaptation of the respective steps in the context of concurrent synchronisation can be described afterwards. As a result of studying related approaches in Sect. 8.2, we use GA and SA for single- and NSGA-II for multi-objective optimisation.

- **SA** is a single-objective optimisation heuristic. It uses local search strategies to iteratively approach an optimal solution, but temporarily permits worse solutions, such that an escape from local optima is possible. The name originates from annealing processes as part of, e. g., metal component production: Before the crystals solidify completely, they can rearrange themselves to form a stable state. With progressing time, the temperature of the crystals decreases, such that the movement is more and more restricted. Similarly, the SA algorithm uses a decreasing *threshold* (the temperature), up to which worse solutions can be accepted.

- **GA** is also a single-objective optimisation heuristic that is inspired by evolutionary processes in biology. In multiple iterations, a population of individuals is formed out of the last generation's population by means of mutation, recombination and selection. According to the Darwinian theory of evolution, only the fittest individuals survive over time, as fitter individuals are more likely used for recombination and selection.

- An advancement of GA is the multi-objective **NSGA-II** algorithm. The basic concept of resembling evolutionary processes is shared between GA and NSGA-II, but individuals can be rated according to multiple fitness functions. Consequently, the result of the optimisation process is not only a single "fittest" individual, but a potentially large set of individuals, denoted as *pareto front*. It consists of all individuals that are not *dominated* by another individual, i. e., there is no other individual that has an equal or higher fitness value for each objective.

All three heuristics share a common generic work-flow, which is depicted in Fig. 8.1 in form of a UML activity diagram. This work-flow describes the optimisation step (D) of Fig. 7.2 more concretely, which was previously done by means of ILP solving.

The first step is to find a suitable encoding (1) for solution candidates, i. e., to translate the *phenotype* into a *genotype*. In the concrete application scenario, the sets of selected

and unselected rule application candidates make up the phenotype. The genotype, in contrast, is a more compact representation of these two sets, e. g., in form of a bit string.

The optimisation process itself repeats steps (2) – (5) until a termination criterion is reached. This criterion is the fundamental difference between exact methods, such as ILP, and meta-heuristics: While exact methods can only stop as soon as the optimum solution is found, heuristics terminate after a predefined number of iterations or other conditions that are largely independent of the solution quality. In this main loop, individuals of a population *n+1* are generated out of the population *n* using different *genetic operators*. When the termination criterion is reached, the main loop terminates, and the best solution found so far is decoded (6) into its phenotype.

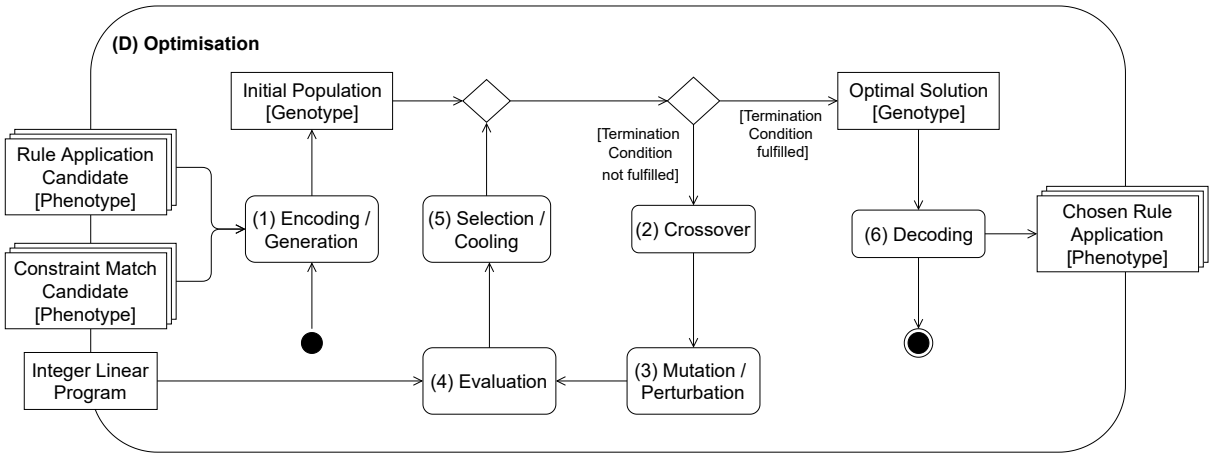


Figure 8.1: Generic work-flow of heuristic optimisation

With respect to the number of optimal solutions, there is a difference between single- and multi-objective optimisation: The optimal solutions of problems with only one objective function are unique up to tie-breaking decisions, i. e., the solution that maximises the objective function can be considered as optimal. For problems with multiple objectives, each solution is considered as *pareto-optimal*, for which there is no other solution whose objective function values are at least as good.

In the following, we will present how solution candidates can be encoded, evaluated and varied throughout the search process for the three chosen meta-heuristics. The encoding of solution candidates and operators can be kept mostly similar for all algorithms, differences are pointed out explicitly. To make all explanations more concrete, and for being able to compare the heuristic and the exact approach, the running example of Sect. 7.6 is reused. Before that, we start with a short comparison of single- and multi-objective optimisation to argue that both techniques are appropriate to model the application scenario of concurrent model synchronisation.

Single- and Multi-Objective Optimisation

While the search-space can be constructed in accordance with the ILP-based approach of Chap. 7, we adapt both single- and multi-objective optimisation heuristics to the problem domain to analyse whether there is a reasonable trade-off between solution quality and runtime performance. The different optimisation objectives become apparent when revisiting the definition of the parametrised objective function (Def. 7.4): Ideally, all changes applied by one of the involved domain experts should be preserved, such that the synchronised solution consists (only) of all elements which were recently added or remained unchanged

since the last synchronisation took place, whereas none of the deleted elements should be kept in the result. This, however, is hardly achievable when all correctness constraints have to be respected as well. Based on the partitioning of the model elements into four sets (cf. Def. 7.3), one can define four optimisation objectives, that maximise (unchanged elements, create delta) or minimise (delete delta, induced delta) the occurrences of such elements in the final solution:

- **Unchanged elements**: Elements that are not affected by any change should be preserved, if possible.
- **Create delta**: New elements in one model should be kept with high priority, which makes it necessary to propagate these changes to the other model as well.
- **Delete delta**: Elements that are deleted in one model are in the first instance marked to be deleted. These markings should turn into actual deletions after synchronisation.
- **Induced delta**: New model elements require the creation of further elements in the other model, but their number should be as small as possible.

Despite the fact that the problem is inherently multi-dimensional, an aggregation into a single objective should be considered as an efficient and simple alternative to compute sufficiently good solutions. It is crucial, however, to use appropriate weights for the scalarisation, as large parts of the pareto front are omitted[2], depending on the problem size and the number of objectives. This aspect was priorly circumvented by leaving the determination of weighting parameters to future work. We therefore propose weightings as part of our evaluation (cf. Sect. 8.5).

8.4 Adaptation of Meta-Heuristics

In order to apply meta-heuristics to a concrete problem domain such as concurrent model synchronisation, the single steps of the generic work-flow must be adapted to this domain, which will be presented in the remainder of this section. For illustration purposes, the example instance of the previous chapter (cf. Fig. 7.12 / Tab. 7.1) is used.

(1) Encoding

Solution candidates must provide information about which rule applications contribute to the synchronisation solution. Therefore, it seems suitable to encode them as a binary string, in which each bit is associated to one rule application candidate. The bit is set to 1 iff the rule application is selected for the synchronisation solution. The initialisation of the population for GA and NSGA-II is straightforward as random binary strings are generated. This would also be possible to initialise SA, which only operates on a single solution in each step. However, as it is beneficial to start with a fairly good solution, the SA is initialised with a string of zeros, which represents the empty triple graph because no rule application is selected. Although the solution is usually far below the optimum, it is guaranteed not to violate any constraint by definition and is therefore part of the feasible region. The suggested solution for the example instance would be encoded as 1111011001001.

[2] cf. Coello Coello et al. [CLvV07] for an in-depth discussion of scalarising multi-objective optimisation problems

(2) Crossover

With the crossover operator, promising genetic features of parent individuals shall be recombined to generate offspring with an even higher fitness value. For GA and NSGA-II, we use single-point crossover, which splits the bit strings A and B representing the parent individuals at a random position into two parts A'/A'' and B'/B''. The offspring is then generated by concatenating A'/B'' and B'/A'', respectively. SA, in contrast, does not have such an operator as only one current solution exists at each point of time. Suppose that the example solution (Fig. 7.12 / Tab. 7.1) is recombined with another solution that propagates the change on the SysML model to the Event-B model, i. e., connects the transition `t2` with the `STOP` state `s2` (encoded as 1111011110001). When splitting after the ninth bit, `t2` has two targets in the one individual, and no target at all in the second, leading to solutions that violate the graph constraints. In this case, the crossover does not generate fitter offspring, which is likely in case the parent individuals have a high fitness value already.

(3) Mutation and Perturbation

The mutation operator is a crucial factor for genetic algorithms to add diversity to the population. A similar concept for SA is perturbation, where neighbouring solutions are generated in each iteration by modifying the previous solution. As solution candidates are encoded as binary strings, it is advisable to use bit flips at random positions of the genotype. For the phenotype, mutation and perturbation add or remove rule applications from the solution candidate. The probability for each bit to be flipped is set to $1/N$, whereby N denotes the number of bits, such that the expected number of bit flips is 1 per individual. Regarding the example instance, a possible outcome would be to flip the second bit from 1 to 0. As a result, d_2 is removed from the solution, such that the port `p` and its corresponding variable `v3` and invariant `i3` are not part of the solution.

(4) Evaluation

The quality of solution candidates is assessed using a fitness function, which also serves as a basis for selecting individuals for recombination. While NSGA-II allows arbitrarily many objective functions, these have to be scalarised to a single function for GA and SA. For multi-objective optimisation algorithms, such as NSGA-II, we define four objectives that resemble the optimisation goals listed in Sect. 8.3. The optimisation problem (cf. Def. 7.4) can be redefined as follows:

Definition 8.1 (Multi-Objective Optimisation Problem)**.**
Given a starting triple graph G_0, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules. The optimisation problem is constructed as follows:

$$
\begin{aligned}
&\textit{max. } \Big(\sum_{d \in D} |mrk(d) \cap unchanged(G_n)| \cdot \delta\Big) \\
&\textit{min. } \Big(\sum_{d \in D} |mrk(d) \cap dltDelta(G_n)| \cdot \delta\Big) \\
&\textit{max. } \Big(\sum_{d \in D} |mrk(d) \cap crtDelta(G_n)| \cdot \delta\Big) \\
&\textit{min. } \Big(\sum_{d \in D} |mrk(d) \cap indDelta(G_n)| \cdot \delta\Big) \textit{ s.t.} \\
&\textit{markedAtMostOnce}(G_0) \wedge \textit{context}(D) \wedge \textit{context}(G_n) \wedge \textit{acyclic}(D) \wedge \textit{sat}(G)
\end{aligned}
$$

For GA and SA, these four functions must be aggregated into a single function with suitable weightings, choosing negative weighting parameters for the minimisation objectives, such that the resulting weighted sum can be maximised. Following the notation of the ILP-based approach, we weight the delete delta with $\alpha < 0$, the create delta with $\beta > 0$ and the induced delta with $\gamma < 0$, such that the optimisation problem can again be constructed according to Def. 7.4. The function for the unchanged part of the model serves as norm with weight 1. Suitable values for α, β and γ are investigated on in the second part of the evaluation (Sect. 8.5).

Treatment of the Infeasible Region

The chosen form of encoding solution candidates adds a noticeably large set of invalid solutions to the search space, i. e., solutions that violate at least one of the correctness constraints. The lack of a rule application which is implied by others immediately leads to invalid sequences, which violates at least one constraint (Tab. 7.1, columns 4–6) and is therefore located in the infeasible region. In turn, a superfluous rule application might translate elements twice, which is also prohibited by constraints. In the running example, for instance, each solution candidate ending with 101 or 011 would violate the constraint that mutually excludes d_{11} and d_{13}, or d_{12} and d_{13}, respectively. Instead of rejecting invalid solutions immediately, a penalty is introduced to the objective function that guarantees that such solutions cannot have a higher fitness than the empty triple graph, which is always valid. Similar to using mutation or perturbation, this technique is beneficial for escaping local optima [KBB+16].

(5) Selection and Cooling

For the genetic algorithms GA and NSGA-II, the computed fitness of individuals is used to decide which of them are selected to enter the next generation. We use binary tournament selection, i. e., two individuals are randomly and repeatedly picked from the population and the one with the better fitness is selected. In SA, in contrast, a "temperature", which is continuously decreased, defines the probability for accepting worse solutions during the iterative search. This leads to a compromise between exploring the search space in the beginning and exploiting it towards the end. For a concrete software implementation, several variants of this meta-heuristic are possible. We followed the proposition of Bill et al. [BFT+19], in which neighbours are generated out of a working solution that is reset to the best solution found when no improvement could be achieved for a pre-defined number of iterations.

(6) Decoding

After the termination criterion has been reached, the individual(s) with the best fitness function value found so far are taken and decoded into their phenotype form, which is the final result of the optimisation process. The decoding operation is the inverse of the encoding operation which was presented in step (1).

8.5 Evaluation

To improve the applicability of our approach for realistic use cases, the concurrent synchronisation implementation was extended with an alternative way of constructing and

solving the optimisation problem. Considering the generic work-flow of Fig. 8.1, an implementation based on the three discussed meta-heuristics (GA, NSGA-II and SA) was added, replacing the ILP solver for concurrent model synchronisation. The comparison of the two implementations shall show to which extent heuristics are beneficial for search-based consistency management. In particular, the following three research questions shall be answered by our evaluation:

RQ1 Is it feasible in practice to present the set of pareto-optimal solutions to users, such that they can choose one of them to synchronise concurrent updates?

RQ2 How can suitable parameters α, β and γ be determined to scalarise multiple objectives for a concrete use case?

RQ3 To which extent are heuristic solutions applicable in practice regarding runtime behaviour and solution quality?

Case Study: The evaluation was conducted with a simplified version of the *JavaToDoc* TGG (cf. Sect. A.2) without graph constraints. The triple metamodel is depicted in Fig. 8.2. On the left-hand side, it is shown that a Java model consists of `Classes` that can form an inheritance hierarchy via the recursive subtypes relation. `Classes` can have arbitrarily many `Fields` (i. e., attributes) and `Methods`. All entities are (non-uniquely) identified by a name attribute of type `String`. On the right-hand side, the documentation metamodel shows that `Documents` consist of a (possibly empty) list of `Entries` and can refer to other `Documents` (via the `href` relation). The correspondence nodes connect Java `Classes`, to `Documents` of the documentation model, and both Java `Methods` and `Fields` to `Entries`.

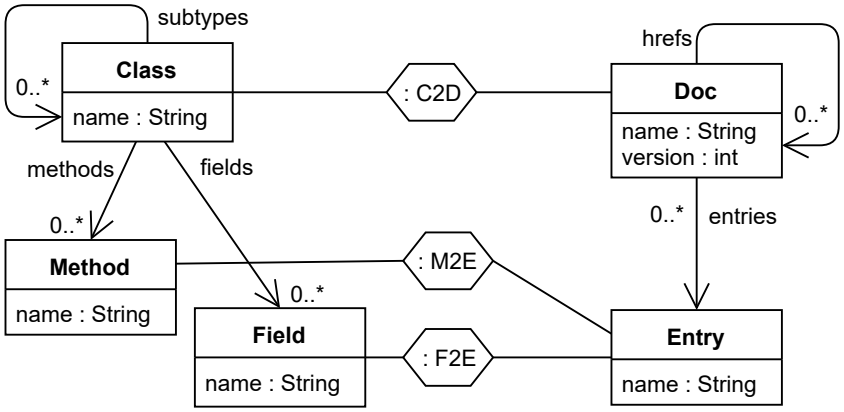


Figure 8.2: Simplified metamodel for JavaToDoc

A small example instance in concrete syntax is depicted in Fig. 8.3. It consists of two classes for geometric figures - `Rectangle` and `Parallelogram` - and some basic methods and attributes that characterise their side lengths, areas and extents. When the last synchronisation took place, both `Parallelogram` and `Rectangle` had attributes `sideA` and `sideB` for the length of their respective sides. For the class `Rectangle`, a method `getArea` was provided. The subclass `Diamond` extended `Parallelogram` with a method `getExtent`. Two conflicting refactorings take place now to improve the software system. In the Java code, an inheritance relation between `Rectangle` and `Parallelogram` is introduced, such that the duplicate attributes `sideA` and `sideB`

can be deleted. For being able to compute the area of the `Rectangle`, an attribute `height` is added to the `Parallelogram`. In the documentation, in contrast, the developer decides to delete the class `Parallelogram`, as it appears to be an unnecessary generalisation of the `Rectangle`. The class `Diamond`, which previously inherited the side length attributes, is in turn equipped with a new attribute `sideA`, such that the method `getExtent` can be kept.

Apparently, these changes are partly in conflict with each other: On the one hand, the `Parallelogram` cannot be deleted and modified at the same time. On the other hand, assuming that `Parallelogram` shall be kept, the attribute `sideA` is present both in `Diamond` and `Parallelogram`, which is not allowed in most programming languages. In the course of this case study, different refactorings were combined to produce conflicts in structured way, which is described in more detail when answering RQ1. The goal behind choosing this example case study is two-fold: First, the example instance must be large enough to produce different conflicts by applying realistic changes to the respective models. Second, the instance should also be as small as possible to keep the number of pareto-optimal solutions manageable. The solutions should be easy to oversee, such that differences can be spotted quickly. Third, the problem domain (code and its documentation) must be comprehensible for undergraduate students, such that participating in the empirical study (cf. RQ2) should be possible for them without additional info material on the problem domain.

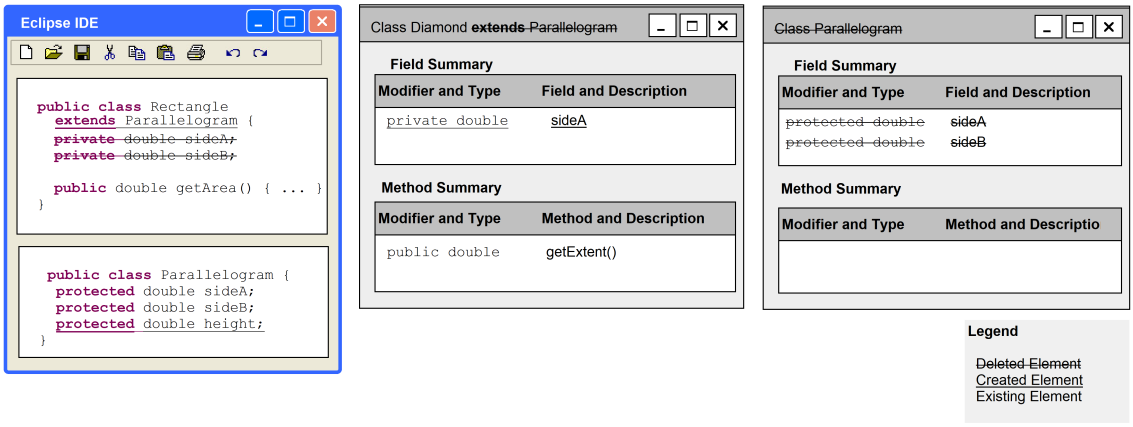


Figure 8.3: Conflicting changes on Java code and documentation

Setup and Parametrisation: Our approach was implemented as an extension of the concurrent synchronisation implementation of eMoflon::Neo (Sect. 9.5). GA and NSGA-II were integrated into the tool via the MOEA Framework[3], whereas SA was implemented manually. We followed the SA implementation proposed by Bill et al. [BFT+19] parametrised with $c_c = 3$ and $c_b = 10$. For a comparison to the baseline approach, the Gurobi optimiser (Version 8.1.1) was used for ILP solving. Neo4j (Version 4.2.1) was used as a graph database for storing (meta-)models. All performance tests were executed on a standard notebook with an Intel Core i7 (1.80 GHz), 16GB RAM, and Windows 10 64-bit as operating system. An installation of Eclipse IDE for Java and DSL Developers, version 2019-09 with JDK version 13 was used. 4GB memory each were allocated to the JVM running the tests and to the graph database Neo4j. For each configuration, only the time required for the optimisation step was measured, because other tasks such as graph pattern matching are independent of the optimisation algorithm. Each test run was repeated

[3] http://moeaframework.org/

30 times to minimise the bias introduced by outliers, the median was taken as final value. An overview of the evaluation results was already given in prior work [WE21]. While a detailed overview of the examples and collected data is available online[4], a summary of the results is presented in the remainder of this section.

RQ1 (Analysis of the pareto front): We set up small case study in the setting of the *JavaToDoc* TGG. Independent of the concrete syntax of source and target models, an overview of the example is provided in Fig. 8.4 in form of a UML class diagram. For four geometric figures, it shall be possible to compute their area and extent out of their attributes. The initial model shall therefore be improved using five refactorings, which are partly in conflict with each other. Elements which are deleted or created via a refactoring are depicted in red (−−) and green (++), respectively, all other elements remain unchanged. Furthermore, the elements are annotated with the number of the refactoring which modifies them. Possible refactorings are the introduction of an inheritance relation between `Square` and `Parallelogram` (1), `Square` and `Diamond` (2), the enrichment of `Diamond` with further attributes (3), a new inheritance relation between `Rectangle` and `Parallelogram` (4) and the deletion of the class `Parallelogram` (5). In the example of Fig. 8.3, refactoring (4) was applied on the Java model, and refactoring (5) on the documentation model.

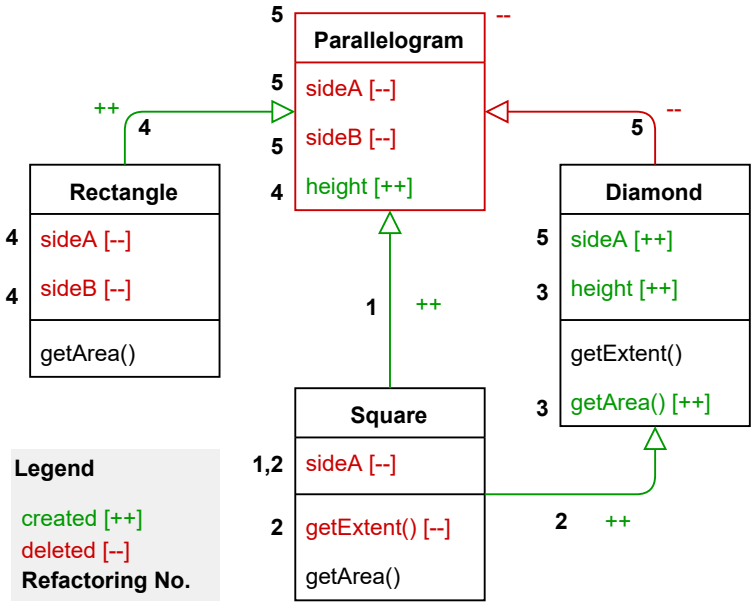


Figure 8.4: Case study: Conflicting changes for geometric figures

In the case study, each subset of three refactorings was applied on the initial instance (two on the Java model, one on the documentation model), such that at least one conflict was induced. The refactorings were distributed such that those two on the Java model are not conflicting. For the resulting ten variants, the set of pareto-optimal solutions was computed using NSGA-II with one million iterations each. Regarding the *number of retrieved solutions*, the median for all test runs and variants (i.e. refactoring combinations) was 13, reaching from 10 to 33 for single variants. The mean was 16.323, reaching from 8.900 to 33.333, and the standard deviation 9.806, reaching from 1.661 to 8.768. The median runtime for all test runs and variants was 32.047 s, the mean was 32.375 s and the standard deviation 2.733 s.

[4] https://drive.google.com/file/d/1elGTTplm6RKm8IQaiNzZyP57V8Rrj_90/view

The experiment reveals some limitations for working with a multi-objective approach in practice. Without further support, it is not feasible for a user to choose from the complete set of pareto-optimal solutions. Although the setting of the case study is not generalisable to arbitrary synchronisation problems, it is very likely that for realistic model sizes, the set of pareto-optimal solutions is substantially larger. Furthermore, the number of evaluations was chosen with the goal of being sufficiently high for retrieving the whole set of pareto-optimal solutions in each test run, leading to an average runtime of about half a minute for solving a single synchronisation problem, which is not acceptable for practical applications. However, there was a high variance in the size and composition of the retrieved solution sets, indicating that substantially more resources would be necessary to find the whole set in a single run. As a result, we formed a union of all pareto-optimal solution sets for each variant and continued with these sets to answer RQ2. After removing duplicates and solutions which are dominated in the union of all sets for a variant, 451 solutions were determined, i.e. 45.1 per variant (median: 52).

RQ2 (Parameter determination): As the experiment to answer RQ1 shows that it is not feasible to let the user choose from all pareto-optimal solutions even for small model sizes, a single objective function must be constructed to make the approach both efficient and applicable in practice. However, it is hardly possible to determine generally valid parameter values for aggregating the four involved objectives into a single function, as their choice depends on the underlying TGG. If, for example, the involved models are extensively nested, a higher absolute value for α is required as elements on top of such a hierarchical structure can never be deleted otherwise, in contrast to rather flat model structures for which an α value closer to 0 might be sufficient. Therefore, we restrict ourselves to an exemplary determination of parameter values for the example case study of Fig. 8.4, and simultaneously propose a method to find suitable values for other use cases as well.

It is furthermore not possible to let users estimate suitable values for those parameters directly, because their meaning is very abstract and even model transformation experts lack an intuition for the consequences of particular parameter choices. Therefore, we follow an indirect approach of empirically assessing the pareto-optimal solutions for the case study of RQ1. 9 undergraduate students with sound programming skills and basic knowledge on MDE were asked to rate the quality of different pareto-optimal solutions on a scale reaching from 0 (no agreement) to 10 (total agreement). The rating shall be based on to which extent the synchronisation solution reflects the intention behind the applied refactorings. As the entire solution set is even too large for this study, scalarisations for different value combinations were computed, and the solution set was restricted to those candidates which are optimal for at least one scalarisation. The choice was made in accordance with the value range defined in Sect. 8.3 ($\alpha < 0, \beta > 0, \gamma < 0$), and the following preliminary considerations were taken into account:

- α should be at most -1, such that for each deleted element in one model, it is possible to delete an element in the other model without decreasing the objective function value.
- β should be at least 1, as it is at least as important to preserve new elements as to preserve unchanged elements.
- γ should be $\in [-1; 0)$, because it should serve as a tie breaker in cases where it is possible to allocate created elements to each other instead of translating them anew (cf. Sect. 7.5). For each new element, it should be possible to add an element in the respective other model without decreasing the objective function value.

We observed that further decreasing α beyond -4 or increasing β or γ beyond 4 and -0.6, respectively, in isolation does not influence which of the pareto-optimal solutions is optimal for the scalarisation. Therefore, the chosen value ranges were $\alpha \in \{-1, -2, -3, -4\}$, $\beta \in \{1, 2, 3, 4\}$ and $\gamma \in \{-0.6, -0.8, -1\}$, resulting in 48 different scalarisations. The mean[5] participant rating was allocated to the parameter combination, for which the respective solution is optimal. The average rating of all 10 variants was taken as rating for the respective parameter combination.

The results are visually depicted in Fig. 8.5, where a cuboid represents the investigated parameter value range and its colouring the average rating of the respective parameter combination. The observed upper and lower bounds correspond to pure blue (9) and pure red (4), while values in between are represented by a proportional mix of these two colours. The front faces in the left/right diagram correspond to ratings for $\gamma = 0.6/1.0$. One can observe that choosing γ closer to 0 leads to better ratings, only a combination with α values close to 1 seem to be problematic. This problem gets more severe when choosing $\gamma = 1$, here only combinations with $\alpha < -2$ and $\beta > 2$ lead to good ratings. Interestingly, the choice of β seems to be more important for combinations with γ close to 1, which indicates that the ratio between β and γ has to be "large enough". This seems reasonable due to the interplay of propagating user additions and avoiding unnecessary model extensions. Similarly, propagating deletions (weighted with α) and preserving unchanged elements (weighted with 1) are especially conflicting, which means that the effect of changing α should be quite independent of the choice of β and γ. This claim is also supported by our measurements. Finally, considering the right diagram in which $\gamma = 1$ holds for the front face, it seems to be useful to keep the ratio between β and γ approximately equal to α as the diagonal from (-1/1/-1) to (-4/4/-1) cuts the front face almost into symmetric halves.

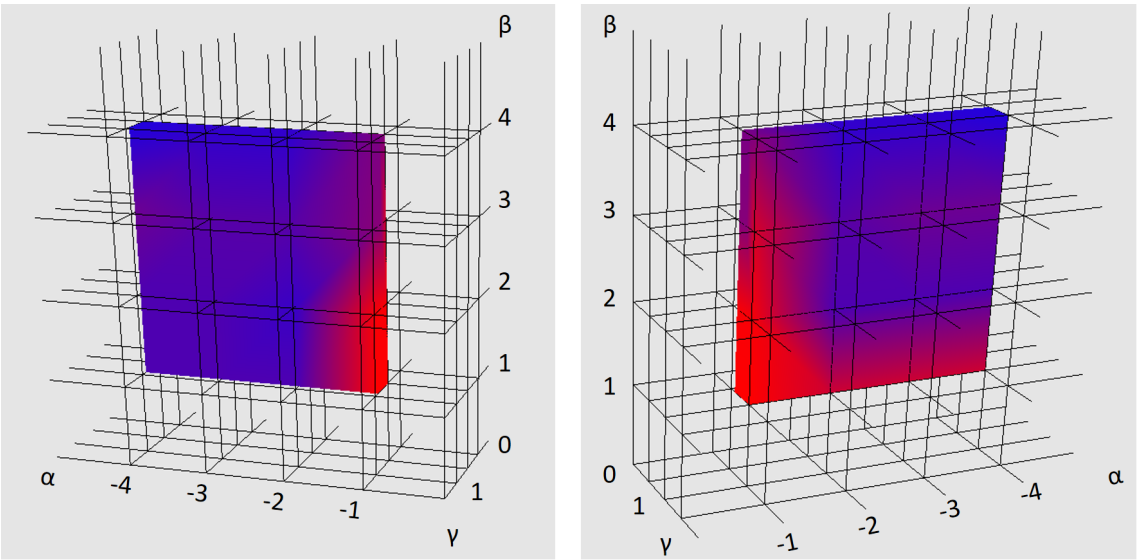


Figure 8.5: Average ratings for parameter combinations

RQ3 (Runtime and quality analysis): To analyse how the runtime performance of GA and SA compare to ILP, we ran the respective algorithms on synthetic model instances with linearly increasing sizes from 460 to 2141 elements. For simulating edit operations, create and delete markers were randomly assigned to 20% of the model elements each, resulting in a comparably high change density. In accordance with the evaluation of the

[5]Using the median instead of the mean appears to be inappropriate here as the loss of precision is more harmful than the effect of statistical outliers on a scale from 0 to 10.

ILP-based approach (Sect. 7.7), we used the *JavaToDoc* TGG as presented in Sect. A.2 (without graph constraints) to compare the different heuristics. The results are presented in an aggregated form, details are available online[6].

For GA, a population size of 100 was used. SA was run with *linear* cooling from the start temperature T_0 down to 0 and a *geometric* cooling using a factor of 0.98, with a re-heating to T_0 as soon as T falls below a threshold of 1. The measurements did not show any statistically significant difference for the cooling schemes regarding runtime or solution quality, therefore we restrict ourselves to the presentation of the geometric variant. As a stop criterion, we defined a limit for the number of iterations per test run, which is equal for all heuristics in order to guarantee comparability. The limit depends linearly on the length of the bit string, such that a larger number of iterations is granted for larger models. For SA, also test runs with only 10% of the iterations were conducted ("short runs") to analyse the effect of this criterion.

The median runtime values per model instance and algorithm are depicted in Fig. 8.6. As expected, the runtime grows slightly super-linear with the model size, because the stop criterion depends linearly on the model size and the generation of new solution candidates becomes more time-consuming for longer bit strings. For SA and GA, the observed runtime behaviour did not show any significant differences. Interestingly, the ILP baseline outperformed these two heuristics, showing better scalability characteristics for larger models, similar to the short run version of SA.

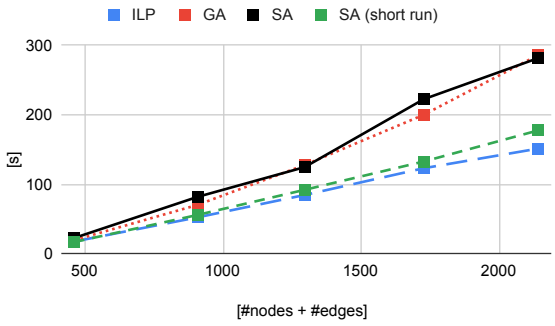


Figure 8.6: Runtime analysis

Figure 8.7: Quality analysis

To measure the solution quality, the objective function value of the returned solution was used. In comparison to the solution determined by ILP solving, which is guaranteed to be optimal, it is possible to assess how close to the optimum heuristic solutions are on average. The median objective function values are shown in Fig. 8.7. For SA, the 95% confidence interval is further displayed with dotted lines.

GA found feasible solutions only in 16 test runs for the smallest model size and in no test run for the largest two model sizes, such that the heuristic is left out in this diagram. SA, in contrast, could determine solutions with objective function values of 81.6% - 94.1% of the optimum, taking the minimum and maximum of the 95% confidence intervals of all model sizes. The short run version of SA was clearly outperformed, though. These results - especially the weak performance of GA - are unexpected in the first place, which is why we assume that the search space is infeasible for the very most part. To support this assumption, we randomly sampled solutions using the same number of iterations as for GA and SA. In none of the test runs (consisting of approx. 1.5 million iterations for the smallest and 7.5 million for the largest model), a feasible solution was found. With

[6] https://drive.google.com/file/d/1elGTTplm6RKm8IQaiNzZyP57V8Rrj_90/view

respect to this observation, it seems to be unrealistic to expect a heuristic algorithm to reliably return a near-optimal solution.

Summary: Revisiting our research questions, one can state that even for small examples, the pareto front contains too many solutions to be entirely presented to the user, such that a scalarisation to a single objective function is necessary [RQ1]. With help of an empirical user study, a method to determine suitable parameter values for α, β and γ was presented for the case study of RQ1, which is applicable to other use cases, too [RQ2]. A performance evaluation has shown that GA is not an adequate heuristic in this setting as it often fails to find even a feasible solution. SA performed much better (conforming to the observation of Bill et al. [BFT+19] for search spaces with a large infeasible region), whereas near-optimal solutions could not be determined faster than the optimal solution via ILP solving. In general, the measured solution quality of 81.6% - 94.1% for SA implies that a noticeable part of the changes is unnecessarily dropped, which will not be perceived as an acceptable solution for most users. In summary, we recommend to stick to the ILP-based approach, while the parameter analysis for answering RQ2 helps to configure the respective optimisation function.

Threats to validity: Although the approach was tested in depth for one example, results for RQ2 and RQ3 may differ when applying it on other synchronisation problems. The number of participants for determining suitable parameters was low, such that the results might be biased by personal preferences. Furthermore, the chosen use case was synthetic and only involved small models, such that the design of the user study might become infeasible for other synchronisation problems and larger models. Although there was a large performance gap between ILP, SA and GA in our experiments, it is possible that another operator implementation or parametrisation of the meta-heuristics could lead to more promising results that outperform the exact solution. As we rely on a standard framework for GA and NSGA-II and the Gurobi solver for ILP, the efficiency of the implementation in external components also has an influence on the runtime performance.

8.6 Summary and Discussion

We presented a heuristic approach to concurrent model synchronisation, with which we aim at reducing the runtime effort compared to the ILP-based version presented in Chap. 7. Multiple heuristics, which have become an integral part of Search-based Software Engineering (SBSE) research in recent years, were applied to efficiently restore consistency after concurrent user edits on different models. By adapting (genetic) operators to the problem domain, an implementation involving three single- and multi-objective optimisation heuristics is presented and integrated into eMoflon::Neo. Compared to the ILP-based approach of Chap. 7, the problem definition was extended towards multiple objectives, while the constraint generation is equal for both alternatives. The use of heuristics did not yield the desired performance improvements due to the large portion of infeasible solutions, though. Therefore, it is still recommendable to use exact solvers in this setting to obtain solutions of satisfactory quality. A generalisable methodology was proposed to configure the objective function for a concrete applications scenario.

To carry this line of research forward, we plan to analyse further use cases involving other TGGs and larger model instances to see in which scenarios recommendable parameters differ from the results of this study. In the presented version, the configuration parameters enable the user to define weights for creations and deletions, whereby the weights do not differ for source and target models. At the end of Sect. 7.6, we have seen that it is not possible to choose the parameters in a way that the change in the source model is preferred

due to the shape of the TGG. This problem could be overcome by a more fine-grained configuration, which in turn makes the determination of suitable parameter values even harder.

After generating the possible solutions for the empirical case study (Sect. 8.5, RQ2), we observed that many (technically correct) solutions do not make sense because user edits were split, although they should be treated as atomic changes. It is problematic, for example, to keep a superclass but remove all attributes and methods from it. This could be prevented by adding further atomicity constraints to the optimisation problem. Furthermore, an implementation of further meta-heuristics, which have proven to be beneficial for solution spaces with a large infeasible region, might be promising to improve the overall runtime performance. While the presented approach is fully automated, the user involvement in resolving synchronisation conflicts shall be improved, such that the search for an optimal solution is at least partially guided by a human user who is able to make case-specific decisions, which will be touched upon in Chap. 10. Besides a UI prototype for concurrent synchronisation, the VICToRy debugger enables the user to interactively guide forward and backward transformations to support the understandability of TGG-based consistency management.

8.7 Wrap-up of the Hybrid Approach

With the hybrid approach of Chap. 5, 6, 7 and 8, a powerful conceptual framework for fault-tolerant consistency management has been proposed. A considerable range of consistency management operations, as well as additional graph constraints are supported. Fault-tolerance is reached by computing the largest consistent sub-triple in case of inconsistent input models. The solution strategy, i. e., constructing an optimisation problem from a superset of rule application candidates, in which each candidate is encoded as a binary variable, is shared between all operations.

The results should be interpreted differently for the different operations, though: For the consistency checking operations CO and CC, the information whether the input models are consistent or not is more important than the transformation result. The largest consistent sub-triple can be indeed used for fault detection purposes. For forward and backward transformation (FWD_OPT, BWD_OPT), the fault-tolerant approach provides the user with a result that is "as consistent as possible" instead of rejecting the input, which eases the user's work-flow substantially. The untranslated elements can be regarded as open to-dos that should be resolved before a new attempt is started. For concurrent model synchronisation, the optimisation pursues a slightly different goal: Instead of maximising the number of marked elements, the goals of preserving as many user edits as possible and changing the rest of the input models to the smallest possible extent must be balanced for this operation. With the exception of very small or trivial examples, the operation will never mark the entire input, especially because marking elements of the delete delta should be avoided.

Most of the results have been presented in prior work already, whereas the integration of graph constraints into the concurrent synchronisation operation is a novel contribution of this thesis. An overview of the involved publications is given in Fig. 8.8. The concurrent synchronisation operation is abbreviated with *CS*. Concurrent synchronisation with graph constraints – which did not appear in any publication yet – is marked with a $\star$.

Besides being the conceptual contribution of this thesis, the hybrid approach is completely implemented as part of the eMoflon tool suite. The four operations of Chap. 5 exist both in eMoflon::IBeX and eMoflon::Neo, whereas the extensions of Chap. 6, 7 and 8 are

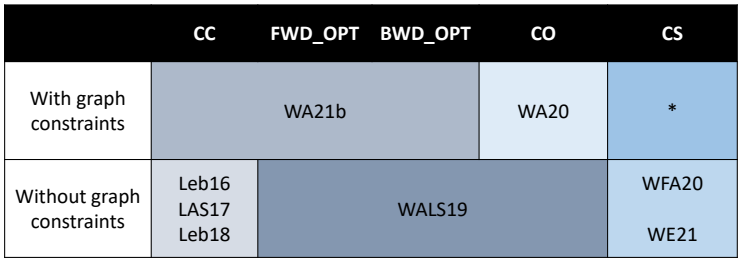


Figure 8.8: Overview of publications related to the conceptual solution

only implemented in eMoflon::Neo. As a meta-result of all four experimental evaluations, one can state that consistency checks as well as forward and backward transformations scale reasonably well, also after adding negative constraints. Further improvements are necessary for TGGs with implication constraints, and for concurrent synchronisation in general. The idea of replacing the ILP component by heuristic solvers did not lead to performance improvements due to the small portion of feasible solutions in the search space.

The influence of the underlying technology of the tool implementation has not been analysed up to here. In order to investigate strengths and weaknesses of graph databases and the EMF framework in the concrete application scenario, the two components of the eMoflon tool suite will be compared in terms of their overlapping parts in Sect. 9.5. To enable a comparison and evaluation of future fault-tolerant approaches to consistency management, we also plan to extend the existing benchmark for consistency management [ABW17] to cover fault-tolerant application scenarios. Finally, two industrial case studies will be presented in Chap. 11 and 12 to demonstrate application areas of the developed approach.

Part III

Tools and Applications

9 The eMoflon Tool Suite

In the conceptual part of this thesis, i. e., Chap. 5 – 8, an extensive formal framework for fault-tolerant consistency management was presented. All aspects of the hybrid approach of combining TGGs and optimisation techniques are implemented as part of the *eMoflon tool suite*, which will be briefly described in this chapter. In Chap. 11 and 12, we will see how the tool suite, and especially the implementation of the hybrid approach, can be used to solve consistency management problems in practice. This chapter is structured as follows: Section 9.1 gives a brief overview of the development history of eMoflon and its predecessor tools, and thereby motivates the conceptual and technological decisions that characterise the tool suite. A comparison with related MDE tools is drawn in Sect. 9.2. The architecture of the components IBeX-GT (Sect. 9.3), IBeX-TGG (Sect. 9.4) and Neo (Sect. 9.5) is introduced subsequently. The tool suite is analysed with respect to scalability and usability in Sect. 9.6 and 9.7, before Sect. 9.8 summarises the findings.

9.1 Introduction and a Brief History

The eMoflon tool suite[1] consists of the two main components *IBeX* and *Neo*, which provide us with all necessary functionality to implement the hybrid approach for fault-tolerant consistency management. In order to be a suitable basis for the implementation, besides supporting the TGG formalism including its language features attribute conditions (cf. Sect. 4.4) and application conditions or graph constraints (cf. Sect. 4.5), a tool must meet several requirements:

- **Incrementality**: The hybrid approach collects matches for rule applications and constraint patterns in multiple iterations. Therefore, it is necessary for an efficient implementation to distinguish between new matches and matches that have been collected in a previous iteration already.
- **Scalability**: For practical use cases, models with thousands of elements must be handled in a time- and resource-efficient manner.
- **Flexibility**: One of the main steps of the hybrid approach is the collection and filtering of potential rule applications, which requires to substantially modify the input models in the course of the consistency management operations. Furthermore, the handling of faulty inputs requires to handle violations of, e. g., metamodel constraints, such that a high degree of flexibility is necessary in this regard.
- **Modularity**: The work-flow of the hybrid approach involves tasks such as ILP solving or pattern matching, which are not specific for consistency management problems. There exist mature external components which can be used for this purpose, instead of reinventing the wheel with a hand-crafted implementation. As a result, the tool in use must have a modular architecture and interfaces to attach these external components.

[1] www.emoflon.org

We claim that eMoflon fulfils these four requirements and provide evidence in the remainder of this chapter. The tool suite results from a development process of several decades and ranging over different components, whereby experiences made with predecessor tools were useful for continuously improving the tool support. Therefore, we start with a brief historical overview that illustrates the most important technological changes. Figure 9.1 depicts the history of eMoflon, showing both direct predecessors and some related tools. Nodes represent tools, while edges indicate that one tool (successor) conceptually or/and technically evolved from another (predecessor). All edges ultimately leading to IBeX and Neo are labelled, indicating the primary reason for the evolution. Related tools are greyed out, while tools that are currently part of the eMoflon tool suite are highlighted with a light-blue background.

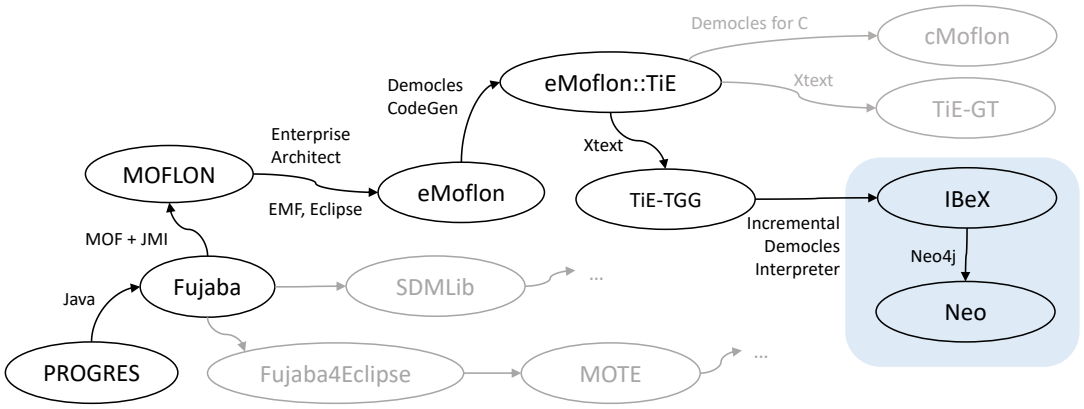


Figure 9.1: History of eMoflon

Starting with PROGRES [Sch89], one of the first tools for programmed graph transformations, Fujaba [BGN+04] was developed based on the mainstream GPL Java. With the goal of implementing the full Meta Object Facility (MOF) 2.0 and Java Metadata Interface (JMI) standard, MOFLON [AKRS06] was developed as a plugin for Fujaba. With the success of Eclipse as an IDE platform, and EMF/Ecore as a *de facto* modelling standard, eMoflon [ALPS11] was developed as a complete re-engineering of MOFLON. In addition, Enterprise Architect (EA)[2] was established as a visual front end for GT.

In its back end, eMoflon was still using the pattern matcher of Fujaba, which became increasingly challenging to evolve and maintain, partly because it was bootstrapped with a different tool chain. Based on Democles as a new pattern matcher [VAS12], eMoflon::TiE [LAS14a] was developed as a Democles-based version of eMoflon. In addition to providing a unified platform for both an interpretative and generative approach to model transformation, Democles was also designed to simplify exchanging all templates for code generation. This was exploited to establish cMoflon [KSG+17], a GT tool that generates embedded C code.

While EA proved to be a scalable and relatively usable front end for eMoflon, it required a separate tool chain based on C# and Visual Studio. Combined with problems concerning licensing and cross-platform support, a decision was made to switch to Xtext[3] as an editor framework, and use PlantUML[4] for generated, read-only visualisations. This led to eMoflon::TiE-TGG for TGGs as a pilot project, and some time later, eMoflon::TiE-

[2] www.sparxsystems.de
[3] www.eclipse.org/Xtext/
[4] http://plantuml.com/en/eclipse

GT for GT. For further details on this EA to Xtext migration, we refer to Yigitbas et al. [YALG18].

Driven by our requirement for incrementality, we developed eMoflon::IBeX based on the incremental Democles interpreter [VD13]. Due to the respective modular interface, support for other incremental pattern matchers such as VIATRA [VBH+16] or HiPE[5] was subsequently added, as well as a proof of concept for the rule engines Drools[6] and nools[7]. While the requirements regarding incrementality and modularity are met by IBeX, we still saw room for improvement with respect to scalability and especially flexibility, caused by the strict metamodel conformance requirements posed by the EMF. As a result, eMoflon::Neo was developed as an EMF-independent component that uses the graph database Neo4j[8] to store models on disk. After a treatment of related tool support in Sect. 9.2, these two components will be presented in more detail.

9.2 Related MDE Tools

In recent years, several implementations of the TGG formalism have been proposed. We will briefly discuss the potential of each tool to serve as a basis for implementing the conceptual framework of this thesis.

MoTE [HLG+11, GHL14] is a TGG-based tool with a strong focus on scalability. While it supports numerous operations, it also poses strong restrictions on the class of supported TGGs (only basic language features and attribute conditions are supported). These restrictions simplify especially synchronisation and consistency checking tasks, but also severely limit expressiveness.

EMorF [KW12] interprets TGG rules and claims to support model transformation, synchronisation, and consistency checking. GT and TGGs are seamlessly integrated, such that users are able to mix and switch between these two formalisms, and TGG developers can reuse functionality of the GT layer. As neither the source code nor an installation of EMorF is publicly available, however, it is difficult to assess under which assumptions the different operations actually work.

The GT tool Henshin [BET12] is an EMF-based Eclipse plug-in with a wide range of supported language features. Inspired by the long-term experience in developing the GT tool, Henshin-TGG [EHGB12] was implemented as a standalone component that extends Henshin towards TGG concepts. HenshinTGG supposedly supports forward and backward transformation, synchronisation and consistency checks, but as far as we can tell from simple experiments, the choice of rules for each operation must be deterministic, severely restricting the class of supported TGGs.

Fujaba [GW06, BGN+04] provided one of the first implementations of the TGG approach and – as far as we can assess – already supported both model transformation and synchronisation. As shown in Fig. 9.1, Fujaba can be considered as a predecessor of both MoTE and eMoflon. The tool is, however, out of maintenance for several years, and therefore only of theoretical value.

Similar to EMorF, eMoflon::TiE [LAS14a], the predecessor of IBeX, integrates GT and TGG concepts into one tool. As a handy feature to reduce code duplication, rule refinement [KKS07] is supported for TGG rules; This modularity feature is generalised in IBeX to uniformly cover patterns, GT rules, and TGG rules. Both unidirectional model

[5] https://hipe-devops.github.io/HiPE-Updatesite
[6] https://www.drools.org/
[7] https://www.npmjs.com/package/nools
[8] https://neo4j.com/

transformation and model synchronisation, as well as consistency checking are supported. Each operation is, however, essentially a completely separate implementation making it increasingly costly to maintain and extend the tool.

The TGG Interpreter [GK10] directly interprets TGG rules and supports model transformation and synchronisation. Existing surveys of TGG tools indicate, however, that the TGG Interpreter does not scale well compared to MoTE and eMoflon::TiE [HLG+13, LAS+14b].

The UML tool USE was extended to support consistency management based on TGGs by translating TGG rules into OCL constraints [DG08]. Specified TGG rules can be operationalised for model transformation, synchronisation and consistency checking. As far as we can tell, however, the *application* of these rules remains a manual task.

While most of the already mentioned tools are based on EMF or purely hand-crafted, there are also a few MDE tools that leverage the potential of graph databases for software modelling. Neo4j has been used as an underlying graph database for the GT tool GRAPE [Web17]. An embedded Domain-Specific Language (DSL) in Closure is used to define rules, from which statements in Cypher – the declarative pattern matching and transformation language for Neo4j – are generated to query the database. GRAPE uses a textual concrete syntax together with a visualisation, and supports GT on untyped graphs. The TGG formalism is not supported by GRAPE, though.

Alqahtani and Heckel use Neo4j for TGG-based model transformations [AH18]. TGG rules are translated into Gremlin code, which is an alternative query language for Neo4j. In an experimental performance comparison with eMoflon::IBeX, their approach shows better scalability results. Their implementation, however, only supports forward and backward transformations without completeness guarantees. Daniel et al. [DJSC17] use Gremlin for ATL-based graph transformations in a similar fashion. Besides Neo4j, adapters for other NoSQL databases exist, such as OrientDB and MongoDB.

NeoEMF [DSB+16,DSB+17] was recently proposed as a seamless EMF-compatible layer over Neo4j and other NoSQL databases. The advantages of graph databases with respect to scalability and the well-known EMF resource handling are synergetically combined, which makes it possible to attach NeoEMF to EMF-based BX tools. Model transformations are supported, but take place in main memory and have to be replicated in the database. Furthermore, using this intermediate layer restricts the control over how, e. g., types are mapped to Neo4j, and prevents leveraging all advantages of the particular database, such as attributed edges for bookkeeping operations.

In summary, an implementation of the hybrid approach would be possible with none of the existing TGG tools without larger efforts. Either the precondition of supporting TGGs and advanced language features are not fulfilled, or one or more of the requirements listed in Sect. 9.1 are not met. Often, the tools are out of maintenance or no longer available. Studying related approaches has however shown that the underlying technologies of IBeX and Neo, i. e., EMF and Neo4j, have been frequently used for software modelling already. In the following, we will present how the hybrid approach was implemented as part of the eMoflon tool suite.

9.3 Graph Transformation with IBeX-GT

eMoflon::IBeX is implemented as a set of Eclipse plug-ins and supports both unidirectional and bidirectional model transformations with GT and TGGs, respectively. In this section, the software architecture of the GT component will be sketched, and extended by the TGG-specific part in Sect. 9.4. By developing the component, we have realised a novel mix

of complementary tool features that have proven to be useful and effective in predecessor tools. We discuss these features and present insights based on an empirical evaluation of eMoflon::IBeX in Sect. 9.7.

Figure 9.2 provides an architectural overview of IBeX: The TGG layer makes use of the GT layer, which consists of a front end and a back end component. The front end consists of an Xtext-based[9] editor combined with a read-only visualisation using PlantUML[10]. As input to the front end, users provide `.ecore` files for all metamodels, and `.gt` files containing graph transformation rules in a textual concrete syntax.

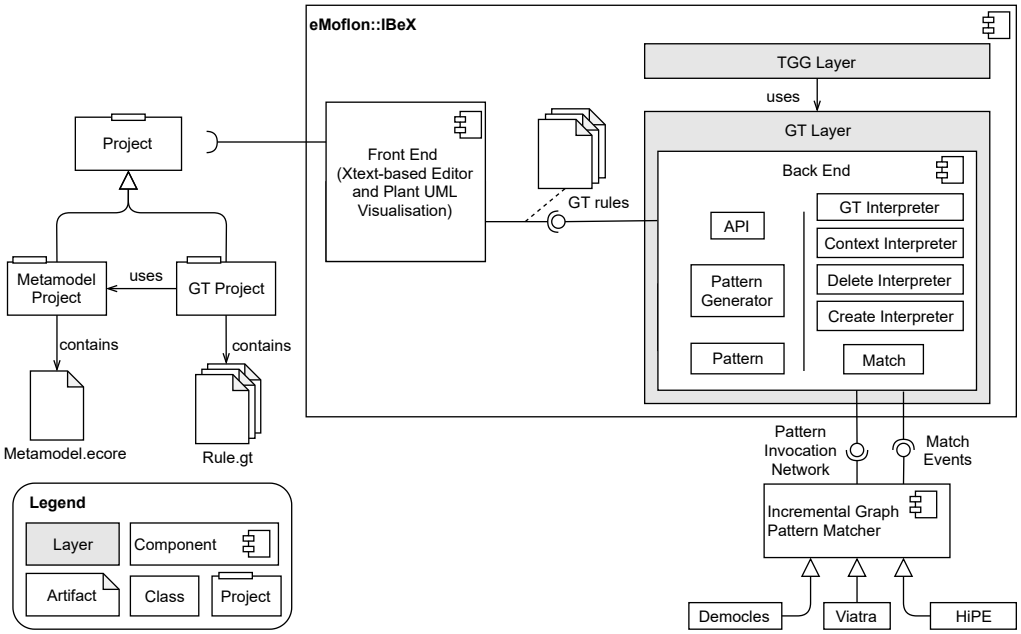


Figure 9.2: Most important components and classes in eMoflon::IBeX

The front end is completely independent of the back end and provides an extension point[11] via which the metamodels and a set of rules (as EMF (meta-)models) are offered. As the syntax for GT and TGG rules is very similar and the Ecore format is used for TGG metamodels as well, we will not go into further details at this point and refer to Sect. 9.4 instead. Note that the metamodels and graph transformations are stored separately, such that a metamodel can be used in multiple GT projects. The front end produces GT rules (as EMF models) and passes them to the back end.

Figure 9.2 depicts the most important interfaces and classes in the back end divided into compile time (to the left) and runtime (to the right). At compile time, the back end uses a `PatternGenerator` to generate a set of separate `Patterns` from a GT rule. These patterns represent the context to be matched, elements to be deleted, and elements to be created. A typed Application Programming Interface (API) specially tailored for the set of GT rules is generated as Java code and produced as output for the user. This `API` wraps all calls to the `GTInterpreter` allowing for type safe access and rich compiler errors if rules are specified inconsistently.

At runtime, the `GTInterpreter` delegates the task of pattern matching to a `Context-Interpreter` as a separate component. A so-called *pattern invocation network* (an acyclic graph with patterns as nodes and invocations as edges) is passed to maximise

[9] https://www.eclipse.org/Xtext/
[10] http://plantuml.com/
[11] In the Eclipse framework, components are *plug-ins* that provide *extension points* and require *extensions*.

reuse of partial matches. The incremental pattern matcher produces *match events* as output, signalling when new matches appear (create match events), and when old matches are violated (delete match events), as the models are manipulated. To use eMoflon::IBeX, an adapter for an incremental pattern matching engine is necessary, whereby adapters for several pattern matchers are delivered with the tool (cf. Sect. 9.1). The `GTInterpreter` collects all `Matches` and performs rule application by delegating deletion to a `DeleteInterpreter` and creation to a `CreateInterpreter`. While IBeX supplies default implementations for deletion and creation, these can be extended or replaced for special cases or optimisations.

Figure 9.3 depicts a communication diagram representing the GT rule application process at runtime. (1) The generated API serves as a factory for GT rules, providing methods for all non-abstract rules. (2) Rules can be used to subscribe for appearing or disappearing matches reported by the `GTInterpreter`. Rules wrap the generic interpreter to avoid casting in developer code. (3) The interpreter initialises the `ContextInterpreter` for pattern matching, (4) the `DeleteInterpreter` for deletion, and (5) the `CreateInterpreter` for creation. When the monitored models are manipulated, (6) the `ContextInterpreter` produces and reports generic match events. (7) The `GTInterpreter` notifies the rule, which then (8) converts the generic matches to typed matches and provides them to the user via a series of methods such as `findAnyMatch` or `forEachMatch`, designed to work together with the standard Java stream API.

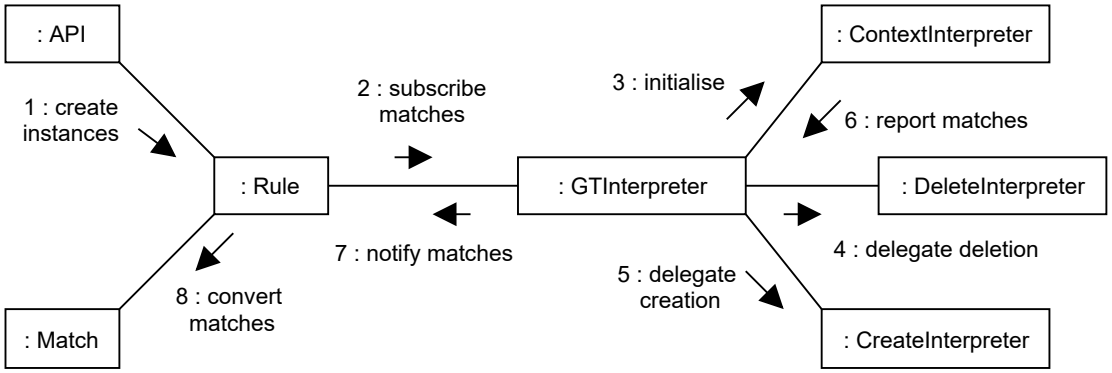


Figure 9.3: Communication between API and GT Interpreter

With the GT component of eMoflon::IBeX, the foundations for rule-based incremental pattern matching were laid as a basis for implementing the hybrid approach as part of the TGG component. In the upcoming Sect. 9.4, the TGG layer will be introduced.

9.4 Bidirectional Model Transformation with IBeX-TGG

In this section, we present the TGG layer of eMoflon::IBeX, in which the concepts of Chap. 5, i. e., fault-tolerant forward and backward transformation and consistency checks, were implemented first.

Back-end

Figure 9.4 provides a structural overview of the architecture of eMoflon::IBeX as an extension of Fig. 9.2, whereby details of the GT back-end are omitted for better readability. New components and artefacts are depicted in green, whereas parts which are shared or equal for the GT and TGG layer remain black. For the front end, the technologies in

use (Xtext, PlantUML) are equal for GT and TGG projects. The TGG is technically specified in form of `.tgg` files contained in a TGG project. To simplify configuration and usage of the system, stubs are generated for every supported consistency management operation as `.java` files. These stubs can be executed directly with default settings, and also configured and adapted as required.

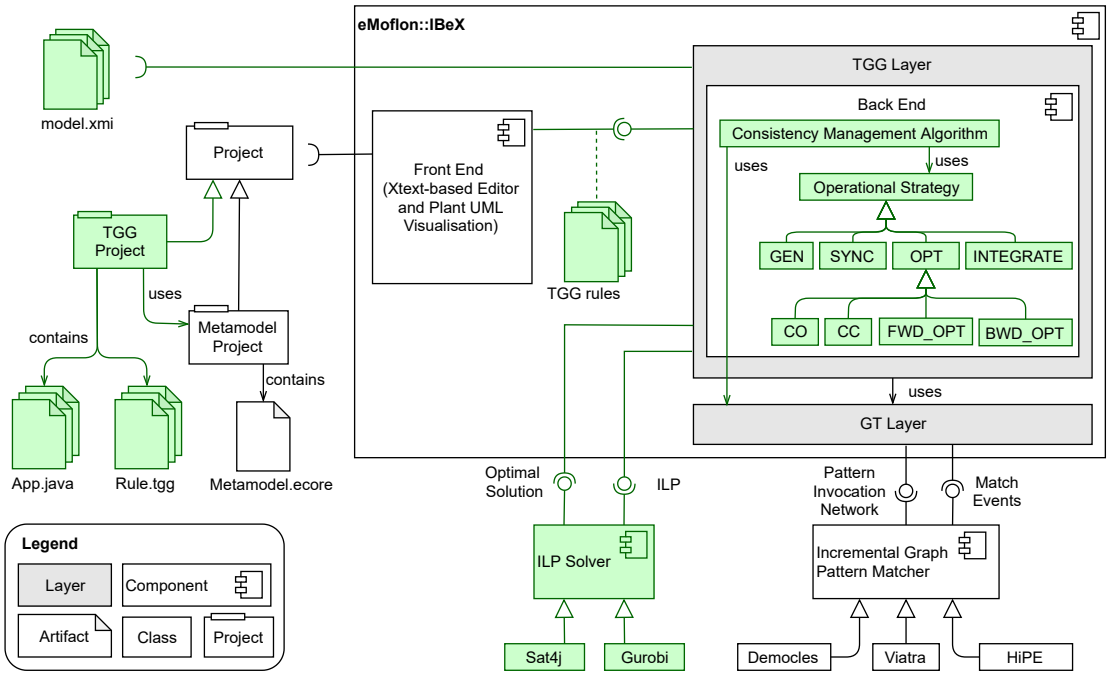


Figure 9.4: Component diagram: Architecture of eMoflon::IBeX

The back end takes a triple of metamodels and a set of rules as input, as well as an input triple (`.xmi` files) that is expected to be typed over the triple metamodel. It relies primarily on two external components for performing consistency management: (i) an incremental graph pattern matcher (known from Sect. 9.3 already) to efficiently determine solution candidates, and (ii) an ILP solver to choose the optimal solution from these candidates. For the ILP solver, IBeX provides the ILP as input, computed from the set of solution candidates, and an objective function that depends on the specific consistency management operation, as presented in Chap. 5. The ILP solver chooses an optimal solution based on the provided candidates by maximising the objective function. Currently, IBeX is distributed with SAT4J[12] as default ILP solver. To test our interfaces, we have implemented well-tested adapters for alternative ILP solvers including Gurobi,[13] CBC,[14] GLPK,[15] and MIPCL.[16] In structured tests, Gurobi outperformed all other solvers in prior experiments [Opp18], which is why this solver was used for all scalability tests of this thesis. Considering the set of connected graph pattern matchers and ILP solvers, our achievements indicate that the interfaces are generic enough to enable an integration of other external components of this kind with acceptable effort.

The uniform consistency management algorithm in the back end comprises two main tasks that can be configured as required to implement various operations: the first task

[12] sat4j.org
[13] www.gurobi.com/products/gurobi-optimizer
[14] projects.coin-or.org/Cbc
[15] www.gnu.org/software/glpk/
[16] mipcl-cpp.appspot.com/

is to generate a suitable pattern invocation network from a set of TGG rules, while the second task is to execute the actual operations, which are technically denoted as "operational strategies". While the first task is carried out by the pattern generator of the GT layer, the second task is the key functionality of the TGG layer. Both interfaces are implemented for the various consistency management operations: Besides the four basic operations of the hybrid approach (Chap. 5), which technically share a common superclass "OPT" due to their similarities, there are also operations for generating consistent models (GEN) and model synchronisation (SYNC), which follow a greedy solution approach. The INTEGRATE operation is an implementation of the concurrent synchronisation approach of Fritsche et al. [FKM+20].

Figure 9.5 provides a dynamic view on the uniform algorithm used for all operations – including GEN, SYNC and INTEGRATE – as an activity diagram.

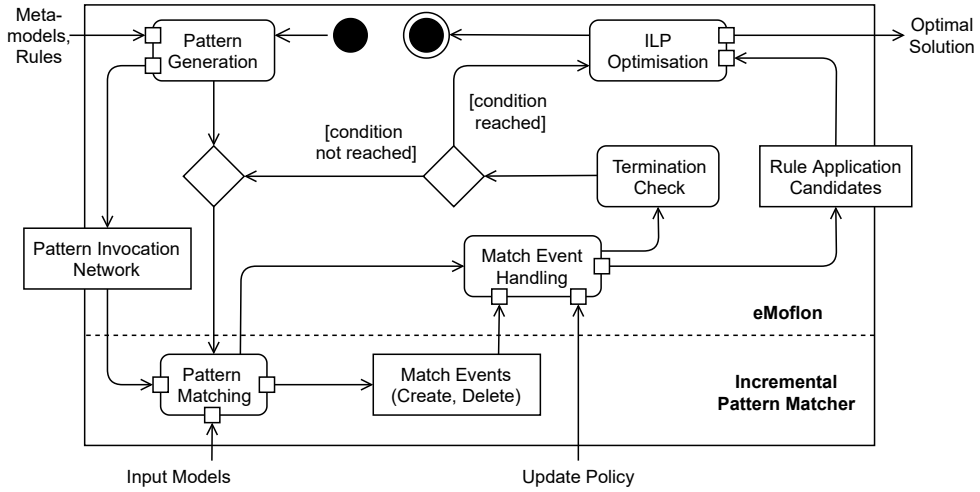


Figure 9.5: Activity diagram: Uniform consistency management algorithm

Compared to the conceptual work-flow of Fig. 5.1, this diagram describes the consistency management process from a technical perspective. The first action is *pattern generation*, which is comparable to the operationalisation step (A) of the work-flow, requiring a triple metamodel and TGG rules as input. This is followed by *pattern matching* on the input triple (step (B) of the work-flow), based on the generated pattern invocation network. This action generates a stream of match events (create or delete), which are passed on to the *match event handling* action. Depending on the concrete operation, match events can be handled by creating, deleting, or manipulating elements in the input models. In general, this action is non-deterministic and can be controlled by an *update policy* that, for example, can choose which match event to handle from the set of all pending events.

After one or numerous match events have been handled (the handler decides when it is finished), a *termination criterion* (depending on the concrete operation) decides if all relevant information for computing the output models are collected, or if the pattern matching and match event handling process is to be repeated. This termination criterion can range from a simple time-out for model generation to a check for an empty set of new match events for all operations of the hybrid approach (indicating that no new rule application candidates can be found). The advantage of using an *incremental* graph pattern matcher is that the latest match events can be produced relatively efficiently without any extra effort or explicit "search" for new or removed pattern matches.

When the stop criterion is fulfilled, the current set of *rule application candidates* is passed to the final action in the process, *ILP optimisation* (comprising the steps ILP

construction (C) and optimisation (D) of the work-flow). While the construction of the ILP is done within eMoflon, the optimisation is taken over by the external solver. This last action is only applicable for operations of the hybrid approach, because for the greedy operations GEN, SYNC and INTEGRATE, a single candidate is produced and returned at this stage without any optimisation.

Front-end

After presenting the software architecture of eMoflon::IBeX from a static and dynamic perspective, the focus is shifted to the front end of the tool in the remainder of this section. Figures 9.6 and 9.7 depict the metamodels of the example transformation from SysML to Event-B (cf. Fig. 3.8) in form of `.ecore` files opened with the Sample Ecore Model Editor of Eclipse (the specification can also take place in any other editor of choice). These (eMoflon-independent) metamodels are connected to a triple metamodels in a separate `.tgg` file, which is shown in Fig. 9.8. For the correspondence metamodel, IBeX provides a dedicated textual concrete syntax with which the roles (source or target) of the metamodels are assigned, and 1-1 correspondence types (e. g., `StatemachineToMachine`) connecting a source type with a target type.

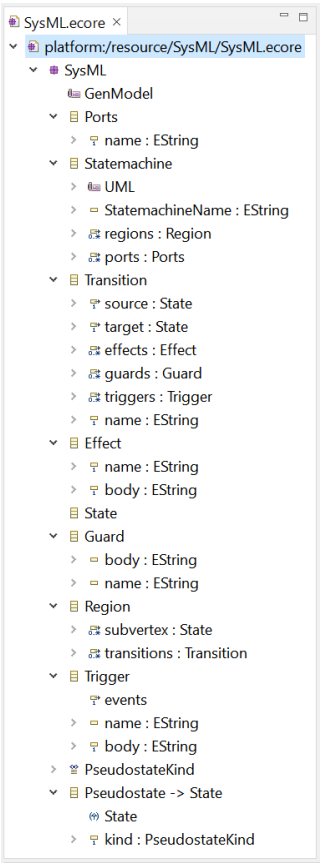


Figure 9.6: SysML

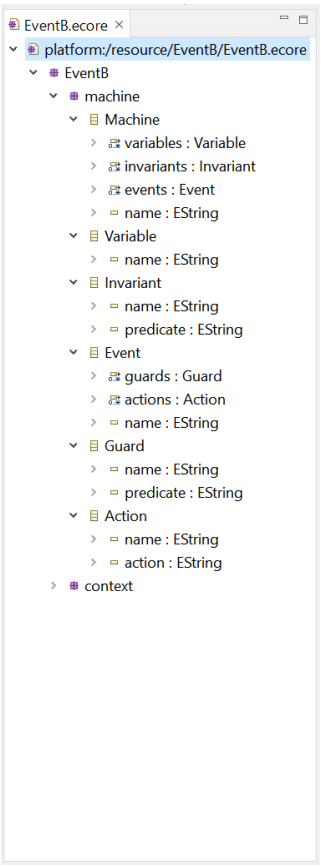


Figure 9.7: Event-B

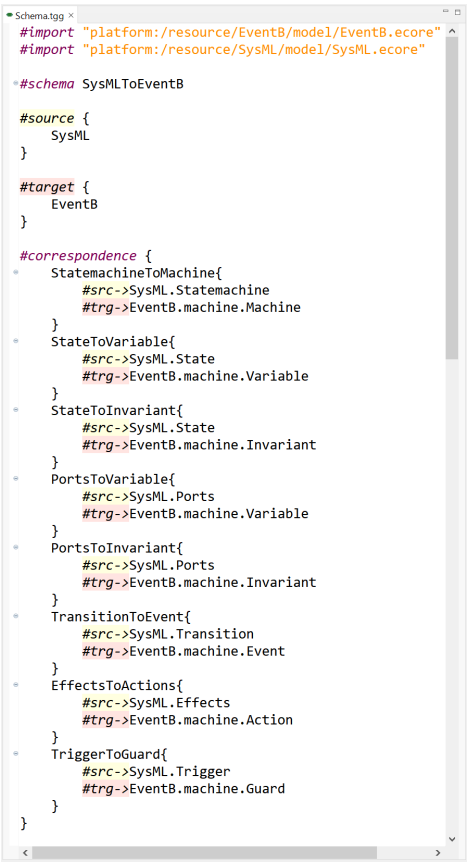


Figure 9.8: Correspondences

Figure 9.9 depicts the rule *StateToVariable* (cf. Fig. 4.3), also specified in a textual concrete syntax. Additional *attribute conditions* such as `eq_string`, from an extensible library of conditions implemented in Java, can be specified in a simple textual syntax. In this example, the conditions `setType` and `addPredicate` were added to express the two rather complex attribute conditions of the rule. The textual representation is

complemented with an automatically generated read-only visualisation, which is shown in Fig. 9.10. The visualisation is dynamic in the sense that it constantly adapts to the current selection in the textual editor. The syntax is almost equal to the compact visual notation used throughout this thesis, and therefore not further explained at this point.

```
StateToVariableRule.tgg
#using SysMLToEventB.*
#using AttrCondDefLibrary.*

#rule StateToVariableRule #with SysMLToEventB

#source {
    sm : SysML.Statemachine{
        -regions->r
    }
    r : Region{
        ++ -subvertex->s
    }
    ++s : State
}

#target {
    m : EventB.machine.Machine{
        ++ -variables->v
        ++ -invariants->i
    }
    ++i: Invariant
    ++v : Variable
}

#correspondence {
    sm2m : StatemachineToMachine{
        #src-> sm
        #trg-> m
        }

    ++s2v : StateToVariable{
        #src->s
        #trg->v
    }
    ++s2i : StateToInvariant{
        #src->s
        #trg->i
    }
}

#attributeConditions {
    eq_string(s.name, v.name)
    setType(s.name, i.name)
    addPredicate(s.name,i.predicate)
}
```

Figure 9.9: Rule: StateToVariable

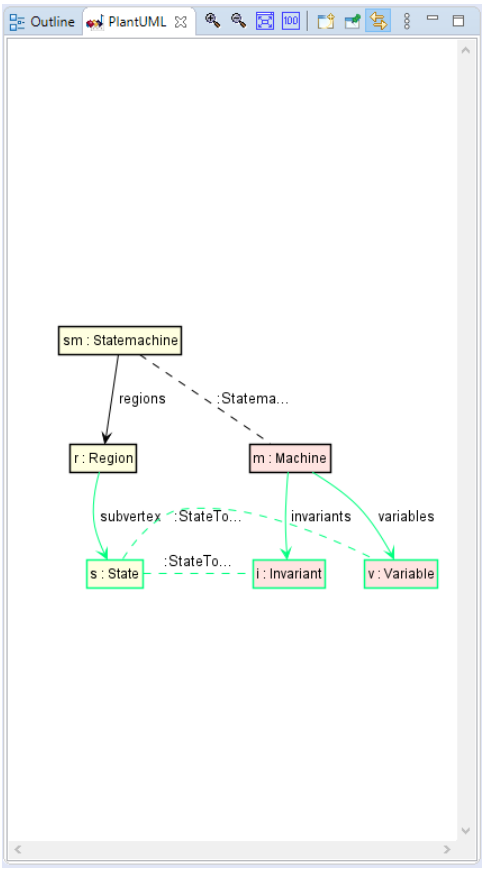


Figure 9.10: PlantUML visualisation

In summary, the implementation of the operations CO, CC, FWD_OPT and BWD_OPT in IBeX benefits from the modular architecture of the tool, that enables us to shift the pattern matching and ILP solving steps to external components. It became apparent, though, that the extension towards tolerating domain constraints violations is extremely difficult, as long as all models and metamodels must comply to the quite restrictive EMF standard. Furthermore, the mix of different file formats appears to be unnecessarily complicated for working with it on a daily basis. For these reasons, we decided to implement the entire hybrid approach as part of *eMoflon::Neo*, which overcomes the restrictions of the EMF by using a graph database as model storage. The second component of the eMoflon tool suite is presented in the upcoming Sect. 9.5, emphasizing the differences to eMoflon::IBeX.

9.5 Consistency and Model Management with Neo

In this section, we present *eMoflon::Neo* as the latest addition to the eMoflon tool suite, which uses the graph database Neo4j as a storage for models and metamodels at runtime. Although the EMF provides a solid basis for developing modelling tools, our experience from developing eMoflon::IBeX (Sect. 9.3 and 9.4) is that EMF has some drawbacks, especially for the extension of the hybrid approach towards graph constraints (Chap. 6)

and concurrent synchronisation (Chap. 7 and 8). With respect to the four tool-specific requirements listed in Sect. 9.1, we have identified the following limitations regarding *scalability* and *flexibility* directly related to representing our runtime models as EMF data structures:

- **Scalability:** EMF models must fit completely into the main memory for operating on them, which limits the handling of very large models. While we are not necessarily interested in extremely large models per se, we
  (i) represent traceability links and other bookkeeping information such as various markers explicitly in models,
  (ii) generate multiple candidate structures before using an ILP solver to pick the best result. Both points mean that we have to handle effective model sizes factors larger than the actual input model sizes.

- **Flexibility:** Although the relatively strict conformance relation between EMF models and their metamodels certainly has its advantages, it is more often a hindrance that our algorithms have to work around. There are mainly two reasons for this:
  (i) being able to enrich model elements with markers and other extra information often simplifies analyses and bookkeeping operations, and
  (ii) when collecting all possible rule application candidates for the final optimisation step, we need to construct a "super model" that violates metamodel constraints such as multiplicities and single containment relations.

  EMF, however, does not support attributes for edges, and temporarily extending or relaxing metamodels of loaded models is non-trivial.

Especially the last aspect is a specific challenge for the fault-tolerant hybrid approach: Greedy operations reject inputs that violate metamodel constraints right away, and do not need to generate more elements than necessary when solving transformation and synchronisation tasks. While the requirements of the EMF framework do not appear too restrictive for greedy operations, they can become a serious obstacle for fault-tolerant implementations. As a reaction to these drawbacks, we have decided to explore graph databases as an alternative infrastructure for handling our runtime model operations.

NoSQL databases in general, and graph databases in particular, have gained popularity in recent years and have been successfully leveraged for developing MDE tools [Web17, DJSC17]. Using a graph database to represent runtime models can address both aforementioned issues: First, graph databases promise improved scalability via on-demand caching, indexing, and a native representation of nodes and edges. Second, models and metamodels are (typically) both represented as plain graphs with type edges and constraints representing the conformance relation. This means that the relation can be (temporarily) violated and later re-established as required with the standard infrastructure.

In order to unify the textual specification of (meta-)models, rules, patterns and constraints, a uniform textual modelling language, denoted as *eMSL*, was developed for eMoflon::Neo. eMSL can be regarded as a family of modelling languages with a uniform textual concrete syntax supported by an Xtext-based editor. Likewise, for each eMSL language construct, there is a corresponding visualisation in PlantUML. We have found this to be beneficial especially for teaching as students only have to learn how to use one consistent family of languages. In the following, the back end and front of eMoflon::Neo will be briefly presented, focussing on differences compared to eMoflon::IBeX.

Back-end

The software architecture of eMoflon::Neo is depicted in Fig. 9.11 as a component diagram. The eMSL language (file extension `.msl`) is used to uniformly specify all involved model management artefacts. For GT and TGG projects, both metamodels and (triple) graph grammars are specified and stored in this format.

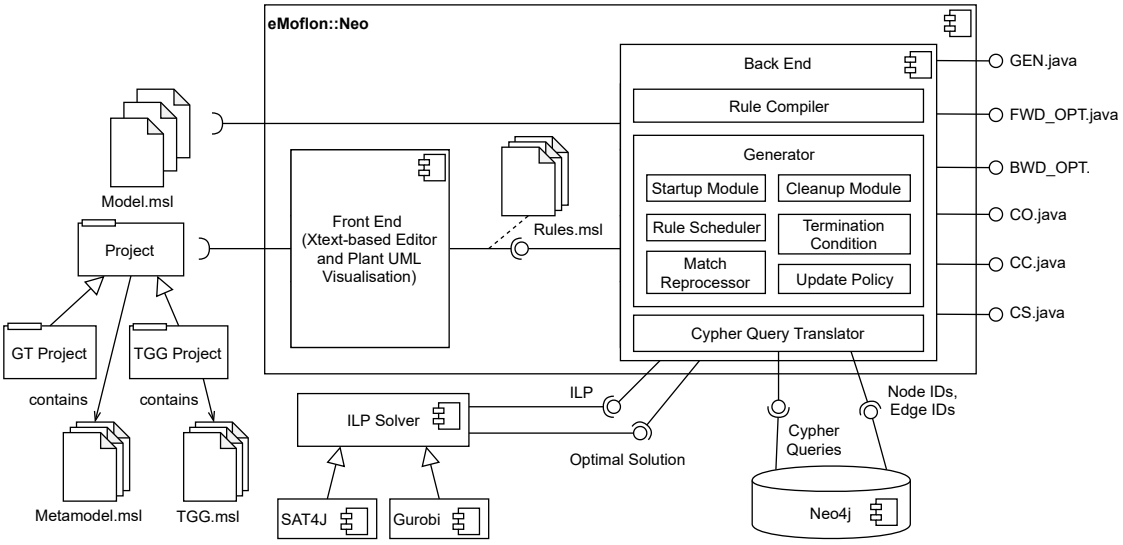


Figure 9.11: Architecture of eMoflon::Neo

The back end of eMoflon::Neo can be subdivided into several components as depicted in Fig. 9.11. At compile time, the `Rule Compiler` uses the TGG specification to generate operational rules (also in eMSL) for all supported operations. IBeX, in contrast, generates patterns for each rule and operation that are specific for each pattern matcher. The `Generator`, composed itself of several modules, is used to perform all consistency management operations, which can be configured via the generated operational rules and Java API code. Finally, a `Cypher Query Translator` connects the back end to the Neo4j database, which contains all runtime models. Cypher queries are generated to collect matches for operational rules and apply them on the models. The results are returned as an array of IDs for nodes and edges, which are either part of the match, or have been created by rule applications. Querying the database in Neo is therefore comparable to pattern matching on EMF models in IBeX.

The range of supported consistency management operations differs slightly between Neo and IBeX. The operations CO, CC, FWD_OPT and BWD_OPT as core part of the hybrid framework are supported by both components, as well as the GEN operation for randomly generating consistent triples. While IBeX uses greedy operations for model synchronisation (SYNC and INTEGRATE), the concurrent synchronisation operation described in Chap. 7 and 8 is implemented in Neo, denoted as CS in the following. As propagating updates on one model to the unchanged other model is a special case of concurrent synchronisation, no second operation is implemented for updates on one model. Similar to IBeX, Neo currently supports SAT4J and Gurobi as ILP solvers; adapters for other solvers can be added as required.

All consistency management operations follow a common work-flow, which we denote as the "core cycle", depicted in Fig. 9.12 as a UML activity diagram. Each activity is implemented as a module, which can be reused and combined with other modules to configure an operation.

The *start-up module* performs initialisation steps, such as setting temporary translation markers to their default value. In a loop, matches for (potential) rule applications and other patterns (e. g., for constraints) are collected: The *rule scheduler* selects rules for the subsequent pattern matching step, for which a maximum number of matches can be set. This is especially helpful for model generation (GEN), and can also be useful for other operations on very large models. For *pattern matching*, the first costly step in the database, the scheduling request is translated into a cypher query for the database. Based on the query results, matches are added to a match container. If this container is non-empty, matches are *selected* from the match container to be applied according to an update policy.

While it is possible and greatly improves performance to select multiple matches to be applied in parallel, the update policy must guarantee that these matches are not in conflict with each other, i. e., make sure that only one of the potentially conflicting matches are chosen for rule application. The selected rules are then *applied* in a subsequent step in the database. Depending on the update policy, it is possible that there are still unused matches in the match container at this stage. *Match reprocessing* denotes the strategy applied to determine which matches can be safely used for the next iteration, and which have become invalid and must thus be removed. In a final step of the main loop, a pre-defined *termination* condition is *checked*. Such a condition could be, e. g., that no new matches were found in the last iteration. If this condition does not yet hold, a new iteration of the main loop begins. Otherwise, the *clean-up* module prepares the operation's termination. Depending on the concrete operation, a clean-up can entail removing all temporary markers, performing ILP solving to determine the optimal result from a set of candidates, deleting all elements created by other candidates, etc. In this step, it is also guaranteed that the produced result fulfils all posed constraints.

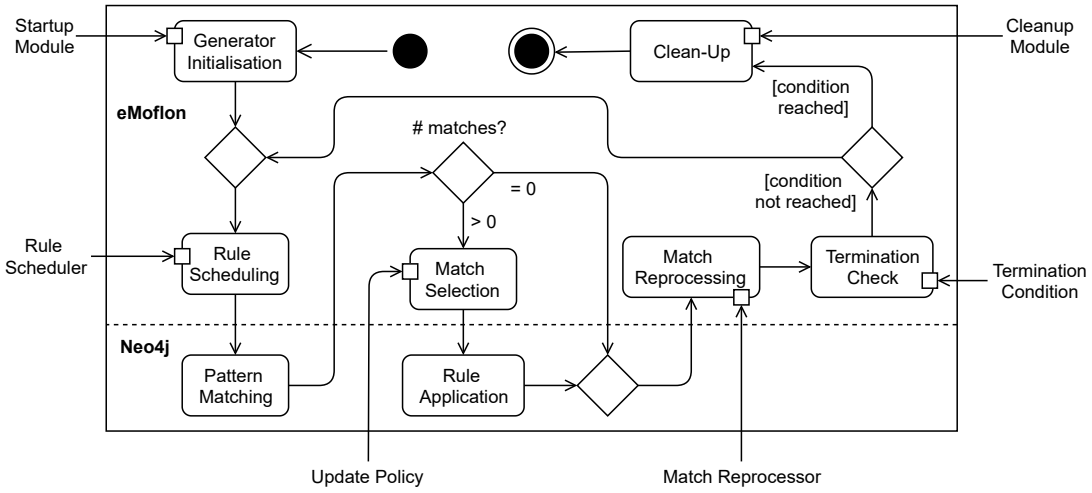


Figure 9.12: Core cycle for consistency management

Front-end

In the following, an overview of the front-end of eMoflon::Neo is provided, and compared to the front-end of eMoflon::IBeX. All textual specifications are expressed in eMSL, replacing the mix of Ecore, XMI and other text file formats. To complement the textual eMSL editor, PlantUML diagrams are automatically generated for all eMSL entities including (meta)-models, graph patterns, constraints and rules. The PlantUML visualisation is almost equal to IBeX, and therefore not repeatedly discussed here.

Figure 9.13 depicts the textual specification of the source (SysML) metamodel using eMSL. For each class of the metamodel, there is a (nested) block containing attribute definitions and outgoing edges to other classes. The most important UML language features for specifying associations, such as multiplicities, aggregation and composition are supported.

An instance of this metamodel is shown in Fig. 9.14, describing the state machine of Fig. 1.3a as an eMSL model. At compile-time, metamodel-conformance is checked and reported by the Xtext-based editor. In eMoflon::Neo, it is possible to export such models to the graph database and thereby provide them as input to consistency management operations. While this is very convenient for functional testing purposes, we are working on exporting XMI files to the database to improve the compatibility to EMF-based tools.

SysML.msl

```
metamodel SysML {
    Statemachine {
        .name : EString
        -regions(0..*)-> Region
        -ports(0..*)->Port
    }

    Region {
        .name : EString
        -subvertex(0..*)->State
        -transitions(0..*)->Transition
    }

    Port {
        .name : EString
    }

    State {
        .name : EString
    }

    Pseudostate : State {
        .kind : EString
    }

    Transition {
        .name : EString
        -s(1..1)->State
        -t(1..1)->State
        -guards(0..*)->Guard
        -triggers(0..*)->Trigger
        -effects(0..*)->Effect
    }

    Guard {
        .name : EString
        .body : EString
    }

    Trigger {
        .name : EString
        .body : EString
    }

    Effect {
        .name : EString
        .body : EString
    }
}
```

Figure 9.13: SysML metamodel

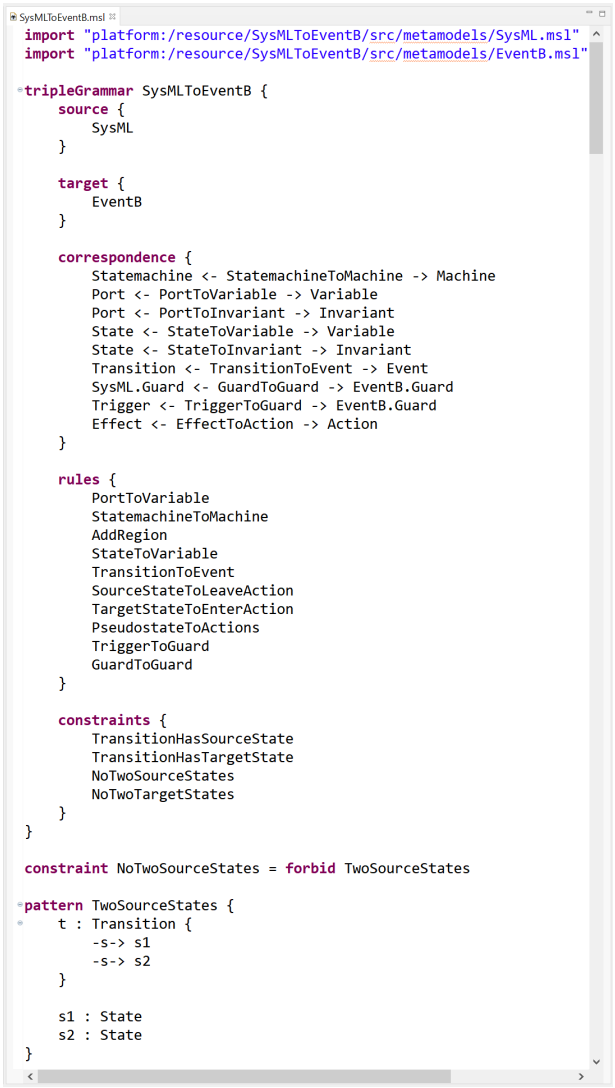
Machine.msl

```
import "platform:/resource/SysMLToEventB/src/metamodels/SysML.msl"

model Machine {
    sm : Statemachine {
        .name : "Machine"
        -regions->main
        -ports->finish
    }

    main : Region {
        .name : "main"
        -subvertex->init_state
        -subvertex->start
        -subvertex->stop
        -transitions->initialisation
        -transitions->completion
    }

    finish : Port {
        .name : "finish"
    }

    init_state : Pseudostate {
        .name : "INIT_STATE"
        .kind : "init"
    }

    start : State {
        .name : "START"
    }

    stop : State {
        .name : "STOP"
    }

    initialisation : Transition {
        .name : "INITIALISATION"
        -s->init_state
        -t->start
        -effects->act1
    }

    completion : Transition {
        .name : "COMPLETION"
        -s->start
        -t->stop
    }

    act1 : Effect {
        .name : "act1"
        .body : "finish := TRUE"
    }
}
```

Figure 9.14: Example instance

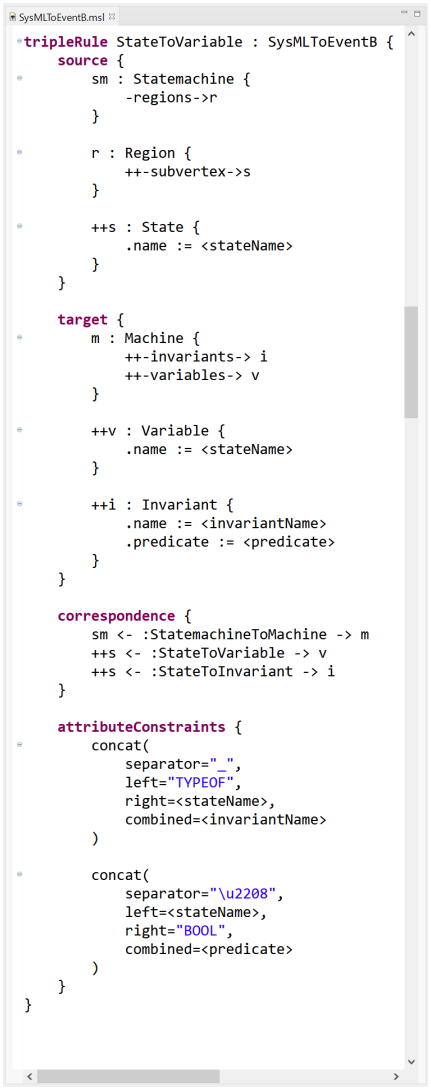
TGGs can be specified in eMSL in a similar way, as shown in Fig. 9.15. Compared to the TGG specification in IBeX, one can observe several similarities, such as the definition of source and target metamodels, as well as correspondence types. In contrast to IBeX, the rule specification is part of the TGG definition, whereby it is possible to spread the textual specification over multiple files to improve readability. Another difference is that constraints can be part of the TGG: On the bottom of Fig. 9.15, the constraint *NoTwoSourceStates* (cf. Fig. 6.2) is expressed in eMSL notation. While the graph pat-

tern *TwoSourceStates* represents a sub-graph that must not occur in the output model (denoted as N in Def. 4.13), a negative constraint is formed out of this pattern via the keyword `forbid`. In a similar manner, patterns can also be attached to metamodels as negative, positive, or implication constraints. As presented in Chap. 6, the consistency management operations ensure that the output models fulfil all constraints of the TGG.



```
SysMLToEventB.msl
import "platform:/resource/SysMLToEventB/src/metamodels/SysML.msl"
import "platform:/resource/SysMLToEventB/src/metamodels/EventB.msl"

tripleGrammar SysMLToEventB {
    source {
        SysML
    }

    target {
        EventB
    }

    correspondence {
        Statemachine <- StatemachineToMachine -> Machine
        Port <- PortToVariable -> Variable
        Port <- PortToInvariant -> Invariant
        State <- StateToVariable -> Variable
        State <- StateToInvariant -> Invariant
        Transition <- TransitionToEvent -> Event
        SysML.Guard <- GuardToGuard -> EventB.Guard
        Trigger <- TriggerToGuard -> EventB.Guard
        Effect <- EffectToAction -> Action
    }

    rules {
        PortToVariable
        StatemachineToMachine
        AddRegion
        StateToVariable
        TransitionToEvent
        SourceStateToLeaveAction
        TargetStateToEnterAction
        PseudostateToActions
        TriggerToGuard
        GuardToGuard
    }

    constraints {
        TransitionHasSourceState
        TransitionHasTargetState
        NoTwoSourceStates
        NoTwoTargetStates
    }
}

constraint NoTwoSourceStates = forbid TwoSourceStates

pattern TwoSourceStates {
    t : Transition {
        -s-> s1
        -s-> s2
    }

    s1 : State
    s2 : State
}
```

Figure 9.15: TGG: SysMLToEventB



```
SysMLToEventB.msl
tripleRule StateToVariable : SysMLToEventB {
    source {
        sm : Statemachine {
            -regions->r
        }

        r : Region {
            ++-subvertex->s
        }

        ++s : State {
            .name := <stateName>
        }
    }

    target {
        m : Machine {
            ++-invariants-> i
            ++-variables-> v
        }

        ++v : Variable {
            .name := <stateName>
        }

        ++i : Invariant {
            .name := <invariantName>
            .predicate := <predicate>
        }
    }

    correspondence {
        sm <- :StatemachineToMachine -> m
        ++s <- :StateToVariable -> v
        ++s <- :StateToInvariant -> i
    }

    attributeConstraints {
        concat(
            separator="_",
            left="TYPEOF",
            right=<stateName>,
            combined=<invariantName>
        )

        concat(
            separator="\u2208",
            left=<stateName>,
            right="BOOL",
            combined=<predicate>
        )
    }
}
```

Figure 9.16: Rule: StateToVariable

The rule *StateToVariable* is depicted in Fig. 9.16. For TGG rules, the eMSL syntax is largely aligned to the textual syntax of IBeX. A main difference is the use of parameter values (`<stateName>`, `<invariantName>` and `<predicate>`) to ease the definition of attribute conditions. Simple checks for equality, such as for the name attribute of the state `s` and the variable `v`, can even be expressed without an explicit condition. The other two attribute conditions, which involve string concatenation, use these parameter values as well. The PlantUML visualisation is again very similar to IBeX and not repeated at this point.

While one can convincingly argue for the advantages of Neo compared to IBeX with respect to flexibility, the intended improvements regarding scalability will be investigated in Sect. 9.6.

9.6 Scalability Analysis

In Sect. 9.3, 9.4 and 9.5, we presented *IBeX* and *Neo* as the two components of the eMoflon tool suite. As scalability was one main motivation for developing Neo, this section's evaluation shall show under which conditions the use of graph databases instead of the EMF framework can be beneficial. A fair comparison is only possible for the operations CO, CC, FWD_OPT and BWD_OPT without graph constraints, because their implementation is largely equal in both tools. While Sect. 7.7 has shown that the concurrent synchronisation approach of Fritsche et al. [FKM+20] clearly outperforms the operation of the hybrid framework, this should rather be attributed to the conceptual background than to the underlying technology. In other words, the differences with respect to runtime performance can be regarded as "the price of fault-tolerance" for concurrent synchronisation.

To compare the scalability of IBeX and Neo, we measured its runtime performance using examples *FamiliesToPersons*, *ClassDiagramToDatabaseSchema* and *CompanyToIT* from the BX example repository[17]. We investigate the following research questions:

RQ1 How does the use of graph databases relate to runtime performance? Are differences for growing (meta-)model sizes or an increasing number of rules observable?

RQ2 Are there differences between the supported operations regarding runtime performance? For which operations is the use of graph databases especially beneficial?

Setup: The three examples were tested for model sizes from 1,000 to 100,000 elements (nodes and edges). We repeated each test run five times with a time-out of 20 minutes and took the median to reduce the effect of outliers. The execution environment for the test runs was a standard notebook with an Intel Core i7 (1.80 GHz), 16GB RAM, and Windows 10 64-bit. eMoflon::Neo was installed based on an Eclipse IDE for Java and DSL Developers, version 2021-03 (4.19.0) with JDK version 13. 4GB RAM were allocated to the JVM running the tests, while 8GB were allocated to Neo4j (version 3.5.8). Gurobi 8.1.1 was used as an ILP solver.

Results: The runtime measurements for the three examples are depicted in Fig. 9.17a - 9.19b for IBeX and Neo. Note that a logarithmic scale is used on both axes to show results for small and large models in the same diagram. While an overview of the experiment was already given in prior work [WA21a], the complete dataset is available online[18].

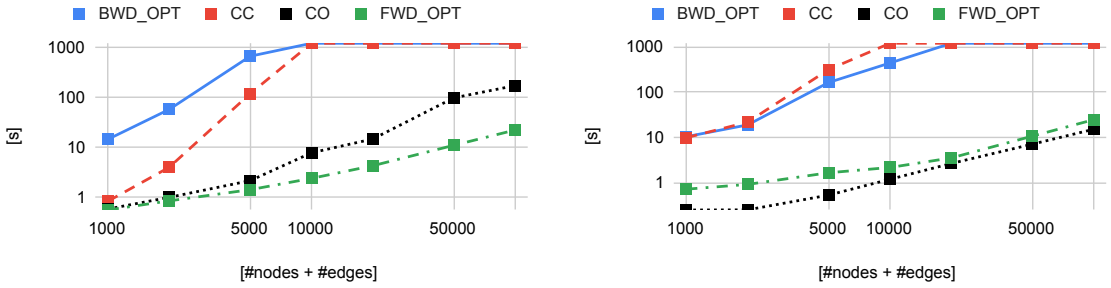
For FamiliesToPersons (Fig. 9.17), IBeX and Neo perform equally well for FWD_OPT, whereas Neo shows better scalability for all other operations. For BWD_OPT and CC, both IBeX (10,000 elements) and Neo (20,000 elements) reached the time-out.

IBeX appears to scale better for ClassDiagramToDatabaseSchema (Fig. 9.18) with the exception of CC on large models. We assume that the linear, hierarchical metamodel structures of this example are advantageous for the pattern matcher of IBeX.

For CompanyToIT (Fig. 9.19), the runtime differences are substantial for all operations except BWD_OPT. For FWD_OPT and CC, IBeX again reaches the time-out earlier than Neo (50,000 and 5,000 elements). Compared to the other examples, CompanyToIT has slightly larger rules and metamodels, and generally a more complex pattern structure.

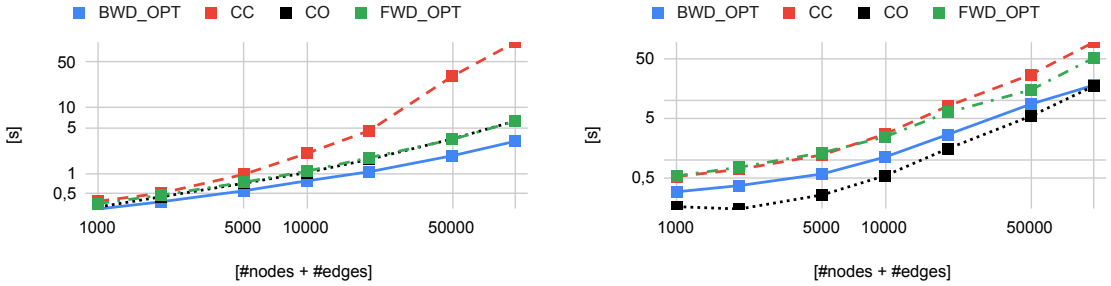
Summary: The results indicate that eMoflon::Neo scales better than eMoflon::IBeX with increasing rule and metamodel complexity, whereas IBeX might show a better performance for simpler TGGs (RQ1). The gain in performance, however, depends more on the nature of the concrete example than on the particular operation (RQ2).

[17] http://bx-community.wikidot.com/examples:home
[18] https://docs.google.com/spreadsheets/d/1ujxPmeCJY7n7-tFh9Ks5qPcBeMB_TUnLhl6tcCgGJQY



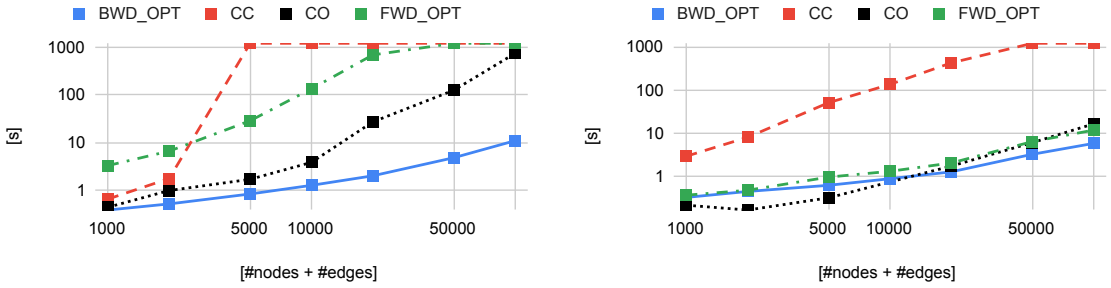
(a) eMoflon::IBeX (b) eMoflon::Neo

Figure 9.17: Runtime measurements: FamiliesToPersons



(a) eMoflon::IBeX (b) eMoflon::Neo

Figure 9.18: Runtime measurements: ClassDiagramToDatabaseSchema



(a) eMoflon::IBeX (b) eMoflon::Neo

Figure 9.19: Runtime measurements: CompanyToIT

Threats to validity: Both tools are under active development and therefore subject to continuous improvement, such that direct comparisons are only valid for a specific point of time. We restricted the comparison to similarly implemented operations, not involving strategies such as (concurrent) model synchronisation, which is implemented very differently in both tools (cf. Sect.7.7). As we have shown that the results strongly depend on the concrete examples, it would be important to test with further realistic (industrial) examples to gain more insights on scalability.

9.7 Teaching MDE with eMoflon

Besides gaining insights about quantitative aspects with respect to runtime performance (cf. Sect. 9.6), we are also interested in getting feedback on eMoflon from the user perspective to further improve the tool. We conducted an empirical study with 40 students of an undergraduate, introductory course[19] on model-based software development at Paderborn University. An online questionnaire[20] was designed as a mix of quantitative multiple choice and qualitative open questions, which refer to the IBeX-GT component. In particular, the following research questions were investigated:

RQ1 How do users perceive the editing experience provided by a combination of textual concrete syntax and coupled, read-only, partial visualisation?

RQ2 How do users judge the ease with which rules and patterns can be mixed with Java code and integrated in Java applications?

RQ3 How do users rate the relative importance of different language features?

RQ4 Do users appreciate our current documentation as a set of handbooks?

An overview of the results was already presented in prior work [WARV19]. For further details of this empirical experiment, the interested reader is referred to Robrecht [Rob18]. Figure 9.20 depicts an overview of the results from the quantitative part of the survey. All detailed results of the entire experiment are available online.[21] To investigate our four questions, we formulated 23 multiple choice questions divided up into 5 categories. A scale of 1 to 5 was used for each question with 1 for "low" and 5 for "high".

The first category *Prior Experience* was used to characterise our participants: programmers with sufficient experience with a modern object-oriented language, moderate prior experience with Eclipse, but with little to no prior experience with MDE, GT, or any visual language at all.

Regarding (RQ1), our results indicate that many users find the textual concrete syntax acceptable, and even more appreciate the visualisation. While some users criticise the fact that the visualisation is read-only, our results show that it is probably not worth developing a visual editor, especially considering that most users are satisfied with the mix of a textual syntax and a coupled visualisation that adjusts dynamically to and focusses only on the current selection in the textual editor. By using the Xtext framework, our results show that it is possible to provide adequate validation errors and other usability features.

Concerning (RQ2), our results indicate that while the expressiveness of the rule and pattern language is judged to be high enough, most students are uncertain if and how IBeX can be used in real-world applications. Regarding the integration of Java and GT code, being able to switch seamlessly between Java and GT files was judged to be acceptable but in need of improvement. The automatically generated JavaDoc for the API is appreciated by only a few users; most are neutral and apparently do not see the direct benefit of this.

Regarding (RQ3), most students regard (positive and negative) application conditions and attribute conditions to be most important, followed by support for modularity (rule refinement), and complex application conditions (combination of conditions via conjunction (`&&`) and disjunction(`||`)). For many students, it is hard to appreciate the potential of incrementality and reactive programming, though.

[19] https://mde-lab-sessions.github.io/running-example-for-lecture
[20] https://docs.google.com/forms/d/1r5pgkTv0CcvTQoqHlUuHcQqULl6A8CmbHgpt961BtHU
[21] https://docs.google.com/spreadsheets/d/1L4UlFPNmM2a4l-fF892FQPzos3R9Mz7-MQQfQ1BipB8

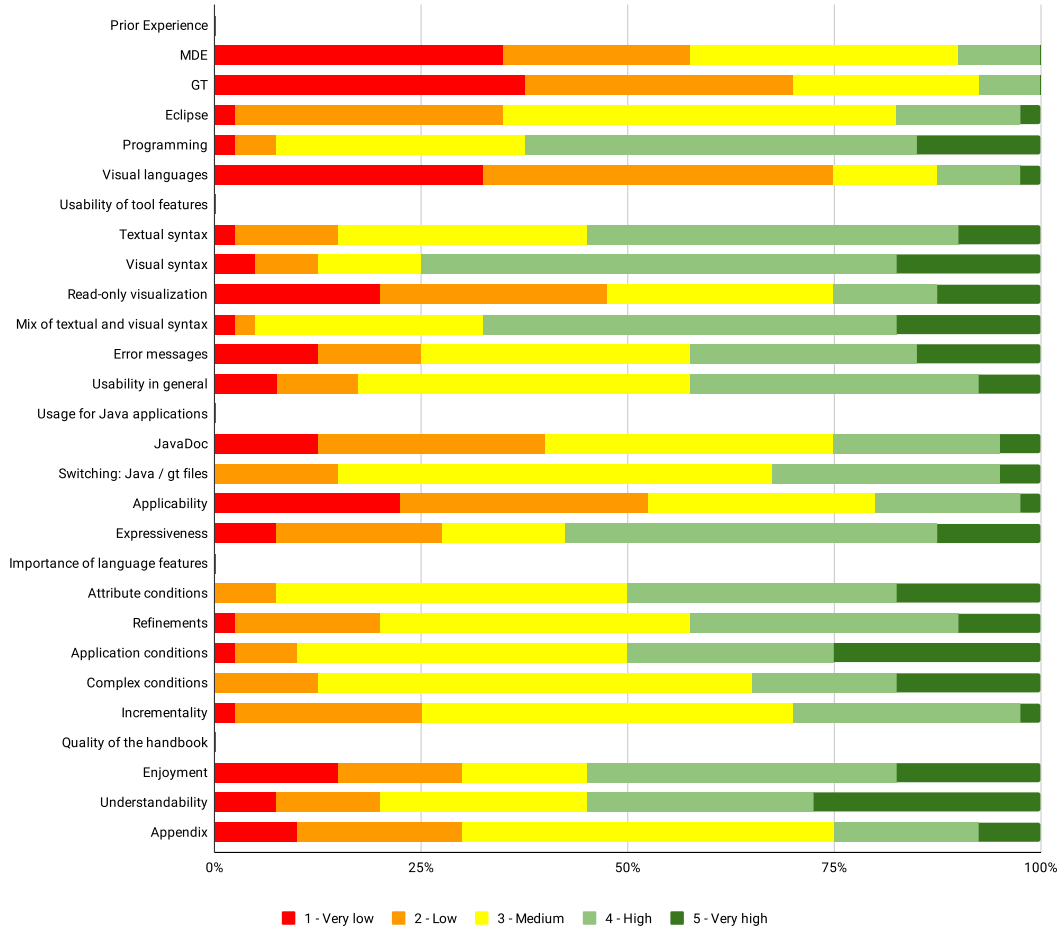


Figure 9.20: Student feedback

Finally, our handbook[22] (RQ4) received mostly positive feedback, with many students preferring the example-driven, tutorial-like explanation to the complete, but reference-like appendix.

9.8 Summary and Discussion

The fault-tolerant, hybrid approach to consistency management, which was presented in Chap. 5 – 8 of this thesis, is fully implemented as part of the eMoflon tool suite, of which an overview was given in this chapter. The tool suite consists of the components IBeX and Neo, while the back end of IBeX can be further subdivided into a layer for GT, and a layer for TGGs that builds upon it. IBeX-GT has a special focus on supporting reactive programming via the incrementality of its underlying and exchangeable graph pattern matching engine. IBeX-TGG supports numerous consistency management operations including forward and backward transformation, (concurrent) model synchronisation, and consistency checking with and without correspondence links. While the transformation and consistency checking operations follow the hybrid approach of Chap. 5, the two

[22] https://bit.ly/3qfDhCr, https://bit.ly/3223koe

synchronisation operations are implemented in a greedy (and therefore fault-intolerant) manner. Neo as the latest addition to the eMoflon tool suite leverages Neo4j as a graph database for all runtime models. In this component, all operations are implementations of the hybrid fault-tolerant framework.

We discussed our most important requirements for fault-tolerant tool support, i. e., incrementality, scalability, flexibility and modularity, and could not identify another tool that sufficiently addresses these requirements already. The hybrid approach was therefore implemented in eMoflon as a conceptually and technically uniform interpretative algorithm, which leverages and suitably combines an incremental graph pattern matcher and an ILP solver.

Concerning the usability of the tool, especially for teaching, ambivalent but still pleasant feedback for IBeX was received in an empirical study with 40 undergraduate students. As a further advancement of the front-end, a novel specification language eMSL was developed for Neo that uniformly supports (meta-)modelling, patterns, constraints, rules and TGGs. A comparison of the front-ends of IBeX and Neo in a further empirical experiment is left to future work.

Our performance comparison of IBeX and Neo indicates that the use of graph databases instead of the EMF framework allows for improved scalability, especially for more complex examples and for growing model sizes. As our evaluation was restricted to a comparison with respect to runtime performance, we plan to investigate on the benefits regarding flexibility and potential drawbacks of storing (meta-)models in an external graph database. Nonetheless, we are working on improving the scalability of both components of the tool suite. For IBeX, this requires understanding how best to structure the generated pattern invocation networks passed to the incremental pattern matcher, taking the nature of the involved metamodels, the size of the models, and the size and connectivity of all patterns into account. For Neo, room for improvement consists in understanding and leveraging caching mechanisms of the database, and optimising the Cypher queries by implementing re-use mechanisms.

With respect to support for fault-tolerance in MDE-tools, it became apparent that the conceptual framework of this thesis could be entirely implemented in Neo. With the partial implementation in IBeX, we were able to show that conventional and fault-tolerant operations can co-exist in the same tool. While the EMF framework places obstacles on the dynamic adaptation of models and metamodels, the feasibility of implementing fault-tolerant operations based on EMF could be underpinned. As future work, we plan to improve the interoperability between Neo and EMF-based tools regarding model and metamodel exchange. Regarding the implementation in Neo, the flexibility of graph databases can be further leveraged for co-evolving models and metamodels.

An important aspect that remains unaddressed up to here is user interaction: The described operations are fully automated, whereas involving the user in controversial decisions was listed as a requirement in Sect. 1.1. To improve the situation, the MDE-debugging component “VICToRy” was developed, which will be presented in Chap. 10. Furthermore, two examples of industrial use cases shall be given in Chap. 11 and 12 to demonstrate the applicability of the eMoflon tool suite in practice.

10 The VICToRy Debugger

The eMoflon tool suite is capable of supporting all operations of the fault-tolerant hybrid framework, leveraging the power of external graph pattern matchers and ILP solvers. All operations are carried out completely in the background, though, which contradicts the requirement of involving the user in controversial decisions. As a step towards improving this situation, we present VICToRy, a debugger for model generation and transformation based on TGGs. In addition to a fine-grained, step-by-step, interactive visualisation, VICToRy enables the user to actively explore and choose between multiple valid rule applications thus improving control and understanding.

The chapter is structured as follows: After a brief introduction (Sect. 10.1), VICToRy is compared to existing MDE debuggers in Sect. 10.2. An architectural overview is provided in Sect. 10.3, before Sect. 10.4 introduces the breakpoint concept. Section 10.5 provides an overview of the debugger's front-end. The UI prototype of a concurrent synchronisation component is presented is Sect. 10.6. The results of empirical studies with both MDE experts and novices are summarised in Sect. 10.7, before Sect. 10.8 concludes the chapter.

10.1 Introduction and Motivation

In Chap. 9, an implementation of the hybrid approach as part of the eMoflon tool suite was demonstrated, including operations for model transformation, consistency checking, and (concurrent) synchronisation. The supported operations, however, run completely in the background with only input and output made visible to the user, arguably reducing both understandability and controllability. One of the requirements for fault-tolerant consistency management approaches, according to Sect. 1.1, is that the user should be involved in controversial decisions, i. e., decisions that cannot be made only based on the consistency relation specification. Even for uncontroversial transformations, we have observed that novice users are unable to fully understand how TGG tools - viewed as black-boxes - determine a specific result.

While debugging facilities are a handy feature for transformation and consistency checking tasks, Chap. 7 and 8 have shown that finding satisfactory solutions after concurrent updates is hardly possible without involving the integration expert. On the one hand, the hybrid approach is inherently fault-tolerant and offers configuration parameters to incorporate user preferences, but does not scale well enough for larger models. Furthermore, the outcome of the optimisation process is rather opaque without in-depth knowledge of the underlying technology. On the other hand, the approach of Fritsche et al. [FKM+20] exhibits a good runtime behaviour, but relies on consistent input models and expects the user to predefine a synchronisation strategy for the entire process.

To address these open challenges, the *VICToRy debugger*[1] was developed as an add-on component for consistency management tools, which enables the user to step-wise execute consistency management tasks. While for most GPL debuggers, a step covers a single instruction, a TGG rule application is this "smallest executable unit" for the VICToRy

[1] github.com/eMoflon/emoflon-victory

debugger. It presents possible operational rules including their concrete application contexts to the user, as well as a history of the involved models as they evolve during the transformation process. Additionally, the user can inspect and choose a valid rule application at each time step, or decide to resume the automated process in the background. In combination with a sophisticated breakpoint concept, it is possible to efficiently debug larger model transformations. This is what distinguishes the VICToRy debugger from all MDE debuggers we are aware of, which focus on making the pattern matching process and the generation of nodes and edges transparent for the user. In order to support the integration expert restoring consistency after concurrent updates, a concurrent synchronisation component was carefully designed and prototypically implemented.

While some TGG tools provide basic debugging functionality for the transformation process (cf. Sect. 10.2), none of them enable the user to track let alone influence the choice of rule applications. VICToRy is currently integrated into the eMoflon tool suite, but can be potentially connected to other Java-based TGG and even general graph transformation tools via the defined interfaces. This means that existing and future tools can be enriched with debugging facilities to increase user involvement and understanding in the transformation process.

10.2 Related MDE Debuggers

Several approaches to debugging in MDE have been proposed, including fundamental concepts, debugging DSL code, and debuggers for non-deterministic approaches.

Mierlo et al. describe a stepping semantics for debugging in MDE with four levels of different granularity [MTV18]. The proposed approach is, however, conceptual and does not provide an implementation to the best of our knowledge. A debugger for Petri nets is based on Modelverse and supports basic functionality including breakpoints known from GPL debuggers [MV17]. The prototype is planned to be extended to support model transformations as well.

A wide range of facilities for DSL debugging is presented in previous work. Omniscient debugging - in contrast to stepwise execution - provides the user with enhanced navigation and exploration features such as reverting execution steps at runtime, impacting performance and scalability. Therefore, approaches are often tailored to rather small instances [CESG17] or specific use cases, such as xDSMLs (a subset of DSLs) [BCC+15]. Lindeman et al. propose a declaratively defined debugger for DSLs [LKV11]. The approach was integrated into the Spoofax language workbench and evaluated by case studies involving the textual DSLs StrategoTL and WebDSL. However, several limitations are mentioned for debugging modelling languages and model transformations. Laurent et al. extended the foundational UML (fUML) by debugging facilities [LBG13]. While the approach is a tool-independent add-on, it considers only the execution of models complying to the fUML standard.

For debugging rule-based systems, Tichy et al. sketch how to execute debugging steps for graph transformations, taking the tool Henshin as an example [TBK17]. In contrast to our approach, the debugging of rule applications is much more detailed and takes the matching process into account as well, whereas an implementation is not described. Similarly, Jukss et al. use graph transformations as an underlying formalism for a debugger integrated into AToMPM [JVV17]. The approach focusses on a fine-grained inspection of the rule application process, whereas the user is not enabled to choose between multiple possible rule applications. For algebraic graph transformation, the tool AGG [RET11] provides a mode for stepwise execution of graph transformations. Rule and match can be chosen by

the user in each step, while it is neither clear which rules are applicable in the current state, nor a protocol of previous rule applications is provided.

Furthermore, multiple TGG tools (cf. Sect. 9.2) have been extended by debugging facilities, which appear to be limited in several respects, though. A concept for debugging TGGs at different levels was introduced to the TGG Interpreter by Rieke [Rie15]. The debugging facilities are, however, tightly interwoven with the specific tool and several open challenges for practical use are mentioned. For MoTE, a monitor is implemented which allows to stepwise execute model transformations [GHL14]. However, the user cannot influence the execution order, which is determined by the order of correspondence nodes in a processing queue and their respective types. A debugging mode is implemented for EMorF as well, but both a detailed description and the tool itself are currently not available. For all other TGG-based tools, debugging functionality is missing to the best of our knowledge.

Besides these rule-based approaches, debugging plays an important role in other MDE-related fields as well. Proposed concepts include work on dynamic meta modelling [BSE10], Discrete Event System Specifications (DEVSs) [MTV17], and story diagrams [KHW12], which are each tailored to a specific tool and use case, though. The tool TETRA Box is based on PaMoMo and involves white-box testing of transformation languages by symbolic execution of model transformations [SKW+13], which is independent of the underlying transformation language but not yet tested with realistic examples. SyVOLT localises errors in the input based on igraph and the T-Core framework [OLVV18], while the focus of debugging is set on detecting reasons for contract violations rather than on the transformation process. Ferdjoukh et al. localise faults in metamodel design based on static analyses and implemented their approach in TIWIZI and GRIMM [FM18], whereas model transformations are not taken into account.

In total, no MDE debugger we are aware of provides a step-wise visualisation of large-scale model transformations, in which the user is enabled to decide between multiple possible rule applications.

10.3 Architecture

VICToRy can be connected to different Java-based TGG tools by implementing an interface for transferring data between the debugger and the respective tool. An overview of this interface is depicted in Fig. 10.1. Compositions have multiplicities of `1` and `0..*`, if not stated otherwise. The central component of this interface is the `DataPackage` class, which bundles the data that is transferred between VICToRy and the TGG tool. A `DataPackage` contains all relevant `Rules`, `Matches`, and `Rule Applications`. Multiple `Matches` can be determined for the same `Rule`. Furthermore, a `RuleApplication` object is created when a `Rule` is applied for a concrete `Match`.

These three classes are represented as `Graphs` consisting of `Nodes` and `Edges`. There exists a mapping from each `Edge` to a source and a target `Node`, reflecting the categorical approach to graph transformation (cf. Def. 3.1). The `Nodes` have a set of `Attributes`, and a `Domain`, i. e., a marking that indicates whether they belong to the source or target model. For `Edges`, the domain can be determined from their source and target nodes: If both `Nodes` are part of the source (target) model, the `Edge` also belongs to the source (target) model. If `Edges` connect `Nodes` of different domains, VICToRy considers the `Edge` as a correspondence link (`EdgeType` “CORR”). Within a rule or a graph instance, `Edges` have the `EdgeType` “NORMAL”. Mappings between rules and graph instances are

represented as `Edges` of type "MATCH". Finally, each element has an `Action`, indicating whether the element is created, translated (marked), or required as context by the rule.

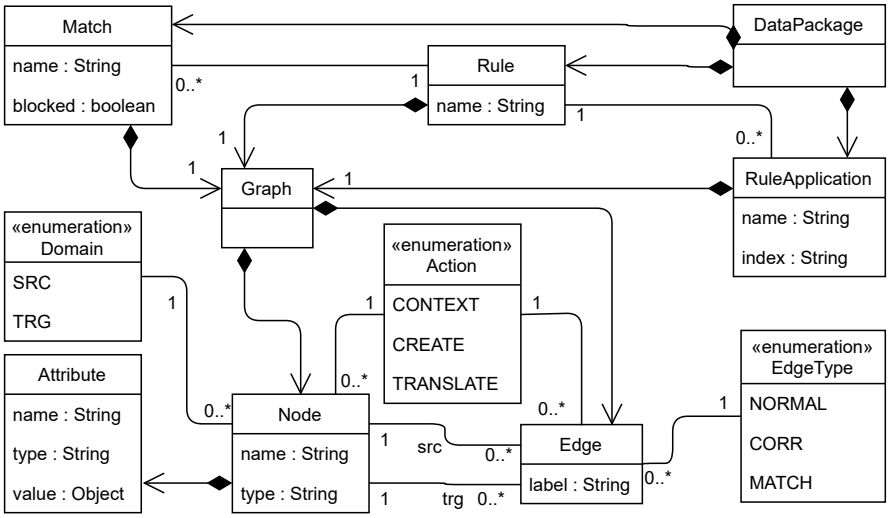


Figure 10.1: Data exchange with VICToRy

The component diagram in Fig. 10.2 describes how the debugger has been embedded into the eMoflon tool suite, and can potentially be connected to other Java-based TGG tools. Currently, both IBeX (Sect. 9.4) and Neo (Sect. 9.5) implement the interface to the debugger. The VICToRy adapter is tool-specific and needs to be implemented in order to connect the debugger to a TGG tool. It is responsible for providing both the debugger and the TGG tool with the required information, as shown in Fig. 10.1. The debugger itself consists of a controller that delegates user commands to the adapter, and in turn receives updated information about new matches and the current state of the models. All relevant information is made available to the user via the UI, whereas the breakpoint manager is responsible for checking breakpoint conditions. In the following, an overview of the breakpoint concept (Sect. 10.4) and the UI (Sect. 10.5) will be provided.

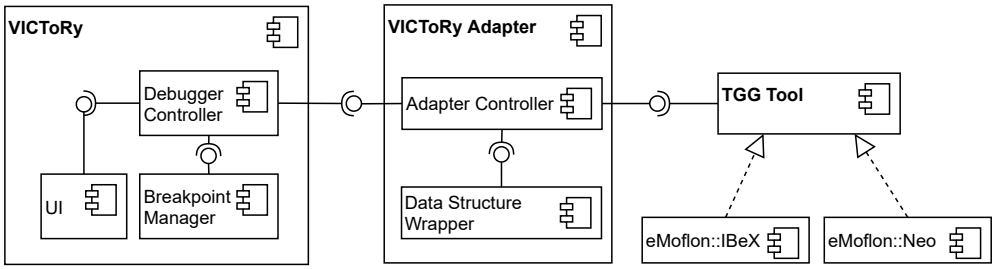


Figure 10.2: Integrating VICToRy into the eMoflon tool suite

10.4 Breakpoint Concept

While performing a consistency management task with VICToRy, the tool switches between the two modi `RUN` and `BREAK`, as depicted in Fig. 10.3. In the `RUN` mode, possible matches for rules are collected and one of them is chosen to be applied. In case of multiple options, rule applications are chosen according to a configurable component (e. g., at random in the simplest case) without user interaction, which is the usual work-flow for model

transformation tools. This procedure is repeated until no further matches can be found (leading to the termination of the process) or until a *breakpoint* is reached. In the latter case, the tool switches to the BREAK mode, where the VICToRy UI is visible and each rule application requires a user interaction: Either the user lets the tool choose the next rule application, or selects a rule application manually from the list of all options. To return to the RUN mode, the user resumes the automated choice of rule applications by a corresponding UI command. This behaviour is similar to debugging concepts in contemporary IDEs, but without the possibility of stepping *into* a rule application.

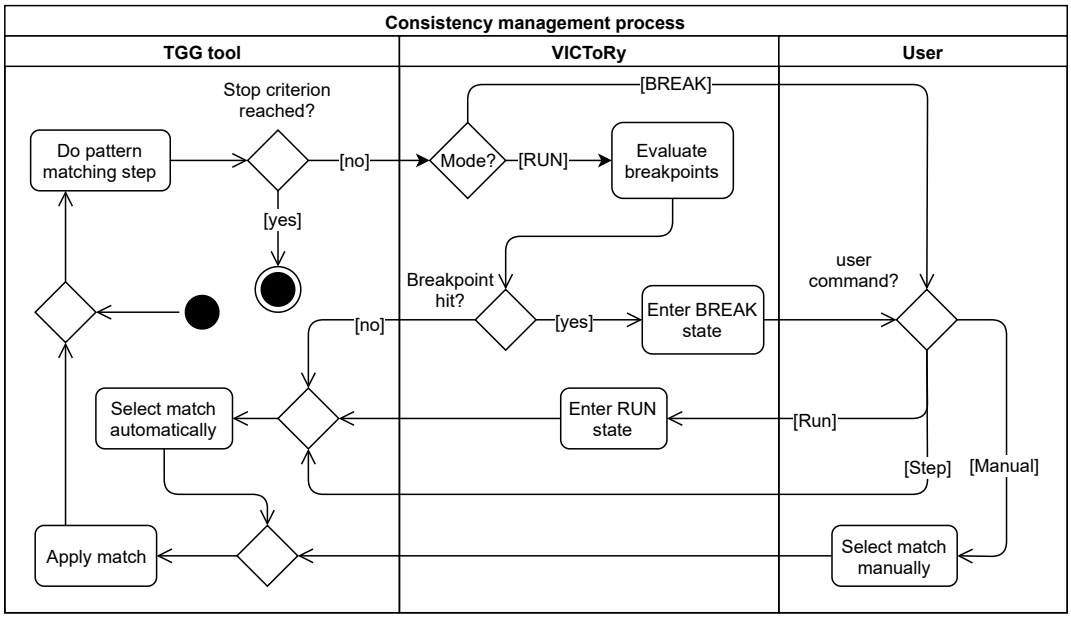


Figure 10.3: Breakpoint concept of the VICToRy debugger

The implementation of the breakpoint concept follows a slightly more complex work-flow, as there are multiple *evaluation times* for breakpoints: A breakpoint can be hit either after the pattern matching step (as shown in Fig. 10.3), after the match selection, or after the actual rule application. While the first and third evaluation time mainly differ in their presentation, the second time enables the user to, e. g., revert an automated rule application step and select another option manually instead. As the handling of breakpoints is equal for all evaluation times, the diagram was simplified to preserve readability.

While the previously described work-flow applies to all breakpoints of the VICToRy debugger, there are multiple types of breakpoints that serve different purposes. They can be subdivided into *model breakpoints*, *match breakpoints*, and *combined breakpoints*, which will be explained in the remainder of this section.

Model Breakpoints

The break conditions of model breakpoints solely depend on the current state of the model instance at hand. While the model size breakpoint is part of the implementation, there is only a conceptual idea for the pattern breakpoint.

- **Model size breakpoint**: This breakpoint type counts the number of nodes in the triple. As soon as a predefined size is reached, the breakpoint is hit. The intention behind this type is to pause the transformation process at some point to inspect the intermediate result.

- **Pattern breakpoint**: Breakpoints of this type are hit as soon as a specified pattern can be matched on the host graph at least once. Especially for faults related to domain constraints, this breakpoint type is useful as graph patterns build the basis for positive, negative and implication constraints in eMoflon (cf. Sect. 9.5).

Match Breakpoints

Match breakpoints refer to the current match to be applied, i. e., the condition rather depends on the next transformation step than on the entire model instance. All match breakpoints except the node breakpoint are implemented for the VICToRy debugger.

- **Rule name breakpoint**: With this breakpoint type, the transformation process breaks each time a predefined rule (identified by its name) is applied. This type is handy in cases where the user assumes that a the definition of a particular rule is faulty.
- **Number of matches breakpoint**: For a predefined number n, a breakpoint of this type is hit if at least n matches are collected. The counted matches can be restricted to one or multiple rules.
- **Element type breakpoint**: Every time an element of a specific type is created, the breakpoint condition is fulfilled. Element types of all three models can be used.
- **Attribute condition breakpoint**: Attribute conditions were introduced as a TGG language feature in Sect. 4.4, but this breakpoint type is not restricted to conditions that are attached to rules of the TGG at hand. Any boolean expression that can be defined using the metamodel attributes and Java standard language features can be expressed, making this breakpoint type especially powerful.
- **Node breakpoint**: Finally, the breakpoint concept involves a breakpoint type that pauses the transformation process as soon as a particular node, identified by its ID, is part of a match. It can be used in situations for which it is probable that a fault occurs when a specific node is created or translated.

Combined Breakpoints

Combined breakpoints can be formed out of all previously described breakpoint types (also denoted as “atomic” breakpoints). In a combined breakpoint, both atomic and combined breakpoints can be connected with AND, OR, and NOT, such that a propositional logic over breakpoints is defined. Considering a breakpoint as a boolean variable that is true if and only if the breakpoint condition is fulfilled, the combined breakpoint is hit as soon as the formed expression is true.

The conceptual introduction to the architecture and the breakpoint concept of the VICToRy debugger is complemented with a brief presentation of the UI in Sect. 10.5.

10.5 An Overview of the User Interface

This section provides an overview of features of VICToRy from the UI perspective, that can help novice users explore an unknown TGG. Besides the main window, a menu for defining breakpoints, and the concurrent synchronisation component are presented.

Configurable Visualisation of Rules and Rule Applications

To understand the effects of a rule application on a concrete model, it is essential to visualise both the rule and the resulting model changes at runtime. VICToRy supports both features via its visualisation section, that is shown in the right part of Fig. 10.4. It shows the visualisation of a model triple resulting from applications of the rules *StatemachineToMachine* and *PortToVariable*. Following the colour scheme of eMoflon, the background colour of source model elements is peach, while target model elements have a rose background. Correspondences are represented as dashed black lines. The visual syntax for rules is equal in VICToRy and eMoflon, and therefore not repeated at this point. The visualisation of rules and the resulting triples is based on PlantUML and is generated automatically on rule and match selection, which will be explained later in the course of this section. Editing rules is only possible in the underlying TGG tool, meaning that rules cannot be adapted at runtime.

To cope with a wide range of TGG rule sizes, model sizes, and the varying proficiency of users, it is crucial to be able to configure the visualisation. Via a pop-up menu that is depicted in the middle of Fig. 10.4, the user has a range of configuration options (available via clicking the "User Options" button):

- **Choice of displayed elements**: For each domain (source, target, correspondence), the user can hide the respective elements. For rules, it is also possible to display only context elements and thus focus on the structure required for a match of that rule on the model instance.

- **Abbreviation of labels**: For nodes, edges and correspondences, it is possible to display the labels completely, in an abbreviated form containing the first and last three letters, or not at all.

- **Neighbourhood of matches**: As models of realistic size can become too large to be completely displayed within the debugger, only the match of a selected rule application and a configurable neighbourhood of this match is displayed. The distance of a node to the match is defined as the shortest path from this node to any node contained in the match; nodes in the match itself are assigned a distance of 0. The k-neighbourhood of a match contains all nodes with a distance of at most $k \in [0; 3]$.

Explorable and Interactive Overview of Applied Rules

Changes in the visualisation can be triggered by selecting rules or *potential* rule applications from the rule section (top left of Fig. 10.4) or *actual* rule applications from the protocol section (bottom left of Fig. 10.4).

The rules section provides an overview of all rules of the TGG. Our example TGG *SysMLToEventB* consists of 14 rules[2], which are depicted as a list. For each rule in the list, the number of available matches in the current model and the number of applied matches are displayed together with the name of the rule. Rules with a dark grey background are not applicable in the current state of the model, whereas rules with a white background have at least one applicable match. This provides a quick overview and is useful for TGGs with a large number of rules. Furthermore, rules that have never been applicable are crossed out, providing a quick visual indication of rules that might be problematic.

[2]Due to technical reasons, the implementeed version of the TGG slightly differs from the running example of this thesis, which only consists of ten rules.

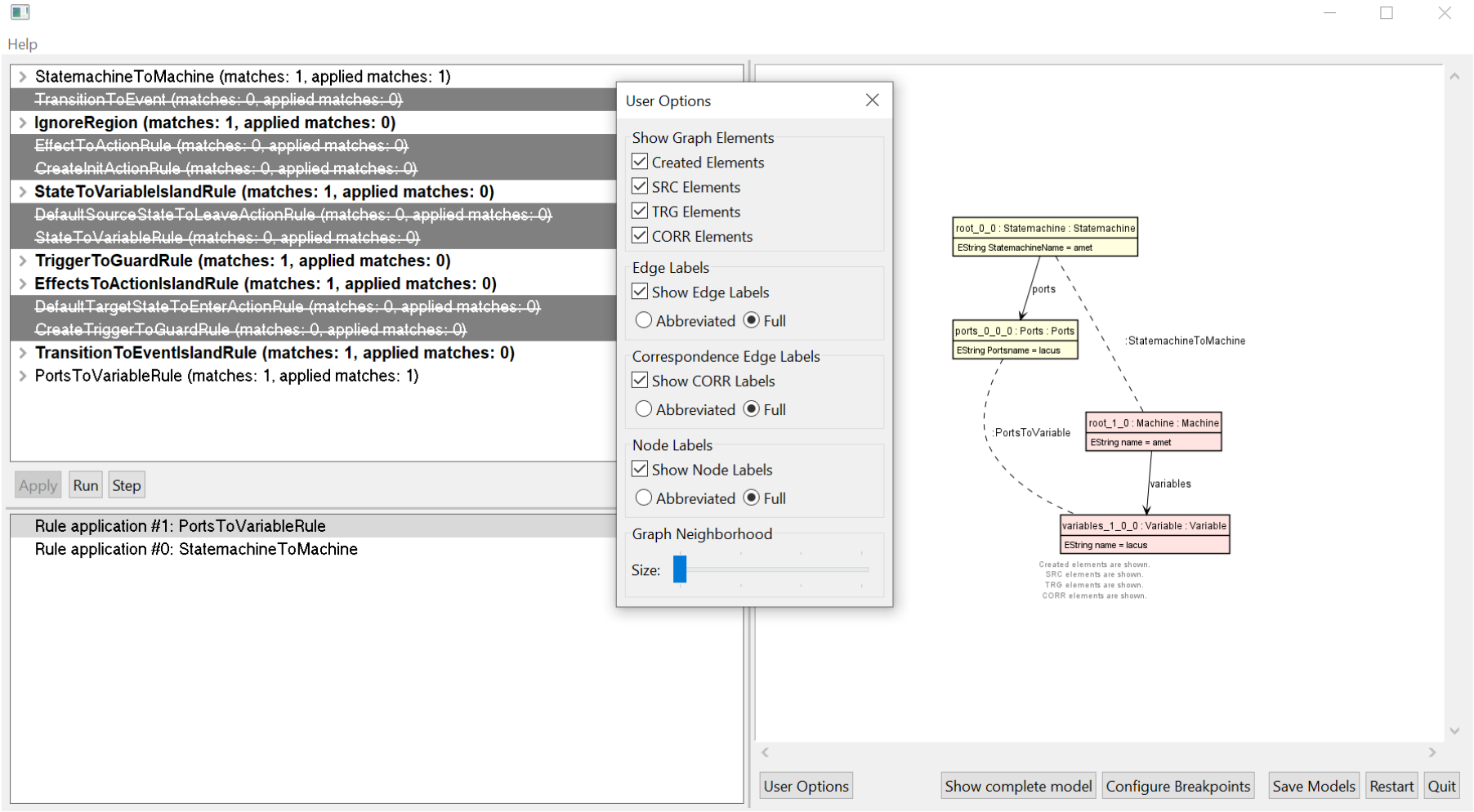


Figure 10.4: Visualising rules and matches

All matches of a rule can be viewed as sub-entries by expanding the corresponding rule entry in the list. When selecting a rule from this list, it is visualised in the right part of the UI. To apply a rule, the user can either double-click on a particular match, select the match and press the "Apply" button, or simply double-click the rule to apply a random match of this rule. The buttons "Run" and "Step" instruct the debugger to select the next match automatically (cf. Fig. 10.3). The user command is delegated to the connected TGG tool, which must handle the actual rule application. As soon as VICToRy receives a response, the UI is updated to reflect the new state of the model and available matches.

The VICToRy debugger provides traceability information by keeping track of all previous rule applications. This sequence of rule applications is referred to as the (transformation) protocol. For each protocol entry, the name of the rule as well as a unique ID for the rule application is displayed (cf. bottom left of Fig. 10.4). If a protocol entry is selected, the state of the model as created by all rule applications up to and including the selected one is displayed with a configurable neighbourhood. It is also possible to select multiple entries: the respective rule applications are then combined into a single step and visualised accordingly.

On the bottom right of Fig. 10.4, several buttons provide the user with additional functionality. The "Show complete model" button resets the selection in the left part of the screen and visualises the entire triple instance. The three models in their current state can be stored as XMI files to continue the debugging process later ("Save models"). It is also possible to start the transformation process from scratch again ("Restart") or to stop the process completely ("Quit"). Finally, breakpoints can be defined on a separate screen, which is explained in the following.

Breakpoint Menu

The definition of breakpoints (cf. Sect. 10.4) for the debugging process is possible via a pop-up menu that can be opened by selecting the "Configure Breakpoints" button on the main window. A list of breakpoints of different types is depicted in Fig. 10.5 and 10.6. The colour scheme indicates that the model size breakpoint (in grey) is deactivated, whereas the number of matches breakpoint (in green) is evaluated after the pattern matching step. The combined breakpoint (in black) is evaluated also after the automatic selection step. Below the list of breakpoints, further options enable the user to configure the breakpoint. The options differ depending on the breakpoint type, such that the UI is dynamically adapted to the type of the selected breakpoint.

In Fig. 10.5, an element type breakpoint as part of the combined breakpoint is selected. The element type can be chosen via a drop-down menu, currently the type `Trigger` is selected. The evaluation time depends on the configuration of the combined breakpoint, but it is possible to disable the element type breakpoint at this level.

The configuration of the combined breakpoint is depicted in Fig. 10.6. Besides the element type breakpoint of Fig. 10.5, it consists of a rule name breakpoint for the rule *TransitionToEvent*. By using the combination type `AND`, we specify that the combined breakpoint is hit only if the conditions for both sub-breakpoints are fulfilled. Via the radio buttons on the bottom, the user can choose whether the conditions must be fulfilled by the same match, or whether it is sufficient if the conditions are fulfilled by different matches. Combined breakpoints can also be used as part of other combined breakpoints, making it possible to create complex nested structures.

After providing an overview of the debugger's main component, the prototypical implementation of the concurrent synchronisation component is sketched in the following Sect. 10.6.

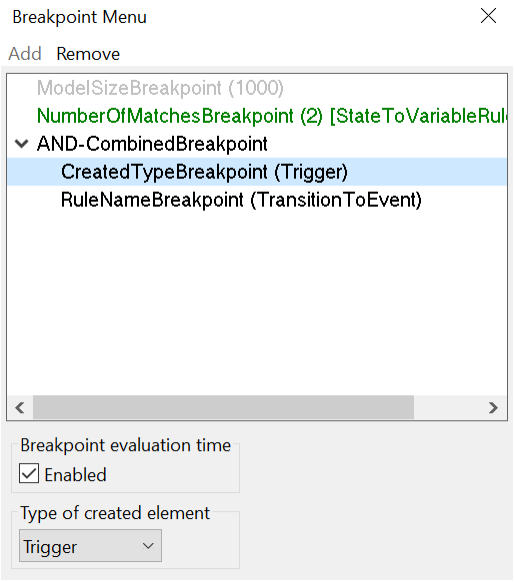


Figure 10.5: Element type breakpoint

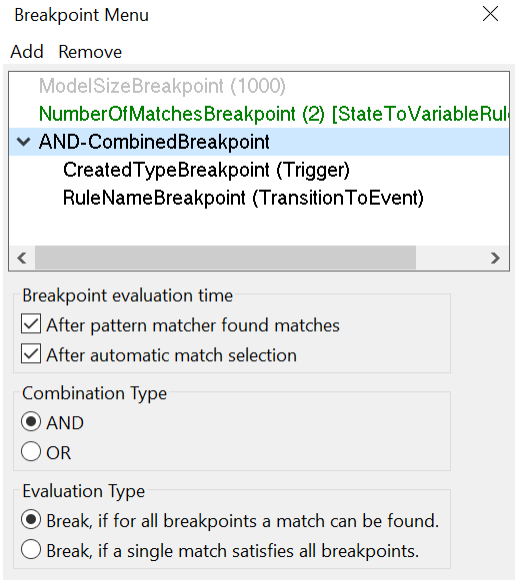


Figure 10.6: Combined breakpoint

10.6 Concurrent Synchronisation Component

The hybrid approach to concurrent model synchronisation as presented in Chap. 7 and 8 computes a solution in a fully automated manner. While this strategy is time-efficient and minimises manual efforts, the acceptance of a consistency management tool would benefit substantially from involving the user, i. e., the integration expert, into the resolution of controversial decisions. To improve the situation, we designed a concurrent synchronisation component, enabling the integration expert to influence the synchronisation process.

As this component – to the best of our knowledge – is the first of its kind, the requirements for such an interactive synchroniser are largely vague. Clarifying the requirements and taking design decisions carefully seemed to be very important for the development process of the concurrent synchronisation component due to the novelty of an interactive model synchroniser. The ARCADIA method [VBNE15], which originates from the systems engineering domain and recently gained popularity for software development processes as well, was used to design the prototype. ARCADIA, which was frequently used in industrial contexts already, promotes a view-point-driven approach and emphasizes a clear distinction between need and solution [Roq16]. In an iterative manner, artefacts of previous phases are re-used in later phases of the development process. Due to time restrictions, we focussed on the *operational analysis* and *system analysis* phases to construct a first UI prototype. In the scope of this thesis, only a standalone UI prototype was developed; the actual integration into the VICToRy debugger is left to future work.

The front-end of the developed UI prototype is depicted in Fig. 10.7. In the middle section, the current state of the model is visualised, which is a representation of the example instance of Fig. 1.3a, that was already used to demonstrate the fully automated synchronisation process of Chap. 7: The SysML engineer has connected the states “START” and “STOP” with a transition, whereas the Event-B expert has deleted the “STOP” state and introduced a self-loop on the “START” state. This leads to two conflicts: First, the “STOP” state cannot be deleted and set as target of the transition at the same time, denoted as *create-delete conflict*. Second, the transition cannot have two targets, which in

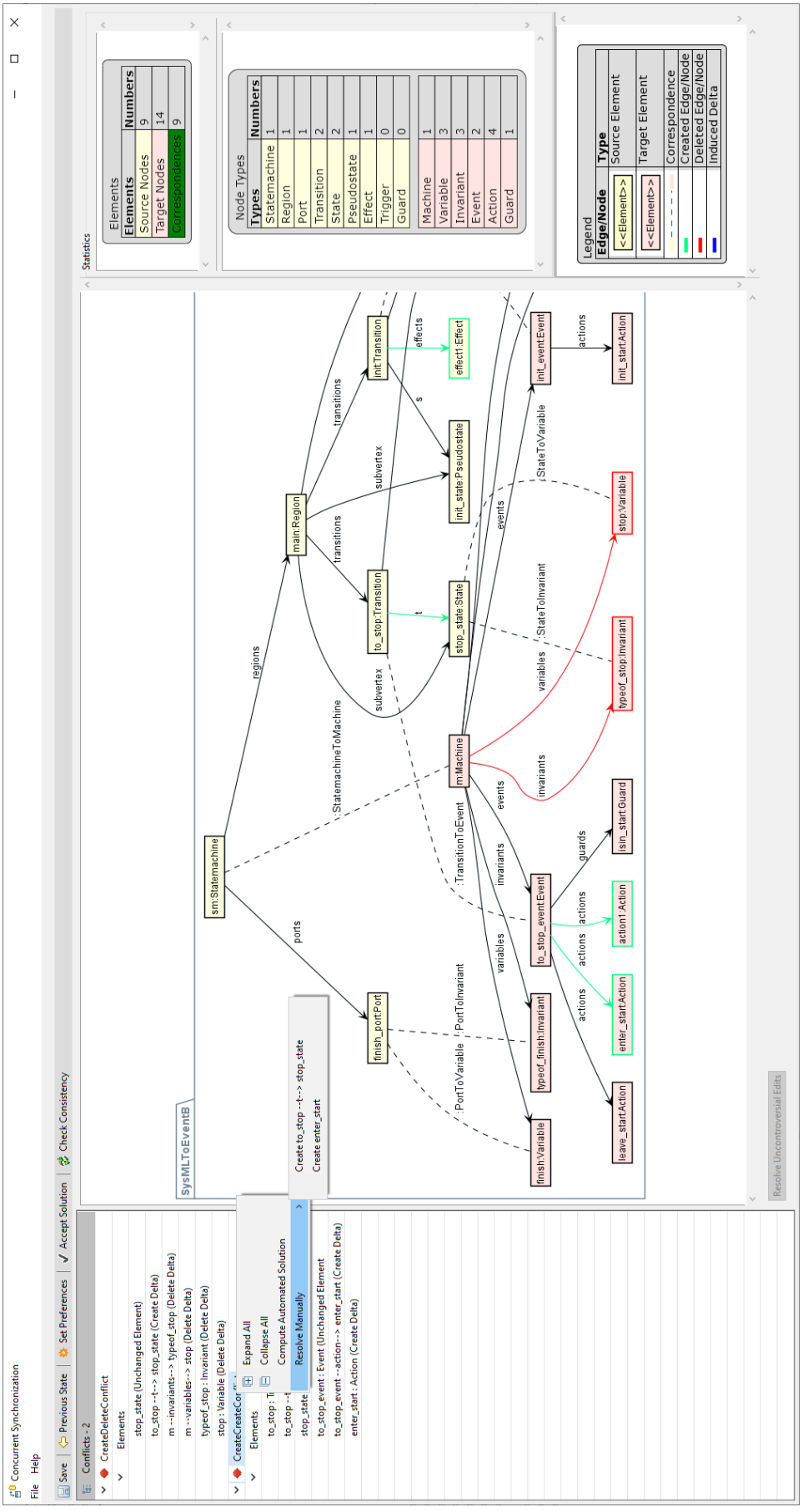


Figure 10.7: Front-end of the concurrent synchronisation component

this case is considered as a *create-create conflict*. For details on the TGG-based detection and classification of conflicts, the interested reader is referred to Fritsche et al. [FKM+20].

The colour scheme for the backgrounds of source and target model elements is the same as for the main component, but the semantics of the elements' frames is different: It indicates whether the respective element is part of a delta structure, as shown in the legend on the bottom right.

On the left-hand side, a list of conflicting changes that have not been resolved yet is presented to the user. For each conflict, all involved elements are listed. Each element of the list, or the entire conflict, can be highlighted in the visualisation via bold lines by selecting it in the conflict list. For each conflict, the user can choose whether it shall be resolved manually or automatically, as shown in the open context menu of Fig. 10.7. For a manual resolution, the user selects one of the generated descriptions. For the concrete example, the concurrent synchronisation component offers the user to resolve the create-create conflict by choosing either of the states as target for the transition. The automatic resolution chooses an option based on a predefined policy (e. g., the solution with the best rating according to Sect. 7.5).

On the right-hand side, some statistical numbers are shown that help the user keeping track of the applied changes. In the tool bar on top of the screen, additional features of the concurrent synchronisation component are depicted: As for the main component, storing the current state of the models on disk ("Save" button) and configuring the visualisation ("Set Preferences") is possible. In case of undesired effects of the last action, the user can step back to the previous state. Furthermore, the user is in charge of accepting the final (conflict-free) solution as outcome of the synchronisation process, and can trigger consistency checks at any point of time.

As part of the VICToRy debugger, the concurrent synchronisation component is independent of the TGG tool in use. The comparison of the component's conflict resolution process and the implementation of the hybrid approach of Chap. 7 in eMoflon::Neo reveals some fundamental open challenges, though: First, there is no explicit definition of conflicts in the hybrid approach. Instead, the construction of the optimisation problem guarantees implicitly that the output triple is consistent, i. e., free of conflicts. Second, the work-flow of the concurrent synchronisation component requires the sequential resolution of conflicts, whereas the hybrid approach encodes all possible options into a single problem definition. Third, with the hybrid approach, it is possible to ensure that the synchronisation result is fully consistent (although the formal proof was left to future work), whereas there is no guarantee that the sequential resolution of conflicts does not produce new conflicts. Finally, no special attention was devoted to the handling of faulty input models while developing the synchronisation component. Further opportunities and challenges will be discussed in Sect. 10.7 based on the feedback we received from experts during the development process.

10.7 Evaluation

In order to qualitatively assess the usefulness of VICToRy for involving the integration expert into consistency management tasks, we conducted an empirical three-part evaluation with potential users with different levels of expertise. This section can only provide a brief summary of the results, the interested reader is referred to a more extensive discussion in prior work [WAC20, Jos21, Sri21]. Our evaluation aims at answering four research questions related to motivational aspects for MDE debuggers:

RQ1 Does VICToRy help to explore and understand a TGG of realistic size?

RQ2 Does VICToRy help to identify faults in rules or in input models?

RQ3 How valuable are breakpoints in different debugging scenarios and for different types of faults compared to debugging without breakpoints?

RQ4 How can a concurrent synchronisation component support the manual resolution of conflicts? Which features should such a component offer?

Case Study - The Adosate TGG

To answer the first two research questions, we conducted a case study adapted from Blouin et al. [BPD+14]. The Architecture Analysis and Design Language (AADL) is a standard language in the aerospace domain, for which a textual editor (OSATE) and a graphical editor (Adele) exist. Blouin et al. discuss the challenge of implementing a BX to synchronise models edited using the different editors. For maintaining consistency between Adele and OSATE models, the Adosate TGG was established. The TGG consists of 60 rules when specified with the model transformation tool MoTE (cf. Sect. 9.2), in which an extended TGG formalism is used that allows the designer to connect more than one element per model with a single correspondence.

For the implementation with eMoflon::IBeX, we created multiple correspondences for each pair of involved source and target nodes. While the TGG is much larger in eMoflon than in MoTE, eMoflon's rule refinement feature [ASLS14] was used to keep the size of the rules manageable. The feature allowed us to define rules involving abstract node types (abstract rules) that can be refined by concrete types and enriched with additional elements in so-called concrete rules. These changes resulted in a semantically equivalent TGG (i. e., a TGG that generates the same language) with 49 abstract rules and 91 concrete rules, of which only the concrete rules are considered at runtime. An overview comparing the (concrete) rules required for the implementations with MoTE and eMoflon is provided in Table 10.1.

	Number of Rules			Number of Rules	
AADL Construct	MoTE	eMoflon	AADL Construct	MoTE	eMoflon
Package (axiom)	1	2	Component Type Features	10	19
Subcomponents	11	12	Feature Group Types	4	4
Component Types	2	13	Feature Group Type Features	10	9
Connections	20	21	Component Implementation	2	9
			Total	**60**	**91**

Table 10.1: Size of rule groups: MoTE and eMoflon

Compared to existing BX benchmark examples (cf., e. g., Sect. 5.8), the number of rules is relatively large, while the average rule size is comparable. The mean of the number of nodes involved in abstract and concrete rules is 6.51 (2.69 created nodes, 3.81 context nodes), and the mean number of edges is 3.31 (2.46 created edges, 0.84 context edges) per rule.

Basic Debugging Features

In an experiment conducted at Paderborn University with 15 computer science graduate students without substantial prior experience with MDE, we attempted to assess if and how VICToRy helps novice users understand a provided, non-trivial TGG.

RQ1: The first task was to identify relations between rules and model elements (which model elements are created/required by which rules?), as well as relations between the rules themselves (which rules depend on other rules?). The students were provided with the Adosate TGG and VICToRy, and asked to work independently on developing an understanding for the TGG. We provided helpful material (tutorials, handbooks, and relevant papers), supervision (answering any basic questions), and held a feedback meeting after two weeks to check the students' understanding for the TGG and ask if VICToRy was helpful and in what ways it was used.

In general, all students stated that the debugger indeed helped to get an overview of the entire TGG. It was especially helpful to identify which rules are applicable for an empty model (axiom rules) without clicking through all of them, and to determine which rules provide context for other rules (rule dependencies). Most students started exploring the TGG rules by generating consistent model triples and inspecting the resulting transformation protocol. In a second step, they then attempted to transform smaller instances in forward and backward directions. As all models can be saved to disk at any point of time, it was easily possible to try out different alternatives starting from a common state of all models.

RQ2: In a second task, the students were provided with the Adosate TGG, and with a test suite consisting of input models and expected output models for both forward and backward directions. One of the students was then asked to either make a change to a TGG rule, or to the supplied input models, resulting in both cases in a mismatch between TGG and test suite. The other students were then asked to determine and explain this mismatch. This task was carried out in a slightly more controlled manner, restricting the allotted time to a few hours, and asking the students not to perform a diff between the original and changed rules.

The feedback from the students for this task (based again on a feedback meeting and discussion) was much less positive. The task turned out to be (1) much too difficult for novice users, and (2) VICToRy proved not to be of much help as it does not provide any information about why a rule is *not* applicable in a specific situation (even though the expectation is that it should be). While the possibility of selecting protocol entries and inspecting the resulting changes on the models was appreciated, many students stated that the opposite direction, i. e., selecting model elements in the visualisation and highlighting the "responsible" rule applications, would be indeed helpful as well and is currently missing.

Furthermore, there was a clear need for the introduction of breakpoints, which did not exist at the time of the experiment. The transformation should start in the RUN mode and stop at a certain point defined by the user, e. g., when a specific rule is applicable for the first time (complying with the rule name breakpoint as described in Sect. 10.4). This enables users to skip irrelevant parts of the transformation that are already clear to them and set the focus on debugging problematic steps. Based on this observation, the breakpoint concept described in Sect. 10.4 was introduced, which is evaluated via the following to answer RQ3.

Breakpoint Concept

To assess the value that the introduced breakpoint concept adds to the debugger, semi-structured interviews with five TGG experts were conducted in May and June 2021. The goal of the interview process was to gather feedback and suggestions for future improvements, both for specific breakpoint types and for the general handling of the debugger. Each interview started with a short presentation that introduced the problem along with

a running example in the context of the *FamiliesToPersons* benchmark [ABW17]. Subsequently, each implemented breakpoint type was briefly described and demonstrated based on the tool. After each demonstration, the participants was given the opportunity to comment on typical use cases for such a breakpoint type and the usefulness in the respective context. In the third part of the interviews, a faulty version of the *FamiliesToPersons* TGG, i. e., a version with a few distorted rules, was shown to the experts. The VICToRy debugger was then used to detect those faults, such that the experts could comment on the usefulness of the debugger (including the breakpoint concept) in general.

Breakpoint	**Assessment**	**Suggestions**
Model size	Suitable for exploring new TGGs.	The PlantUML visualisation should be exchanged, it might crash even for medium-sized models. The scope could be restricted to a specific element type.
Number of matches	Also a suitable breakpoint for exploration purposes. Could be used to detect faults, if a rule is applied more often than expected.	
Rule name	Intuitive and easy to use. It can be used to detect the misbehaviour of a certain rule, or to skip rather uninteresting rule applications.	For larger TGGs, the selection of multiple rules would be a handy feature, e. g., by using regular expressions.
Element type	Similar to the rule name breakpoint, it is easy to use and understandable.	The user should define whether the breakpoint holds for created or translated elements (or both).
Attribute condition	Can be used when searching for specific matches or rule applications. Another typical use cases is to find faults in attribute conditions of TGG rules. While it is a powerful feature for experts, it can get confusing for novices.	A drop-down menu or an autocompletion feature could support inexperienced users in defining advanced conditions.
Combined	Enables the user to define statements of propositional logic for breakpoints. The visualisation as a tree structure is appropriate.	Plausibility checks should be added to avoid faults in the breakpoint definition itself.

Table 10.2: Feedback for specific breakpoints

RQ3: The experts' feedback concerning single breakpoint types and the debugger in general is summarised in Tab. 10.2 and 10.3. Regarding the breakpoint types, the combination of element type breakpoints and attribute condition breakpoints was perceived as very powerful. All implemented breakpoint types are useful for different purposes. The two types that were left to future work (pattern and node breakpoints) were regarded as useful additions, involving a presumably high implementation effort, though.

Also the overall feedback for the debugger was largely positive. Although the breakpoint concept is powerful and offers several configuration options, the handling of the debugger is perceived as adequate for the intended user group. In accordance with our observations

during the initial student experiment, the experts stated that the debugger is very helpful to understand *why* certain transformation steps take place, but unable to explain why, e. g., a particular rule is *not* applied.

Criterion	Assessment	Suggestions
User Interface	Clean and well-structured, the visualisation of rules and (partial) models, and the history of rule applications are useful.	For the hybrid approach, the visualisation of markers would be a helpful improvement.
Features	Sufficient for practical use. All breakpoints are helpful, especially the combination of the breakpoints for element types and attribute conditions.	Further configuration facilities could be implemented, a pattern breakpoint would be useful to examine graph constraints.
User Interaction	The interaction possibilities with the debugger are well-structured and understandable.	A handy feature would be to undo rule applications (as for the concurrent synchronisation component). This must be supported by the TGG tool as well, though.
Usability	Positive feedback for the overall usability. It is valuable that the main window, the configuration pop-up and the breakpoint menu can be used in parallel.	More support for the attribute condition breakpoint and another visualisation technique (cf. Tab. 10.2).
General	Eases the access of novices (students and practitioners) to TGG tooling. Useful for why-debugging, but not for why-not-debugging.	A quantitative user study and connections to other TGG tools could underpin the debugger's applicability.

Table 10.3: General feedback for the VICToRy debugger

Concurrent Synchronisation Component

To receive early feedback throughout all design phases of the component, we conducted semi-structured interviews with ten MDE experts between October 2020 and February 2021. Experts from different fields (TGGs, BX, and other sub-branches of MDE) were involved to gather requirements and assess the prototype from different perspectives. Similar to the evaluation of the breakpoint concept, each interview started with a brief introduction to the problem based on a running example. As the interviews took place in parallel to the development process and feedback from earlier interviews was continuously incorporated, the structure of the interviews was continuously adapted as well. While the first five interviews were purely based on diagrams, the last five interviews included a live demo of the current state of the prototype (cf. Fig. 10.7).

RQ4: In the following, a short summary of the gathered feedback is provided, grouped by six aspects that played an important role in all interviews.

Goal: The concurrent synchronisation component should be able to list conflicting changes and provide the user with different options for resolving them. Besides resolving

conflicts manually, it should be possible to start an automated resolution process, e. g., for incorporating uncontroversial changes.

Actors: All experts agreed on involving both technical and human actors into the synchronisation process. The "background operation", i. e., the synchronisation operation of the underlying tool, can be regarded as a technical actor. There were different opinions about the involved human actors: While some experts stated that one or two domain experts should use the tool (resolving conflicts collaboratively by negotiations), others regarded an integration expert as the system's key user.

Capabilities (main features): First and foremost, the component should provide visual support for change and conflict detection and resolution on models. A history or list of changes and resolved conflicts with time stamps should give an overview of the current state and recent actions. The tool should also be able to provide an "explanation" of the conflict in some way, and offer consistency checks to validate the results.

Resolution process: Both manual and automated conflict resolutions should be possible, resembling a "mixed-initiative approach" known from the human computer interaction domain. During the process, it should be possible to undo and redo operations, and to have a preview for the effects of possible next steps. The confirmation or acceptance of the final solution should always be done by the user. Interestingly, the expert disagreed on whether the conflict resolution should operate on the level of rule applications (as in [FKM+20]) or user edits (as in Chap. 7). A question that remained open is how to automatically generate (understandable) descriptions of conflicts.

Additional features: Some of the suggested features were considered as handy, but not mandatory for the synchronisation process. The tool could offer a comparison of the current state and the previous state to visualise the effects of the last change. In accordance with the main debugger, an option to save intermediate model states was suggested. Furthermore, configuration options such as label abbreviation and hiding, and displaying the neighbourhood of a sub-graph (cf. Sect. 10.5) are perceived as helpful. Some features that could not be integrated into the prototype are search functionalities for, e. g., particular conflicts or elements, and tutorials or wizards to ease the access of novices to the tool.

Visualisation options: The visualisation of models and highlighting of conflict is perceived as very helpful, but the experts criticised that the reordering of elements after each edit operation in PlantUML is confusing. The resolved conflicts should be marked as such to visualise the changes made during the synchronisation process. A legend that explains the visual syntax in use was considered as necessary. Some persons stated that a visual concrete syntax would be helpful for domain experts, but is hard to implement.

Overall, helpful feedback could be gathered for multiple aspects, including usability, conflict resolution strategies, and visual modelling aspects. In most cases, there was agreement between the experts, encouraging us to integrate the suggested features into the prototype.

Summary

Revisiting our research questions, our initial exploratory experiment at least indicates that (RQ1) VICToRy appears to help obtain an overview of a non-trivial TGG the user is not familiar with, but (RQ2) leaves room for improvement regarding the detection of faults in either models or rules. The expert interviews indicate that the introduced breakpoint concept substantially eases detecting these faults as precise conditions for pausing the transformation process can be defined (RQ3). An extension towards *why-not-debugging* facilities [AC19] still seems to be necessary to properly address bug finding tasks. For

the interactive resolution of conflicting changes in concurrent synchronisation scenarios, requirements and feature requests were gathered in a series of expert interviews that ended up in a UI prototype (RQ4). The actual development of the concurrent synchronisation component is left to future work, though.

Threats to Validity

The most striking issue of the evaluation procedure is its purely qualitative nature. Our experiment with students is at best a pre-study for a more formal, controlled experiment and quantitative evaluation with multiple use cases, a larger group of test persons, and objective measures. For assessing the breakpoint concept and designing the concurrent synchronisation component, only 5 + 10 experts were involved in the interview process. Many of them possess expert knowledge about TGGs and have worked with the eMoflon tool suite before. The interviews followed a uniform structure, but due to the lack of a standardised questionnaire and open discussion phases, the results are not fully repeatable. While we cannot generalise our results, our goal was not to provide hard empirical evidence for the effectiveness of VICToRy but rather to explore the design space in a realistic setting and brainstorm together with both novice users and MDE experts for promising features to guide future extensions of VICToRy.

10.8 Summary and Discussion

We presented the add-on component VICToRy for interactively visualising single steps of the model generation and transformation process. As user involvement is a key feature of fault-tolerant systems, the VICToRy debugger enhances our tool support for consistency management. Besides the inspection of possible rule applications in the current state, the user can retrace the prior transformation process using a transformation protocol. The GT- and TGG-based tool is fully integrated into the eMoflon tool suite but can be used along with other applications via a defined interface.

A concept for switching between different debugging modi via breakpoints is presented. Breakpoints of different types can be configured in many ways and combined to form complex breakpoint conditions. The debugger can be run for all consistency management operations of the eMoflon tool suite. For the operations of the hybrid framework, the user can step through the process of finding all rule application candidates, whereas the determination of the final solution via ILP solving remains fully automated. A component for interactively synchronising conflicting changes after concurrent updates was designed to address the special challenges of concurrent model synchronisation.

Even though expert interviews were conducted to assess the applicability of this component, structured user acceptance tests with respect to the understandability and controllability of the consistency restoration process are left to future work. As an extension towards supporting why-not debugging, information about reasons for blocked rule applications should be presented to the user to support detecting logical faults in TGG rules, or a mismatch with expectations in provided input models and tests.

After having presented a conceptual framework and tool support for fault-tolerant consistency management, the applicability of these concepts and tools in practice shall be underpinned with two industrial case studies in the remainder of this thesis.

11 Automating Model Transformations for Railway Systems Engineering

In the previous chapters of this thesis, a hybrid framework that enables fault-tolerant consistency management was introduced, as well as its implementation as part of the eMoflon tool suite. Several experimental evaluations have been presented in Chap. 5 – 8, focussing on the runtime performance of the conceptual framework in different settings. Furthermore, user studies and expert interviews have been conducted to empirically assess the tool support in Chap. 9 and 10.

In the remainder of this thesis, the applicability of the hybrid framework shall be underpinned with two industrial case studies. This chapter introduces the practical application of the running example of this thesis, i. e., a BX between SysML and Event-B, in the context of railway systems engineering. The operations for forward transformation and (concurrent) model synchronisation are of particular interest, while the other operations seem promising for future use as well. With respect to tool support, we will show how eMoflon::IBeX can be coupled to IDEs for different modelling languages.

The remainder of this chapter is organised as follows: After introducing the industrial context in Sect. 11.1, an overview of related approaches is provided in Sect. 11.2. The automated transformation is motivated with an example in Sect. 11.3. As the conceptual part of the use case has already been introduced throughout this thesis, we continue with a sketch of the implemented tool chain in Sect. 11.4. Section 11.5 presents a qualitative evaluation of our approach based on three representative test cases provided by DB Netz AG, our industrial partner for this case study, and semi-structured interviews with three practitioners. Finally, Sect. 11.6 summarises the results and proposes directions for future research.

11.1 Industrial Context and Motivation

MBSE, already standard practice in domains such as defense and aerospace engineering, is also gaining popularity in the railway domain, where MBSE tools are used to create a standardised system architecture, functions, and interfaces for railway systems [ABC+20]. As failures of such safety-critical systems can lead to serious damage, a formal verification of the expected system behaviour is very important. To limit roll-back and re-implementation costs, the verification and validation of safety requirements should be integrated into the early stages of development. [Fre12]

At DB Netz AG, a railway infrastructure manager that operates major parts of the German railway system, an MBSE process is to be introduced for interface standardisation as part of the EULYNX initiative[1]. The challenge of enabling both high-level system modelling *and* formal system verification is addressed by employing a BX between SysML and Event-B.

To support high-level systems modelling, the PTC Integrity Modeler[2] is used for creating SysML state machines. While simulation-based testing can be used to identify software

[1] https://www.eulynx.eu/
[2] https://www.ptc.com/en/products/windchill/modeler

faults in SysML models, a formal proof of correctness with respect to expected system behaviour is not supported. To overcome this limitation, Event-B [AH07, Hoa13] – a formal method for system-level modelling and analysis – is used to verify safety properties via proof obligations for all possible system configurations. As a simulation and verification environment for Event-B, the RODIN platform [BH07] is used.

The current development process is depicted in Fig. 11.1. Compared to the process depicted in Fig. 1.1, the initial transformation from SysML to Event-B in step (ii) is split into two parts: In step (ii.1), the SysML models are translated *manually* by experts in formal methods to semantically equivalent UML-B [SB06] state machines (U). In step (ii.2), these UML-B models are transformed automatically into Event-B code (E) using the UML-B plug-in of RODIN.

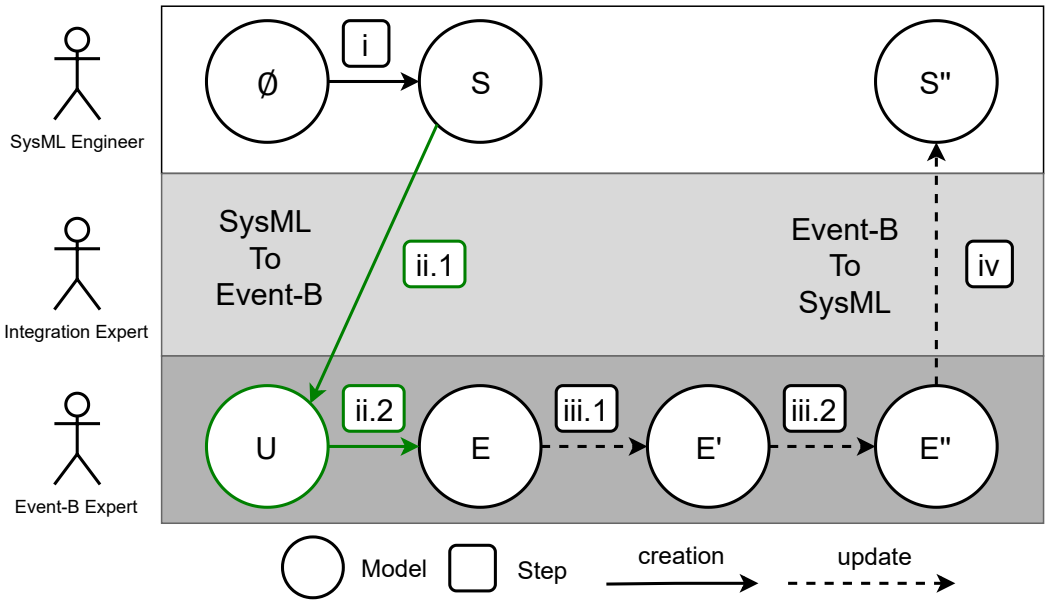


Figure 11.1: Process overview

The current process is tedious and error-prone due to the manual transformation from SysML to UML-B state machines, as well as the equally manual backward propagation of corrections in the Event-B code to the original SysML models. This clearly indicates a need for an increase in the level of automation. The intermediate UML-B representation is not required for the verification itself, but rather a necessary concession to keep the manual transformation manageable, as writing Event-B code directly is challenging. It is moreover impossible to certify these manual steps of the process as there is no transformation specification that could be reviewed by experts.

We argue that the SysML to Event-B forward transformation and coupled backward synchronisation are best viewed as a BX, as the SysML models cannot be simply reconstructed from their Event-B counterparts. Instead, applied changes in the Event-B model should be propagated back to the SysML model based on the same consistency relation, being a key property of BX languages. Consequently, the practical use case can be solved using the hybrid framework presented throughout this thesis, using the FWD_OPT operation for the forward transformation from SysML to Event-B, and an appropriate synchronisation operation (CS, SYNC or INTEGRATE) for propagating changes on the Event-B model back again. We use the set of TGG rules that was introduced in Chap. 3 and 4, which originates from the knowledge of practitioners about the concrete application scenario. Using the hybrid framework to accomplish the automated BX, the generation of

an intermediate UML-B model can be completely omitted, merging (ii.1) and (ii.2) into a single transformation step. Traceability information between informal requirements and the modelled system, specifically for safety properties, can further be maintained. For implementing the transformation, we establish a tool chain connecting the modelling tools used at DB Netz AG for SysML and Event-B.

We demonstrate the feasibility of our approach by solving three representative test cases provided by DB Netz AG. Based on these case studies, we provide a qualitative evaluation by conducting semi-structured expert interviews with employees of the company to assess the applicability of the proposed solution in practice. Our contribution is to investigate and evaluate a practical application scenario of a model-to-model transformation from SysML and Event-B in the railway systems engineering domain. The identified potential and limitations of our approach can be used to drive further research towards improving the practical applicability of model transformation technology in general, and the fault-tolerant hybrid approach in particular.

11.2 Related Work

Comparable to our contribution and focus in this chapter, there have been several projects investigating the application of TGGs in an industrial context. Giese et al. present an approach for transforming SysML models to AUTOSAR[3] using TGGs [GHN10]. In contrast to our application, however, SysML block diagrams are transformed and not state machines. The application domain is also different, i.e., supporting the transition from system design to software design in the automotive domain as opposed to supporting a formal analysis for safety requirements in the railway domain. These differences also lead to a different set of relevant challenges: Giese et al. focus on transforming and synchronising large models in a scalable manner, while we focus more on comprehensibility and expressiveness, especially regarding attribute manipulation.

Hermann et al. present an approach to translate satellite procedures from one language to another also using TGGs [HGN+14]. While Hermann et al. face similar challenges as we do, in this case comprehensibility and formal correctness of the transformation, the application domain is of course different (aerospace vs railway). Moreover, the supported development process is simpler than ours as Hermann et al. only require a forward transformation while we require both a forward transformation and a backward synchronisation. In general, existing industrial case studies with TGGs tend not to focus on a qualitative evaluation, investigating instead quantitative aspects such as scalability of the solution, which might arguably not be the strongest argument for BX in general and TGGs in particular. Aspects like fault-tolerance during the development process or the use of multiple consistency management operations are not dealt with in existing work.

Concerning our choice of TGGs as a BX language to solve industrial use cases, Anjorin et al. [ABW+20] provide a recent overview and benchmark of various BX languages, and state that TGGs scale well in practice for transformation and synchronisation tasks. To solve our use case, we could have used any equally mature BX language such as BiGUL [KZH16]. While we cannot (yet) back the following conjecture with any empirical evidence, we suspect that TGGs might be more comprehensible than, e. g., BiGUL in the context of our application scenario as relevant domain experts are familiar with visual modelling languages and the concept of transformation rules, as opposed to a functional approach and Haskell-like syntax. The situation is probably completely reversed for other application domains.

[3] https://www.autosar.org/

The general problem of SysML not being sufficiently formal has been addressed in different ways by numerous authors. Pais et al. present an approach to transform SysML state machines to Petri Nets (PNs) [PBG14] to be able verify formal properties, generate code, visualise and execute the resulting PNs. As the authors use ATL for the transformation it remains unclear how insights gained from the formal analysis are reflected back to the SysML state machines. Huang et al. present a transformation of SysML activity diagrams to PNs also for formal verification [HMM20]. According to the authors, the execution semantics for UML and SysML are not sufficiently precise to be unambiguous and thus they use the fUML standard to provide a precise semantic definition of SysML activity diagrams. Again it is unclear how changes to the resulting PNs are reflected back to the corresponding activity diagrams.

There are several further examples for transformations from SysML to formal languages such as Alloy [ABGR10], NuSMV [CLFLW16], Promela [CLS20], or to CSP# processes [AYK+14]. While it is possible to verify the transformation results with state-of-the-art model checker in each case, the subsequent incorporation of findings remains a manual task. In general, we argue that the work-flow of applying a "formalising" transformation, gaining insights from a formal analysis, and reflecting these insights back to the initial models in a productive manner is a clear application of BX languages.

11.3 Motivating Example

To motivate the need for formal verification and validation in the railway domain, a small and simple case study provided by DB Netz AG is introduced. The case study originates from a technical specification and requirements document describing a *point machine* interface to an interlocking. In railway signalling, an interlocking is the part of a signal apparatus that prevents conflicting movements through an arrangement of tracks such as junctions or crossings.[4] An interlocking is designed so that it is impossible to display a signal to proceed unless the route to be used can be proven to be safe.

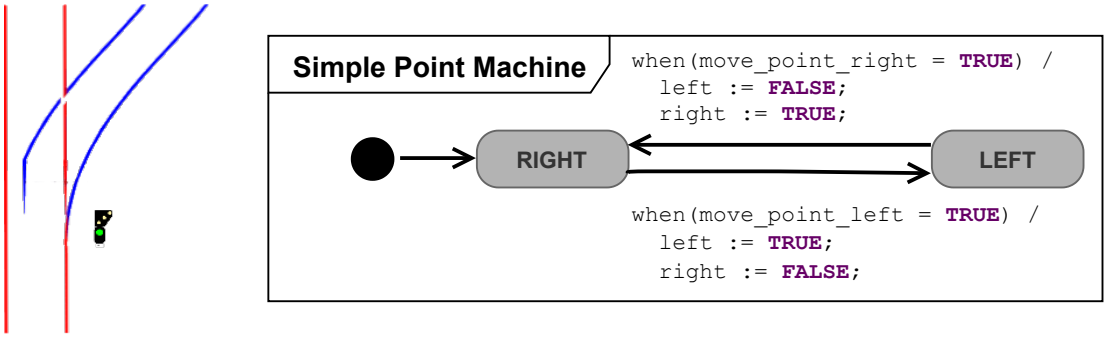


Figure 11.2: Point machine case study (left) and SysML state machine (right)

Figure 11.2 depicts the configuration of a point machine: Two tracks represent two possible positions, denoted as left and right. The lamps represent the position of the tracks after the movement. The main requirement is to move the tracks to the left or right position depending on the commands from the interlocking. Being a safety-critical system, properties such as: "When the track is set to right, the lamp should be lit" have to be proven to hold as soon as the command is given by the interlocking.

[4] https://projects.au.dk/into-cps/industry/railways-case-study/

The EULYNX initiative[5] develops the railway interlocking specification and requirements, involving various state machine diagrams for individual systems of the interlocking. These include level-crossing systems, interlocking systems, light signalling systems, and other auxiliary systems. Each of these systems interacts and communicates with each other through a communication interface, which must guarantee safe communication. To this end, the relevant behavioural part, i. e., the state machines of the SysML models must be verified against safety properties necessary to fulfil user requirements. To perform the verification, the state machines are transformed to Event-B code, so that Event-B can be used as a formal method.

The implementation of the transformation was prepared in two steps: First, based on the list of state-machine diagram elements from the SysML 1.6 specification[6], we defined a supported subset of features sufficient to fulfil the requirements of the involved stakeholders at DB Netz AG, i. e., domain experts both in systems engineering and formal verification. Second, we identified semantic interrelations between the two languages, which form the basis for defining the consistency relation between the two language subsets. As a result of these two steps, the triple metamodel of Fig. 3.8 and the TGG rules which have been introduced in Chap. 3 and 4 were created. In the following section, we show how eMoflon::IBeX can be used to seamlessly integrate existing SysML and Event-B tools to implement the transformation.

11.4 Implementation

This section introduces the tool chain used to implement the transformation, and sketches the work-flow established for the existing tools at DB Netz AG.

PTC Integrity Modeler

For creating SysML models, the PTC Integrity Modeler[7][8] is used at DB Netz AG. The PTC Integrity Modeler provides a development environment that allows different systems engineering teams to work in a collaborative setting, from the conceptual level to the delivery and maintenance of the system. It helps define an unambiguous single model definition of the system, including requirements, functions, as well as hardware and software components. Besides SysML, several other standards of the Object Management Group (OMG) are supported. For creating SysML state machines, a visual editor is used. It is also possible to simulate modelled behaviour to detect errors in early stages of development. The tool provides interfaces for synchronisation with other modelling tools (e.g. Simulink[9], Doors[10]). Code in different GPLs (e. g., C, C++, Java and Ada) can be generated from the models [ZRH+20]. For our tool integration solution, the model export to EMF-compatible XMI is relevant.

RODIN Platform

As an IDE for formal modelling with Event-B, the *RODIN* tool [ABH+10] is used at DB Netz AG. It is provided as an open-source Eclipse plug-in that supports the construction

[5] https://www.eulynx.eu/
[6] https://www.omg.org/spec/SysML/1.6/PDF
[7] https://www.mathworks.com/products/connections/product_detail/ptc-integrity-modeler.html
[8] https://www.ptc.com/en/products/windchill/modeler
[9] https://www.mathworks.com/products/simulink.html
[10] http://www-03.ibm.com/software/products/en/ratidoor

and verification of Event-B models. Besides basic support for formal modelling, RODIN provides feedback for the developer at design-time. Event-B development (modelling and programming), and formal verification are decoupled into distinct phases to ease, for example, tracing the origin of a failed proof obligation. As verification techniques, both model checking and theorem proving are supported. Model checking can be used as a pre-filter, before theorem provers are applied to proof obligations [ABH+10]. In addition to their textual representation, formal models can be visualised and simulated to make the models more comprehensible for the developer. These features are integrated via a range of plug-ins for the RODIN platform. Various analysis tools, such as theorem provers[11], model checkers[12], step-wise simulation[13] and translation tools such as for UML-B[14] have been developed as extensions for the RODIN platform. The UML-B plug-in, for instance, helps to diagrammatically visualise the formal model, and thereby aids construction and validation. Event-B machines are stored and imported as XMB files, which are syntactically similar to XMI files. XMB files can be visualised with the Rose Structured Editor from different viewpoints.

eMoflon::IBeX

To bridge the modelling environments for SysML and Event-B (cf. Fig. 11.3), an additional tool is required to perform the transformation and synchronisation steps in both directions. In our approach, eMoflon::IBeX (cf. Sect. 9.4) is used to address this task. The decision between the components IBeX and Neo was made in favour of the former due to its file persistence and exchange format: Metamodels for source, target, and correspondence models as well as the respective models are all EMF compatible and can be persisted in any EMF compatible format including the default XMI. The common use of EMF as a modelling standard substantially eases the establishment of a tool chain to connect the PTC Integrity Modeler, eMoflon::IBeX, and RODIN. Although only forward transformation and backward synchronisation are of primary interest for our application, the practitioners also appreciate the support of other consistency management operations, such as consistency checking (cf. Sect. 11.5).

From PTC Integrity Modeler to RODIN and Back

The setup for the established tool integration is depicted in Fig. 11.3. eMoflon::IBeX is used as a bridge between the PTC Integrity Modeler and RODIN. After specifying the consistency relation between SysML state machines and Event-B as a triple metamodel and a TGG, eMoflon::IBeX is used to operationalise the TGG as required for the scenario.

The behaviour modelling of the system is done using SysML with the PTC Integrity Modeler. The resulting model is exported as an XMI file from which a relevant part (state machines) is extracted and passed on as the input model for eMoflon::IBeX. The forward transformation is executed by eMoflon::IBeX, using the derived forward rules and the respective operational strategy (cf. Fig. 9.4). Besides the correspondence and target model, IBeX also generates a transformation protocol containing information about which rules were executed in which order to create which elements. This protocol was already used to visualise the history of rule applications in the VICToRy debugger (cf. Fig. 10.4). While only the target model is strictly required for the tool integration, the correspondence model and transformation protocol are useful for debugging and understanding the

[11] http://www.b4free.com/index.html
[12] http://www.stups.uni-duesseldorf.de/ProB/overview.php
[13] http://www.brama.fr/indexen.html
[14] users.ecs.soton.ac.uk/cfs/umlb.html

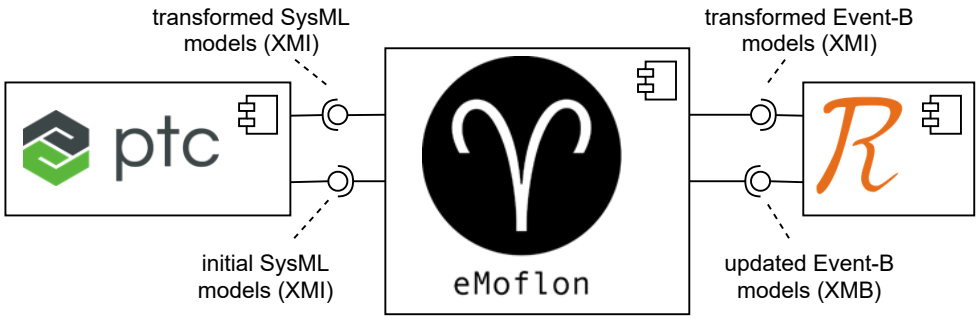


Figure 11.3: Overview of the tool integration setup

transformation. The next step is to convert[15] the generated target model from XMI to XMB and to import it into RODIN.

RODIN is used to automatically generate Event-B code from the XMB file. The Event-B code can now be used to perform formal verification and check if all safety requirements are fulfilled. Gained insights are integrated directly in the Event-B code, resulting in a new version. To reflect these changes to the Event-B code back to SysML, the backward synchronisation with eMoflon::IBeX requires (i) the old triple of source, correspondence, and target models, and (ii) a *target delta* representing the changes applied to the target model, which should be backward propagated incrementally to the existing source model. Currently, this target delta (also a model) must be created manually based on a text diff between the initial and final Event-B code. This is certainly a step that could be improved in the future by automatically transforming a diff on Event-B code to a target delta that eMoflon::IBeX directly understands.

The current state of the implementation should be regarded as a proof-of-concept prototype as (i) the level of automation can still be improved, and (ii) only the most important parts of the two behavioural models are covered. We are convinced, however, that the scope can be extended to cover the remaining parts of the state machine specification and even further SysML models without fundamentally changing the overall work-flow. Using XMI as a uniform data exchange format seems promising as it is supported by all three tools, but has a number of drawbacks including its missing ability to represent diffs or delta structures [ZRH+20]. Further tool integration could replace XMI by a more suitable standard for data exchange.

This would at the same time also ease an additional implementation with eMoflon::Neo, which offers CS a fault-tolerant operation for backward synchronisation, whereas this task is solved with greedy operations (SYNC or INTEGRATE) in IBeX. The data exchange format was the main argument for using IBeX for this practical use case, while we are convinced that an implementation with Neo as a connector for the two modelling tools would be equally possible. To assess strengths, weaknesses, and the potential of our approach, the results of a qualitative evaluation are presented in Sect. 11.5.

11.5 Evaluation

We now provide a qualitative evaluation of our implemented solution based on three small but representative test cases provided by DB Netz AG. After running the transformation and synchronisation chain for the three test cases, we then conducted a semi-structured

[15] In most cases this just involves changing the extension of the file from ".xmi" to ".xmb".

interview with three SysML and Event-B modelling experts at DB Netz AG. In particular, we aimed to investigate the following research questions:

RQ1 Feasibility: Is it possible to transform representative examples of SysML state machines into Event-B?

RQ2 How is the applicability, extensibility and usability of the solution perceived by relevant practitioners?

An overview of the qualitative evaluation was already presented in prior work [WSA+21]. The interested reader is referred to Salunkhe [Sal20] for a more detailed treatment of the three test cases and the subsequent expert interviews.

Representative Test Cases

To validate the transformation results, formal modelling experts at DB Netz compared them to the expected Event-B models and conducted simulations using the RODIN platform. Although the test cases are rather simple and contain only language elements that were presented in Sect. 3.1, they differ in the structure of transitions and states, and combine different triggers and actions with each other. In particular, they can be used to investigate whether the rule-based transformation produces a correct result that complies with the modelling experts' expectations.

Log-In Form State Machine The first test case consists of a state machine designed for a simple *log-in* process. It consist of three states: In the `IDLE` state, the log-in form is ready to accept requests, while the `ACTIVE` state indicates that a user has logged into the system. The `SERVICE_ERROR` state represents an error state for the system. It is possible to switch between `IDLE` and `ACTIVE`, as well as between `IDLE` and `SERVICE_ERROR`, whereas a direct transition from `ACTIVE` to `SERVICE_ERROR` is not possible. The transition from `IDLE` to `ACTIVE` has a trigger (in round brackets) and a guard (in square brackets), whereas all other transitions have only one trigger. Figure 11.4 depicts the state machine for the log-in form.

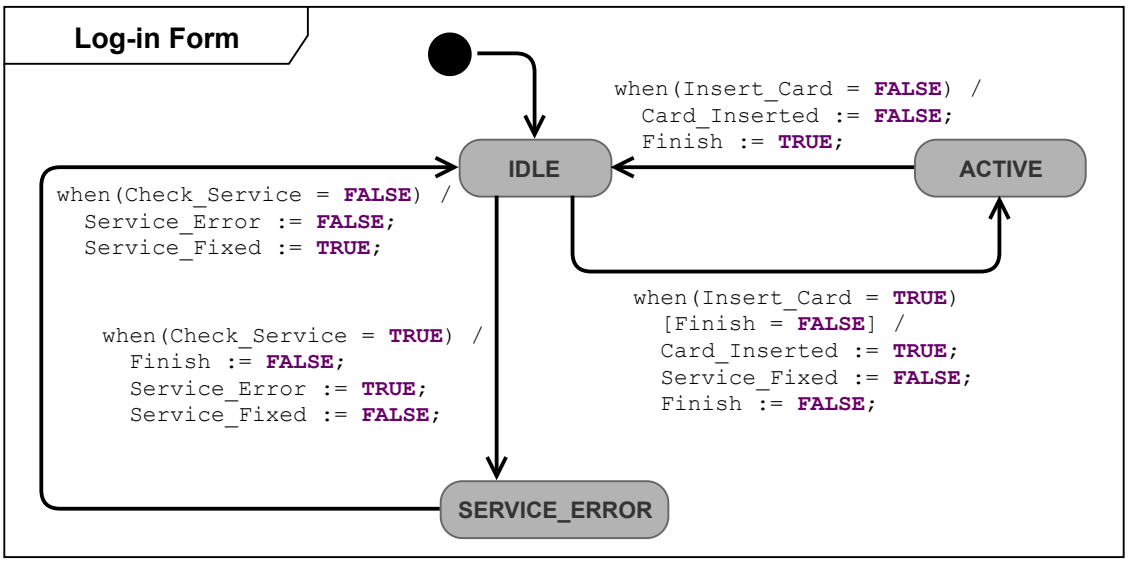


Figure 11.4: Test case 1: Log-in form

Light System This state machine represents a basic *light system*. Similar to the log-in form test case, there are three states, of which one is an error state. In this example, however, there is a transition from the state `ON` to the state `SYSTEM_ERROR`, forming a cycle between all three states. Figure 11.5 depicts the state machine for light system.

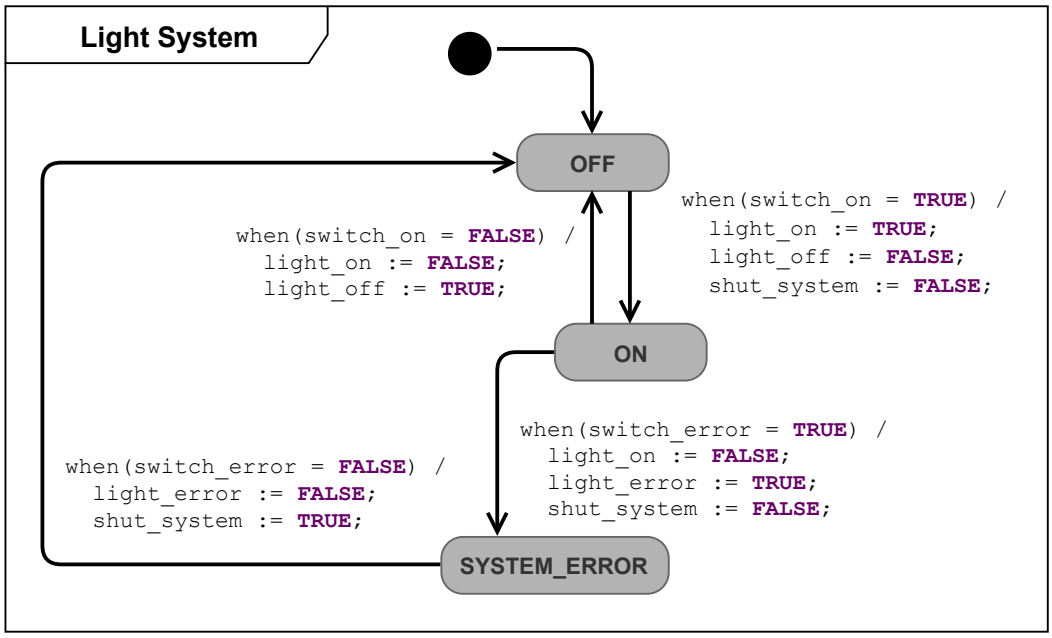


Figure 11.5: Test case 2: Light system

Trip Planner System This test case is slightly more complex than the previous two as there is a fourth state and a fifth transition to be transformed. The system models a *trip planner*, e. g., for a cab ride. As soon as a user requests a trip, they are asked to pay a certain amount of money. The user can either confirm the payment and choose a driver, or go back to modify the requested trip. As soon as a driver is assigned, the trip can be conducted. In a last step, the driver is unassigned and the state machine goes back to the state `TRIP_REQUESTED`. Figure 11.6 shows the state machine for the trip planning system.

Summary Using our implemented solution, it was possible to generate Event-B machines from the SysML input models. The generated Event-B machines were all syntactically correct and could be displayed in the RODIN platform. According to domain experts, the transformation works as expected for the three test cases.

Interviews with Modelling Experts and a Project Manager at DB Netz AG

To investigate RQ2, we conducted three semi-structured interviews presented in the following, which were based on the implemented test cases. We were able to obtain interviews with (i) a modelling expert from the SysML semi-formal modelling side, and (ii) a modelling expert from the Event-B formal modelling side in order to get technical feedback, as well as with (iii) a project manager in order to get feedback on the complete approach and aspects related to strategic plans for future developments. The interviews presented in this section are a summary of the complete interviews provided online[16].

[16] https://drive.google.com/file/d/1CPJ01Usls_kafLeh0KdmCOrdwgG0ukKM

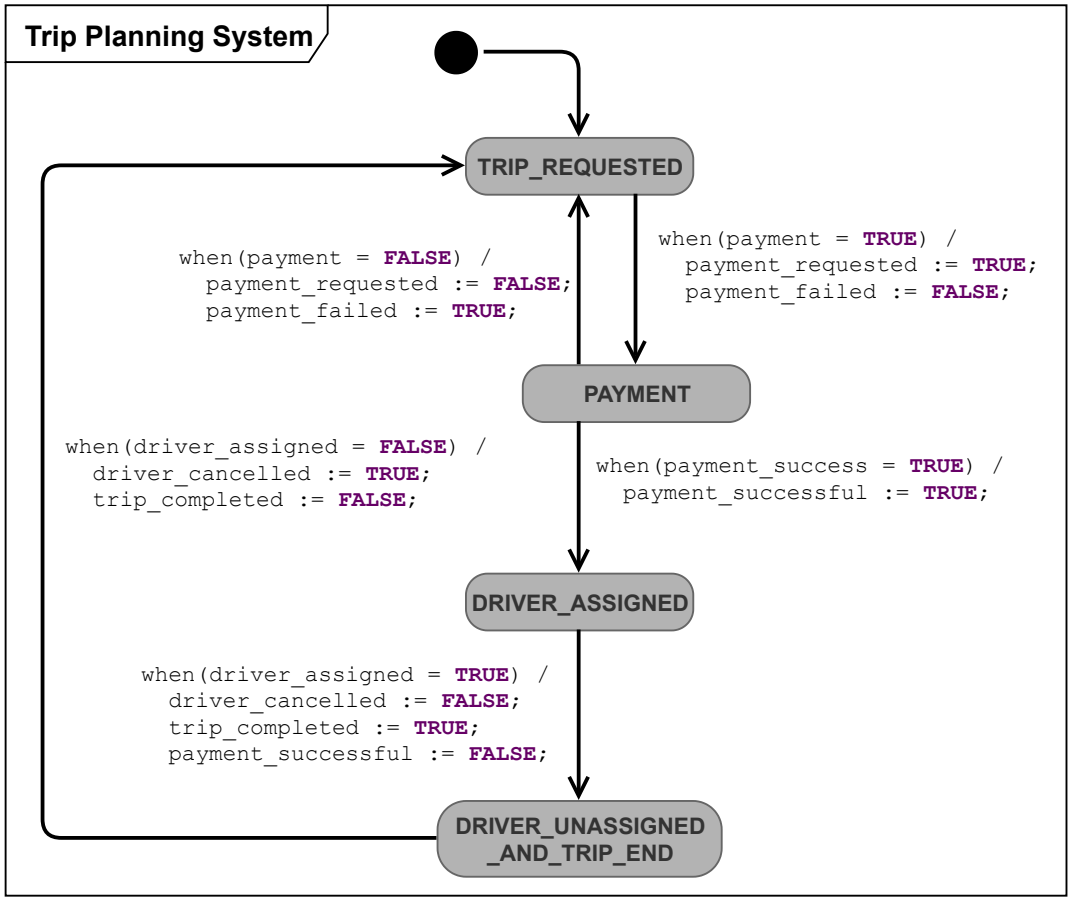


Figure 11.6: Test case 3: Trip planning system

Applicability The modelling experts expect the rule-based transformation to save time and lower the error rate compared to the current manual process. Even for project members with only basic knowledge about formal modelling, it should now be possible to generate and verify the formal model produced from a SysML state machine thanks to the fully automated procedure. Furthermore, based on expert reviews of the TGG, a certification of the entire process is now a possibility.

In order to use the approach in practice, the considered subsets of the SysML and Event-B metamodels must be large enough. For a first proof-of-concept version, the currently supported subsets are sufficient but must be extended in the future. This extension depends on requirements, however, as it is important to only cover features that are actually useful for the formal verification process.

An advantage of using a BX language for the transformation is that several other consistency management operations can be derived from the same specification, i.e., from the developed TGG. While forward transformation and backward synchronisation are currently most relevant to reflect fixes in the formal model back to the semi-formal model, also forward synchronisation could be relevant for reflecting changes in the SysML model at later stages of the engineering process. Other supported operations, such as consistency checking, are certainly perceived as being potentially useful for future workflows in the context of the EULYNX project.

Considering the goals of the EULYNX project, the automated approach supports the synergetic combination of formal and semi-formal methods. A primary goal in EULYNX

was to establish a well-understandable semi-formal language such as SysML to be used by all project partners, as well as to offer separate, complementary formal verification facilities. With the automated approach, this goal can be achieved to enable a uniform validation and verification process that incorporates safety requirements.

Extensibility As the EULYNX project is constantly evolving, it is important to have a solution that can easily be adapted and extended. For example, a recent change was the replacement of flow ports by proxy ports, so the supported SysML language subset would have to be extended accordingly. As the EULYNX specification also involves interconnected state machines, the current subsets also have to be extended to handle such state machines as well.

According to the interviewees, the automated approach can be easily extended and adapted to the expectations of the formal modelling experts for verification. The current transformation results provide a solid basis for the further development of the tool chain by providing reliable results. The only thing lacking at the moment is "conformity", which means that the translated models are not yet certified. If the complete TGG-based tool chain could be certified, then the overall validation and verification approach could be provided to other project members using different target formal modelling languages.

Regarding the future role of the automated transformation in research projects, a medium-term goal is to convince the project managers of EULYNX and RCA[17] of the importance of an automated transformation approach. A short-term goal is to apply the approach to EULYNX models, which can form the concrete basis to convince other project members.

Usability As the complete transformation process still requires some user interaction, usability aspects of the tool chain cannot be ignored. The most important manual steps are including safety requirements and safety invariants. An additional manual step is to add further invariants of two types: State invariants have to hold for a single state, whereas global invariants apply to the complete model. While state invariants can already be generated automatically, global invariants must be specified and added manually by the formal modelling experts. This is because global invariants do not have a corresponding SysML construct and can only be added to the semi-formal model as an informal annotation.

Despite these manual tasks, the use of an automated transformation still leads to a substantial reduction of effort. For complex models such as several state machines with refinements communicating with each other, the automation promises to be especially beneficial in this regard. Moreover, assuring the correctness of the resulting Event-B machine for the manual process is still an unresolved issue, as errors can occur when humans perform the transformation manually. With the automated process, only safety invariants have to be added, and the rest of the Event-B code can be generated automatically, increasing time efficiency possibly by around 70-80%.

The main obstacle with the automated approach is that an average engineer might have reservations about processes they do not understand in full detail. To increase acceptance, it would be very helpful to have a UI that supports the engineer while conducting the automated transformation. The UI should support the user to initiate the transformation process via a single click of a button, and it should present the translated model as well

[17]The RCA initiative is driven by several EULYNX project members and strives for improving command, control and signalling systems using MBSE techniques: https://eulynx.eu/index.php/news/61-rca-gamma-published

as the corresponding verification results. It should also provide a means of visualising and editing the formal model if necessary, and should ideally be easy to use for inexperienced users without any expert knowledge on formal modelling. The UI should provide the domain experts in both semi-formal modelling and formal modelling with a simple workflow to run the simulation and verify user requirements without having to fully understand the underlying details.

Another aspect relevant for future maintenance of the system is the required knowledge for refining the consistency relation definition, i.e., for adding or modifying TGG rules. Different levels of expertise are conceivable: To maintain the set of rules and modify the tool chain, knowledge about both the semi-formal and formal language are required, and probably also expert knowledge in MDE. All other modelling experts should understand the overall process, but should not require in-depth knowledge to apply it. To achieve this, the solution should be well-documented based on the triple metamodel as a central and formal artefact.

Summary and Threats to Validity

For all three test cases presented in Sect. 11.5, the transformation yields correct results according to the modelling experts. As the SysML state machines involve all syntactic constructs presented in Sect. 3.1 in different arrangements, the rule-based approach appears to work as expected for this language subset (RQ1). From the interviews, we conclude that the implementation forms a solid basis for automating the transformation from SysML to Event-B, which can be extended in the future. An extension to support the full expressive power of the formal and semi-formal modelling languages is, however, necessary for practical usage. Furthermore, the handling of the tool chain still requires advanced knowledge about all three tools and underlying concepts. Usage should, therefore, be simplified by offering a suitable UI to support inexperienced users (RQ2).

While the provided test cases incorporate more states, transitions, and actions than the motivating example (cf. Fig. 11.2), the models are still rather small and simple. Larger, interconnected state machines would be necessary to determine corner cases for which further rules might be necessary. Regarding the assessment of the approach, it is important to note that the interviewees were already strongly in favour of automating the transformation, such that chances might be overstated and risks underestimated. Finally, only three people were interviewed, all employed at the same company and working on the same project at the time of implementation. It is questionable, therefore, if our results can be directly transferred to other industries or application contexts. Regarding maintainability, it is important to note that at least one person with advanced knowledge about TGGs must be involved in the project to add and adapt rules whenever this is necessary. For certifying the process, i. e., validating the correctness of the transformation itself, each new rule must be considered. As the TGG-based consistency relation definition is purely syntactic, the certification process should also involve simulation and different testing strategies to complement the formal verification.

11.6 Summary and Discussion

This chapter presents an industrial application scenario for the running example of this thesis, i. e., a BX between the semi-formal language SysML and the formal language Event-B. With help of consistency management operations of the hybrid framework, a previously manual transformation process is automated to take a step forward in certifying the verification and validation process.

The approach is demonstrated with a prototypical implementation that is able to connect the modelling tools PTC Integrity Modeler for SysML, and the RODIN platform for Event-B via eMoflon::IBeX. To validate our approach, we provide a qualitative evaluation based on three representative test cases provided by DB Netz AG and semi-structured interviews conducted with domain experts. Our interviewees stated that the prototype is indeed a good basis for substantially reducing manual efforts and for eventually certifying the transformation process. It became also apparent that other operations (e. g., consistency checks) are indeed helpful and can be used in the future without additional implementation effort.

The scope of the transformation is limited to minimal subsets of both languages covering most of the relevant language constructs, such that it was possible to use the triple metamodel depicted in Fig. 3.8 as a basis for typing the consistency relation. Accordingly, the set of TGG rules presented in Chap. 3 and 4 (with minor adaptions for technical reasons) suffices to properly specify the consistency relation. While the implementation should be considered rather as a proof-of-concept than as a fully functional solution, the practical applicability of the hybrid approach is demonstrated via this case study.

Future steps include extending the transformation of SysML state machines to cover more language constructs, thereby increasing the practical applicability of the approach. This extension involves further tests with real-world models from the EULYNX project. Depending on the results, other project partners can be convinced to use BX techniques in a similar fashion to establish model transformations between their respective modelling languages.

Regarding the tool implementation, a workaround had to be found for one rule (cf. Fig. 4.13) as the hybrid approach does not support multi-amalgamation yet. This motivates the integration of further language features into the framework, in order to make the tool suite more powerful and even more suitable for real-world use cases.

Although fault-tolerance was not an explicit requirement for this project, we are convinced that the respective mechanisms will ease and accelerate the entire process noticeably. While in the manual process, faults in both SysML modelling and the transformation itself made it necessary to restart the process, the automated solution based on the hybrid approach provides the domain and integration experts with early feedback on design errors and enables the Event-B experts to focus on their task of verifying safety properties. The following Chap. 12 will introduce a further case study, in which fault-tolerant mechanisms are crucial for obtaining a feasible solution at all.

12 Automating Test Schedule Generation with Domain-Specific Languages

In the course of this thesis, a hybrid approach to fault-tolerant consistency management was presented and backed up with appropriate tool support. The applicability of the developed concepts in practice was demonstrated in the previous chapter, that has shown how a BX between SysML and Event-B can be used for verification and validation in the railway domain.

In this chapter, we leave the context of the running example of this thesis and use the hybrid approach to create an optimal test schedule that allocates human resources, i. e., software testers, to testing tasks of a product release cycle. We show that a certain degree of flexibility is required to solve this task, as "perfect matches" between the available human resources and the testing tasks to be executed are usually impossible to achieve.

This chapter is structured as follows: After a brief introduction to the industrial context in Sect. 12.1, different strategies for solving scheduling problems in practice are discussed in Sect. 12.2, before comparing our approach to related work in Sect. 12.3. Sections 12.4 – 12.8 present our main contribution by covering the process of establishing our model-driven and fault-tolerant solution. A quantitative and qualitative evaluation is provided in Sect. 12.9, before Sect. 12.10 summarises the results and concludes with an overview of future work.

12.1 Industrial Context and Motivation

In this chapter, we investigate an industrial application of the hybrid approach, or more concretely, the *correspondence creation (CC)* operation presented in Sect. 5.3. While the operation is primary intended for being used as a consistency checker, we show that the created correspondence model can be regarded as an *allocation model* in the context of test scheduling. To begin with, the industrial context and the need for an automated test scheduling solution shall be explained.

At dSPACE GmbH,[1] a well-known developer of software and hardware for testing mechatronic control units, automated testing is often complemented by a series of recurring manual tests executed in every development and release cycle. As such manual testing requires human resources (developers and testers), a test schedule allocating human resources to tests must be created and maintained throughout the testing process. Creating and maintaining a test schedule must take numerous, potentially conflicting constraints into account. This includes the availability of the human resources (holidays, sick leave, requirements of other more important projects), their suitability for particular tests (experience with executing this type of test, developers of the component under test), and the relative importance of the tests. Test scheduling is an optimisation problem as it is typically impossible to utilise all human resources and cover all manual tests. Instead, the goal is to cover as many test cases as possible taking metrics such as priorities and risk into account, while satisfying the given constraints.

[1] www.dspace.com

Prior to this work, test scheduling at dSPACE was performed manually by an experienced test manager. Creating an initial schedule took more than one working day, while maintaining it to take new changes into account required about an hour of work per week. This was, therefore, not only costly but also tedious, error-prone, and difficult to share among multiple people. It also posed a risk to the company: the manual scheduling task cannot be easily delegated to a new or different test manager as it requires experience from creating and maintaining previous schedules. In close collaboration with dSPACE, our primary contribution is to investigate how a fully automated test scheduling strategy can be implemented such that test managers can understand, validate and configure it without prior knowledge of the solution domain.

The problem definition clearly motivates the use of a flexible, search-based strategy, such as the hybrid approach presented throughout this thesis. With the CC operation, the available human resources (source model) and the required testing tasks (target model) can be provided as input to the operation, and the computed correspondence model represents an optimal allocation of testers and testing tasks. To further address the requirement of encoding information such as the suitability of testers for a specific task, or its relative importance, the objective function is enhanced with further parameters in a similar way as for the concurrent synchronisation operation (cf. Sect. 7.6). We establish an additional, very simple DSL for *rating* these allocation candidates, which is simple enough for the domain expert to use with confidence. The overall solution is highly configurable: Domain concepts are captured with metamodels that can be adjusted, TGG rules can be added and modified, and the rating function can be configured as required. Clearly, it would also be possible to build a small configuration tool where parameters can be entered and edited via UI elements. However, we want to provide a working environment for the test manager that uniformly integrates all tasks into the eMoflon tool suite, and thus decided to develop a simple DSL for configuring parameters as well.

While the runtime evaluation results for the CC operation are satisfactory for the used example TGGs (cf. Sect. 5.8), achieving acceptable scalability turned out to be challenging in this scenario due to the number and complexity of TGG rules. Therefore, part of our contribution is also to discuss techniques of achieving scalability in this context with respect to the size of the search space involved. We evaluate our solution quantitatively by comparing automatically generated and adapted test schedules with manually maintained test schedules for the same testing period, and qualitatively by using it productively (completely replacing manual test scheduling) at dSPACE, and interviewing the test manager at the end of the testing period. By applying our approach at dSPACE, recurring manual efforts can be reduced, and the generated test schedules are optimised with respect to the configured parameters. The results are also transferable to other industrial application scenarios, as test scheduling is an essential part of the software development process, further motivating the use of fault-tolerant MDE concepts in practice.

12.2 Approaches to Test Scheduling

Scheduling problems are relevant for a wide range of application domains including scheduling home care services in the health care domain [Gue16], allocating software components to electronic control units in the automotive domain [PH19], and mapping a virtual network to physical resources [TLWS18] in the network virtualisation domain.

Standard solution approaches (see Guericke [Gue16] for an overview) focus on *solving* different variants of scheduling problems on the assumption that a problem definition (typically a set of constraints and an objective function) can be fixed. Indeed, the main

contribution of work in this area is to model and formulate the problem in an adequate manner. While such a formulation is then evaluated and shown to be suitable for a specific family of scheduling problems in a certain domain, two challenges are usually completely out-of-scope and remain unaddressed: (i) that domain experts *understand* the problem definition and can validate it, and (ii) that domain experts can substantially *configure* and adapt the problem definition with reasonable effort and without deep knowledge of constraint solving techniques. Keeping these requirements in mind, three standard approaches to solving test scheduling problems are briefly discussed in the following.

Directly programming a solution in a GPL is problematic for several reasons: First, a lot of work is involved, because from importing the input data to solving the optimisation problem, the entire solution must be implemented manually (at best supported by reusable libraries). Second, as soon as requirements change, the code base must be adapted, making it necessary to recurringly involve the tool developer into the test planning process. As this contradicts our project goal of allowing a flexible adaptation of the problem definition, we decided to exclude such directly programmed solutions.

Another standard approach would be to formulate the problem as a set of constraints and an objective function, so that established constraint solvers can be applied to determine an optimal solution. Following this approach, one would develop a UI that enables the test manager to hand over the problem definition to an ILP solver, such as Gurobi. Considering our requirements, however, this solution is not satisfactory as our test manager would not be able to fully understand the problem definition, let alone validate, adapt and configure it confidently without help.

A (purely) model-driven approach would be to establish a DSL for our domain expert, ensuring that the problem definition now expressed in this DSL can be easily understood, validated, and adapted. The problem definition can then be generated from programs in the DSL so that established constraint solvers can be leveraged, making additional specifications such as TGG rules superfluous. The challenge with this solution is that it requires a considerable effort: The scope and semantics of the DSL have to be chosen carefully as the domain expert will most probably not be able to make substantial changes to the DSL itself once it has been established. Such a DSL would make more sense after enough experience has been collected and the required solution space for the DSL is clearer. As a result, we decided to address the requirements of the application scenario by synergetically combining a simple DSL, TGGs, and optimisation techniques.

12.3 Related Work

In this section, we discuss related work in two main groups: (i) approaches that apply MDE to defining and solving scheduling problems, and (ii) approaches that introduce a DSL to simplify validation and configuration by domain experts:

Applying MDE to Scheduling Problems

An increasing number of approaches to solve scheduling problems make use of MDE techniques. Some of these approaches rely on proprietary formalisms such as Ptolemy II [BLL+07], which is used for functional modelling, or Metropolis II [DDG+13], which is used for architectural modelling. Metronomy [GZN+14] integrates these two through a mapping interface and enables timing verification as well as design space exploration. Other approaches are based on industrial standards, e. g., AADL [FGH06], ADL [WRT+13], or UML\MARTE [HPP+14], which have the advantage of being more accepted in industry but often lack formal verification mechanisms.

There are also works that combine both cases by transferring standard models to a methodology where formal methods can be applied. Eder et al. use SysML to model different viewpoints onto a system for which a formally defined DSL exists, and can be used to express constraints, requirements and objectives for scheduling [EZV+17]. OpenMETA is a design tool suite for CPSs which is based on the model integration language CyPhyML for which different adapters exist [SBN+14].

Another MDE approaches leverage model transformations for scheduling problems: Al-Dakheel et al. [ADAA17] transform their software component models into Coloured Petri Nets (CPNs), for which allocation solutions already exist such as the Octopus Toolset [BvBG+10]. Different CPN tools are integrated for stochastic simulation of timed systems, model checking and schedule optimisation.

Our approach, in contrast, applies the fault-tolerant consistency management framework to compute an optimal allocation. One domain model each is created for human resources and testing tasks, such that the computed correspondences can be regarded as an allocation model.

Simplifying Validation and Configuration

In order to ultimately solve the scheduling problem, various optimisation techniques can be used such as SMT solvers [TWV+19], ILP solvers [TLWS18, PH19, KPP+15], or genetic algorithms [HDM05, AM01, SCV13].

While there are frameworks that come with a set of predefined constraints for specific domains such as Archeopterix [ABGM09], or the approach of Zverlov et al. [ZKC16], it is often difficult to extend these approaches with new constraints as domain experts must be familiar with different optimisation techniques. Various approaches introduce, therefore, a new DSL to simplify constraints definition, most of which are limited, however, to specific domains. AAOL [KP14] is designed for the automotive domain, while SAOL [KPP+15] and DSE-ML [EZV+17] are tailored to embedded systems. The approach of Pohlmann et al. [PH19], in contrast, provides a framework based on the OCL that is not limited to a particular domain. DESERT [NSKB03] is a domain-independent tool chain for defining and exploring search spaces that also uses OCL to specify constraints.

Similar to the work of Tomaszek et al. [TLWS18] applied to the domain of virtual network embedding, our approach leverages TGGs and is therefore domain independent. While OCL is a constraint language, TGGs represent a rule-based approach, specially designed for expressing the consistency relation between two modelling languages. The use of TGGs permits a more compact problem definition in the considered application scenario, compared to an equivalent definition with OCL constraints. While the original approach focusses on scalability and pruning the search space via pre-solving, a fine-grained cost function and guarantees for optimality were added in subsequent work [Tom21]. The cost function is defined in terms of extended OCL constraints, whereas we introduce an additional configuration DSL to further simplify this task for the domain expert by clearly separating problem and solution domain.

Finally, there are other approaches that attempt to prune the search space to increase the runtime performance. Kang et al. [KJS10] use user-defined equivalence functions in order to specify so-called symmetry breaking predicates. Combining these with SMT solving, they can ensure that every new solution is non-isomorphic to the previous one. The approach is neither applied to a scheduling problem nor evaluated in a real-world setting, though.

12.4 Test Schedule Optimisation via Correspondence Creation

Our approach to optimal scheduling is implemented as a variant of the CC operation presented in Sect. 5.3. While input models for human resources (source) and testing tasks (target) are provided as input, the operation computes an optimal schedule as an *allocation model* for software testers and testing tasks. In contrast to the CC operation in its original form, the objective function is more fine-grained to take aspects like the priority or risk of each specific task, or the suitability of testers for a particular task area into account.

A conceptual overview of this process is depicted in Fig. 12.1. We identify three distinct roles, which are well-aligned with the user roles for fault-tolerant software systems in Sect. 1.2: (i) a *test manager* (taking the role of the domain expert and integration expert), (ii) a *tool developer*, who developed the test schedule generator for the test manager, and (iii) a *meta-tool developer*, who provides generic tooling used by the tool developer.

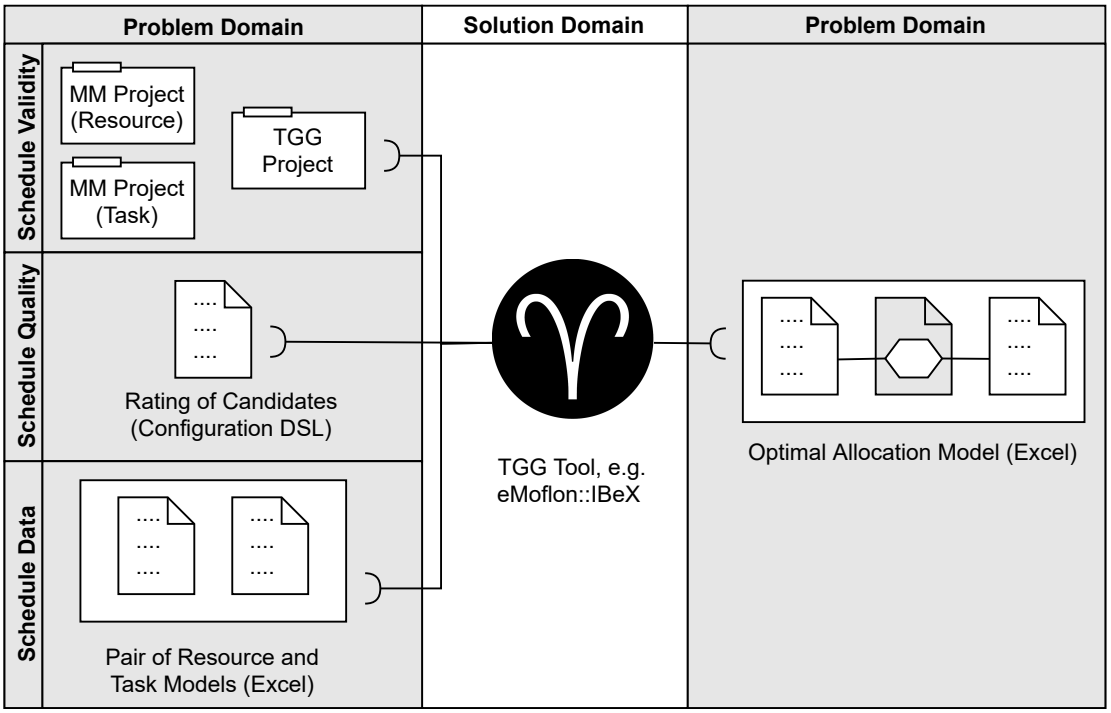


Figure 12.1: Conceptual overview of the test schedule generation process

Going through Fig. 12.1 from left to right: the test manager (probably in close collaboration with the tool developer) provides a triple metamodel and a TGG to define test schedule validity. These two artefacts are implemented using a suitable front-end as provided, e. g., by the TGG tool eMoflon (Ch. 9). The test manager then encodes domain knowledge as ratings, deciding which rules represent better allocations than others, which can take task- or tester-specific information into account. These ratings are specified using a DSL that was created for exactly this task (explained in detail in Sect. 12.7). The ratings are then used to refine the objective function to be maximised, which in this case represents the quality of a test schedule. As a final artefact, the test manager provides the schedule *data*, i. e., a resource model containing all people with their availabilities and suitabilities, and a task model containing all testing tasks and desired executions.

To simplify the evaluation of (and migration to) the automated solution using existing resource and task models, the schedule generator accepts these models encoded as Excel tables. A conversion to the input format expected by eMoflon (XMI or eMSL) was im-

plemented as part of the project. All input artefacts (converted to appropriate formats) are passed on to the back-end of the TGG tool. In this application scenario, the solution yielded by the CC operation represents an optimal test schedule, which is converted to an Excel table in the problem domain as the final result of the process.

In general, it is impossible to cover all desired executions of all testing tasks; the allocation model is a "best possible" schedule with respect to the provided validity rules and rating functions. This means that tasks of high priority and criticality should be done as early as possible and by an employee who is most suitable for this task with respect to their skills, while hard constraints such as availability and a fixed number of recurring test runs have to be respected. In this regard, the use cases demonstrates how fault-tolerant software systems can be used to solve optimisation problems in a model-driven way. While the input models should not be considered faulty (as they reflect the reality in an appropriate manner), a "perfect match" between them is nearly impossible, such that a flexible solution approach is required.

12.5 Domain Analysis via Metamodelling

To provide an overview of the input data provided by dSPACE, as well as the expected output data (a test schedule), we start by presenting a small but representative example.

Illustrative Example

One of the inputs is a table (as Excel file) encoding the available human resources, their availability over a series of consecutive weeks, areas they are responsible for, and their experience in all areas. Table 12.1 depicts such a table consisting of three people: `Anton`, `Betty`, and `Carl` (first column). The next three columns show their availability in hours for three weeks (`W1, W2, W3`). In this example, `Anton` is *responsible* for an area called `Dialogues`, `Betty` for `Code Generator`, and Carl for `Diagrams`. A person is responsible for an area if (s)he can be contacted to discuss and approve change requests in the area. The last three columns, one for each area, show the *suitability* of the three people in testing these areas.

Note that some values here are missing as it is sometimes difficult or even impossible to obtain a reliable estimation (new people or a new area). A person is suitable for testing an area if it is advantageous (from a test planning perspective) to assign the person to testing tasks in the area. This does not directly correlate with experience in testing this area – inexperienced interns, for example, are very suitable for testing a simple area that requires little prior experience (while experienced developers are *not* suitable for such an area).

Person	W1	W2	W3	Responsibility	Diagrams	Dialogues	Code Gen.
Anton	6h	14h	15h	Dialogues	--	80%	10%
Betty	31h	8h	31h	Code Generator	80%	--	90%
Carl	8h	23h	0h	Diagrams	60%	75%	--

Table 12.1: Available human resources and their characteristics

The second input is a table (also provided as an Excel file) representing the testing tasks to be accomplished in the testing phase, together with their characteristics and all constraints that are to be taken into account. Table 12.2 depicts such a table consisting of

three testing `Tasks`, the `Area` to which each task belongs, how long it approximately takes to perform the task (`Duration`), how important the test is (denoted as its `Priority`), the minimum suitability requirements of the task (`Suit.`), and the desired number of times the task should be repeated in the testing phase (`# Exec.`).

Task	Area	Duration	Priority	Suit.	# Exec.
Test Diagrams	Diagrams	8h	Medium	40%	2
Test Dialogues	Dialogues	5h	Low	40%	3
Test Code Generator	Code Generator	10h	High	75%	4

Table 12.2: Testing tasks together with their characteristics and all relevant constraints

Given two such tables as input, the test manager has to produce a *test schedule* assigning testers to a number of testing tasks in each week. Table 12.3 depicts such a schedule for our simple example, assigning the three testers to testing tasks in the three weeks. Our example already demonstrates that it is impossible to fulfil all constraints: the `Test Code Generator` task should ideally be repeated four times, but only `Betty` is suitable enough, and she does not have enough time in `Week2` (8h < 10h). As realistic test schedules range over 8 – 10 weeks, with about 250 testing tasks, and involving 20 – 30 people, creating and maintaining "good" testing schedules manually is indeed challenging. This, again, motivates the use of a flexible approach that does not enforce that all tasks are executed as as often as planned, but rather aims at determining the best possible schedule.

	Week1	Week2	Week3
Test Diagrams	Carl	Betty	--
Test Dialogues	Anton	Carl	Anton
Code Generator	Betty	--	Betty

Table 12.3: A feasible test schedule for our running example

Applying Metamodelling

While a tabular concrete syntax (see Tables 12.1, 12.2, and 12.3) allows for a compact representation of our input and output data, applying our hybrid, model-driven approach requires a representation of the relevant concepts and relations in form of models and metamodels. In the context of test scheduling, we define a source metamodel for modelling the resource availability, and a target metamodel for describing testing tasks. The correspondence metamodel represents an allocation of software testers and tasks.

Figure 12.2, simplified for presentation purposes, depicts a triple of metamodels, capturing the most important concepts and relations in our problem domain. In accordance with Fig. 3.8, multiplicities are only shown when they are not `0..*`.

In the source metamodel, a `Person` can be responsible as well as suitable (have a `Suitability` with a certain value) for multiple `Areas`. People can also be available (have an `Availability` in hours) for testing in certain `Weeks`. Available testing `Environments` fix the operating system and the versions of all installed software used for testing.

In the target metamodel, `TaskAreas` are used to group testing `Tasks`, each with numerous attributes representing characteristics (e. g., duration, priority) and constraints

(e. g., minSuitability) for the task. Every task also specifies the desired number of times the task should be executed in the testing phase (`Executions`).

The correspondence metamodel consists of four correspondence types: `Related` connects the corresponding areas in source and target models, while `ExecEnv` connects every execution to the environment used for testing. This execution/environment correspondence is useful, as repeated executions of the same testing task should use different testing environments if possible. `Allowed` and `Allocated` identify people who can potentially be assigned to executions of a task, and the person finally allocated to an execution.

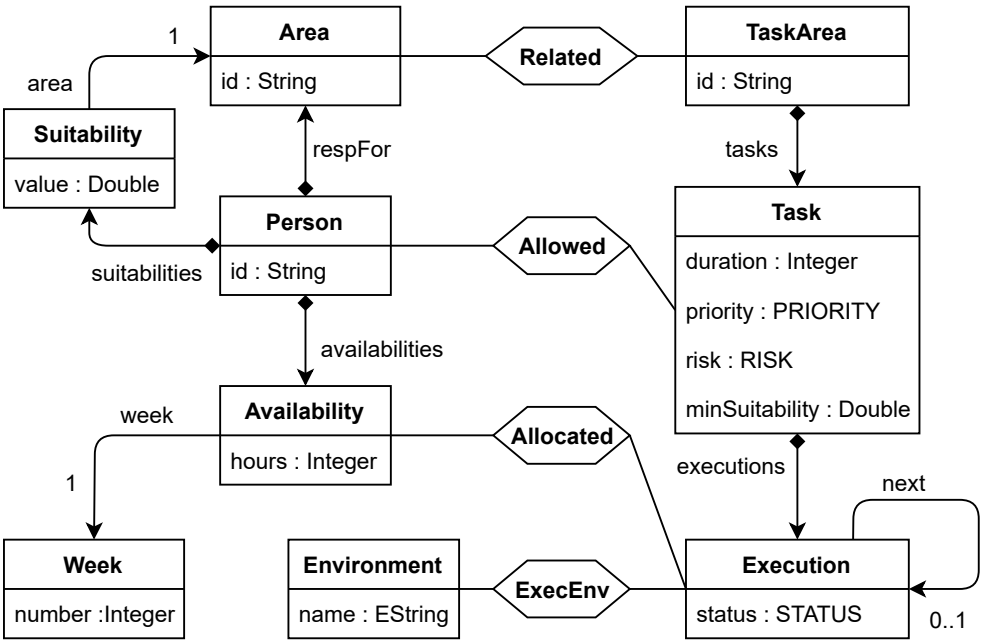


Figure 12.2: Source, correspondence, and target metamodels

Besides the triple metamodel depicted in Fig. 12.2, a set of TGG rules needs to be defined in order to fully specify the "consistency" relation, i. e., valid allocations of human resources and testing tasks. As the presentation of the entire rule set is out of scope for this thesis, we restrict ourselves to the two most interesting rules in the following Sect. 12.6.

12.6 Defining Test Schedule Validity via a TGG

We first formalise the validity of test schedules by specifying consistent schedules using a TGG. Due to the large amount of rules that were necessary to express the relation between human resources and testing tasks, only the most important rules are presented (in a simplified version) as examples. All elements in rules are typed according to the triple metamodel of Fig. 12.2. For the specific use case, we are primary interested in the CC operation for correspondence creation, which was introduced as one operation of the hybrid framework in Sect. 5.3. Therefore, the example rules are depicted in their operationalised form for CC.

Figure 12.3 depicts a TGG rule *FirstExec* allocating an availability `a` of a person `p`, who is allowed to execute the corresponding task `t`. According to the rule, this should only be possible when the availability of the person `p` is sufficiently long, i. e., when `a.hours` $\geq$ `t.duration`. The rule also picks out an environment `e` in which the task is to be executed.

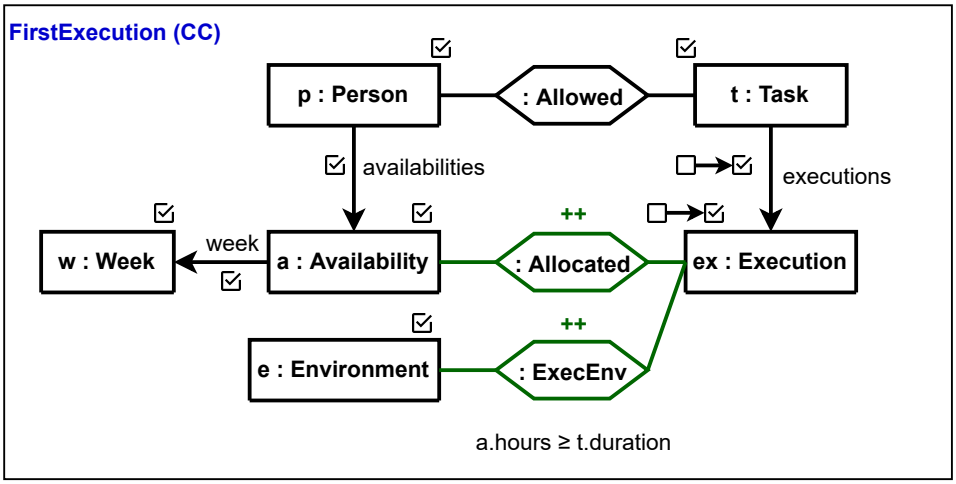


Figure 12.3: Marking a first execution of a task (FirstExec)

An important point to note here is that the granularity of availabilities per week (in hours) must be chosen carefully to fit to the granularity of execution times for testing tasks (also in hours). This is because the allocation problem was simplified to enable manual schedule creation by allowing only one testing task for each availability slot. In the TGG implementation at dSPACE, we decided to maintain this simplification as existing input data was already in this form, and as it also fits well to TGG rules, which are inherently local in nature. To drop this restriction, an additional "global" validity constraint, such as the negative constraint of Fig. 12.4, would have to be used to ensure that the total duration of all tasks assigned to the same availability remains less than or equal to the length of the availability. The depicted constraint can only restrict the duration of two executions `ex1` and `ex2`, though. Up to some number n, one could define more constraints of this form, and restrict the number of executions per availability to at most n with a further negative constraint.

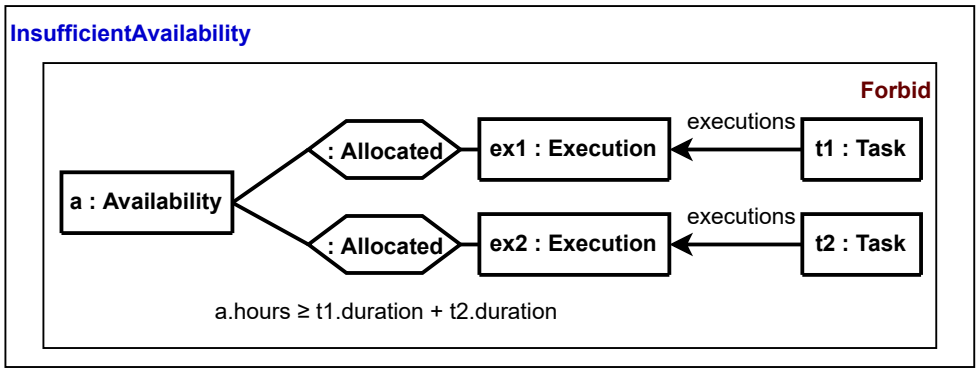


Figure 12.4: Negative constraint for guaranteeing sufficient availability

The second TGG rule *FurtherExec* we wish to discuss is depicted in Fig. 12.5. This rule marks additional executions for a task `t`. This is accomplished by demanding a previous execution `lex` and marking a new execution `ex` as its `next` execution. Note that there can only be at most one next execution according to the target metamodel. As additional executions for the same task are "costly", they should be conducted in a way that the test effect is maximised. This is formalised in the source domain of the rule by (i) making sure that a different environment `e` $\neq$ `le` is used, (ii) making sure that a different person

`p` $\neq$ `lp` is chosen, and (iii) making sure the test is executed in a different week (`w` $\neq$ `lw`). Note that this rule can mark multiple executions in different weeks at a varying distance (`w.number` $\geq$ `lw.number + c`) from the last execution week (`lw`), whereby `c` is a configuration parameter for this distance. Choosing the most advantageous week for a repetition requires domain knowledge and is therefore not fixed by the TGG, but determined by the target model at hand.

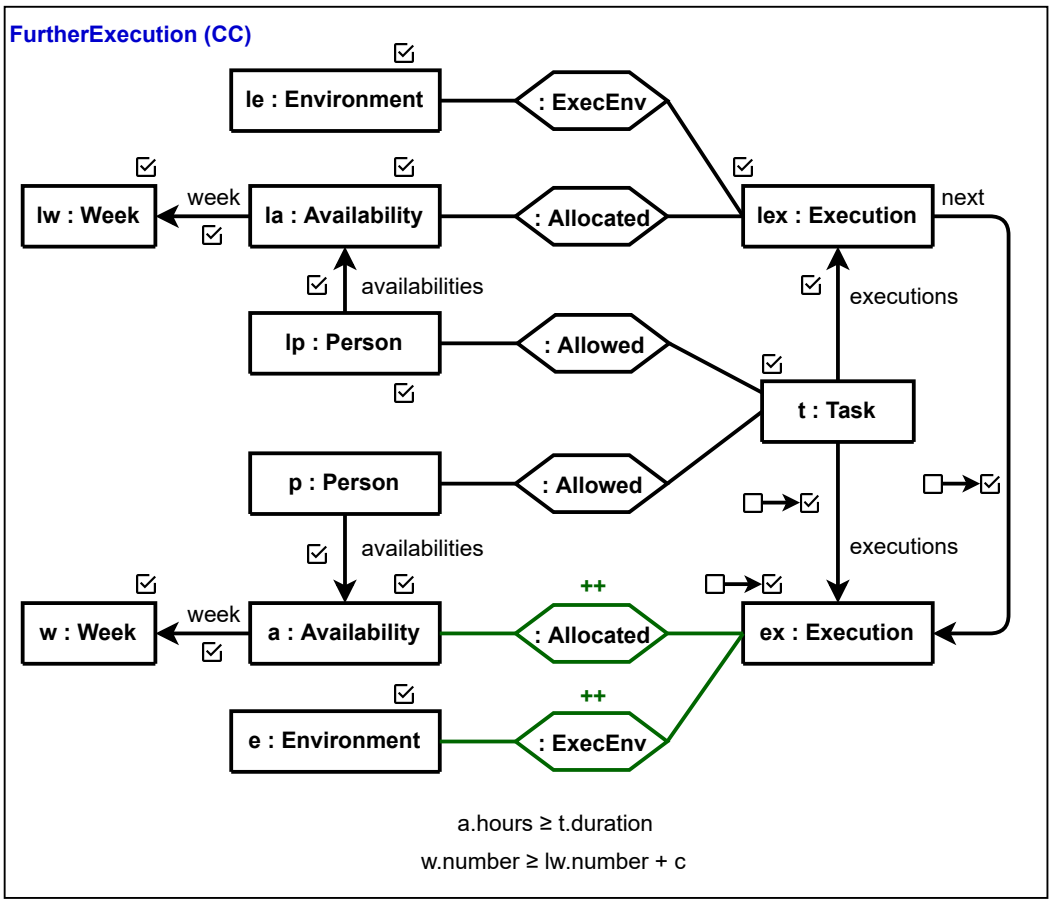


Figure 12.5: Marking further executions of a task (FurtherExec)

The CC operation in its original form (cf. Sect. 5.3) aims at maximising the number of marked elements, such that inter-model consistency can be checked by trying to mark the input models entirely. Applied to the scenario at hand, this strategy would lead to solutions in which the overall number of executions of *all* testing task is maximised, regardless of whether, e. g., an execution belongs to a high-priority task or not. Also, the task would be assigned to *any* person whose suitability is sufficiently high to take this task, independent of whether there are other available testers with a higher suitability value. While this strategy would indeed generate *valid* test schedules (as all "hard" constraints such as availability and minimum suitability of testers are satisfied), further information about the tasks, such as priority and risk, should be taken into account as well, such that not only *valid* but also *good* schedules are computed. Test schedule *quality* is yet to be defined and will be discussed in the following sections.

12.7 Configuration via a Domain-Specific Language

Lightweight configuration was a primary motivation for our approach; our domain and integration expert (the test manager) expressed a strong preference for configuring the process by providing different weights for rule applications (cf. the specification of *ratings of candidates* in Fig. 12.1). For the specific use case, the determination of an optimal test schedule can be formalised as follows:

Definition 12.1 (Optimal Test Scheduling)**.**
Given a starting triple graph G_0, a TGG $(TG, \mathcal{R})$ and a schema $(TG, \mathcal{GC})$, let $D : G_0 \overset{*}{\Longrightarrow} G_n$ be a derivation via operational rules for CC. Let $rating : \mathcal{D} \to \mathbb{Q}$ be a rating function for rule applications. The ILP to be optimised is constructed as follows:

$$max. \sum_{d \in D} rating(d) \cdot \delta \ s.t.$$
$$markedAtMostOnce(G_0) \wedge context(D) \wedge context(G_n) \wedge acyclic(D) \wedge sat(G_n)$$

Compared to the definition of the generic optimisation problem (Def. 6.7), the number of marked elements as coefficient for each rule application candidate is replaced by a *rating* function that assigns each rule application an individual value. Weighting via the rating function is essentially a *local* decision as it can only access the structure referenced by a rule application, similar to the rating of rule application candidates for the concurrent synchronisation operation (cf. Sect. 7.6). The scope of this local decision can, however, be increased by specifying rules such as *FurtherExec* that range over allocations created by other rule applications; this comes at the price of reduced scalability as it substantially increases the number of candidate allocations.

To provide an impression for the design of the rating function, we now discuss some heuristics from our concrete use case for rating the creation of executions, i. e., rating rule applications of *FirstExec* and *FurtherExec*.

The rating of rule applications of *FirstExec* should:

- be directly proportional to the priority, risk, and duration of the task (cf. matched context object `t` in Fig. 12.3). The rationale here is that such tasks are more important than others and their scheduling should thus be preferred.
- be indirectly proportional to the number of the week (cf. `w` in Fig. 12.3). The rationale is that all tasks should be scheduled as early as possible to remain flexible in later phases.
- be directly proportional to the suitability of the allocated person. The rationale is that the most suitable people should be assigned to allowed tasks.

The rating of further executions created by *FurtherExec* should

- be directly proportional to a problematic status of the last execution `lex` (if its status has been updated during maintenance of the test schedule). The rationale here is that failed executions indicate problems that should be revisited.
- be indirectly proportional to the divergence from an *ideal distance* between the weeks of the executions (formalised as parameter `c` in the second attribute condition of Fig. 12.5). This ideal distance depends on the particular testing task and is specified by the test manager.

- be directly proportional to a change in suitability of the last person `lp` and the allocated person `p`. The rationale is that a further execution should involve a more (less) suitable person if a less (more) suitable person was chosen previously.

To simplify the specification of the rating function for the domain expert, a configuration DSL was developed for exactly this use case. Listing 12.1 depicts an illustrative excerpt of the rating function for the (simplified) running example. After importing a TGG, *rating* functions can be implemented for a choice of TGG rules. The name of every rating function must be the name of a TGG rule, and there can be at most one rating function for every TGG rule (supported by auto-completion and validation). A rating function returns a real number and can access all matched objects (and their properties) of a match of the corresponding TGG rule. For example, `t.priority` accesses the task object matched as context in *FirstExec* (cf. Fig. 12.3). This is again supported and enforced by the editor. The DSL was developed using Xtext[2] and is based on the expression language library Xbase,[3] allowing for Java-like expressions for flexibly calculating the required rating of a match. As access is restricted to the objects in a corresponding match and their properties, the locality of the rating function is enforced; it is impossible to navigate to other objects in the graph. The DSL was developed together with the testing manager and should be straightforward to use for anyone familiar with a Java-like language. When calculating the objective function value (cf. Def. 12.1), the $rating(d)$ for every application candidate d or a rule r is calculated by invoking the rating function for r, passing d as an argument.

```
1  import ".../schemaValidity.tgg"
2
3  rating FirstExec {
4    val prioFactor = switch (t.priority) {
5      case HIGH: 5
6      case MEDIUM: 2.5
7      case LOW: 1
8      default: 1
9    }
10   val riskFactor = switch (t.risk) {...}
11   val durationFactor = if (t.duration > 10){...}
12                        else {...}
13   val weekFactor = ...
14   val suitabilityFactor = ...
15
16   return prioFactor*riskFactor*durationFactor*
17          suitabilityFactor/weekFactor
18 }
19 rating FurtherExec {...}
```

Listing 12.1: Configuring the rating function with a DSL created for exactly this purpose

12.8 Applied Techniques to Improve Scalability

While there was no absolute maximum duration for the schedule generation process (a "few" hours would still be acceptable according to the test manager), a clear requirement was that the process should be executable on a standard desktop PC, implying in this context about 16GB of memory, 4 cores, and 3,6 GHz.

[2] https://www.eclipse.org/Xtext
[3] https://www.eclipse.org/Xtext/documentation/305_xbase.html#xbase-language-ref-introduction

The average problem size proved to be too complex for our graph pattern matcher and ILP solver. Compared to the BX benchmark examples that were used for the previous runtime evaluations (cf. Sect. 5.8, 6.6, 7.7, 8.5 and 9.6), the rules of the TGG at hand are substantially larger, making it necessary to match larger patterns to determine rule application candidates. As a result, the pattern matcher did not have enough memory to determine all candidate allocations. It is important to note that eMoflon::IBeX was used for the implementation at dSPACE, which means that problems which occurred and improvements which were subsequently implemented are tailored to graph pattern matching on EMF models. The use of different technologies, such as graph databases as part of eMoflon::Neo, could have led to less or different problems and solutions, which is left to future work at this point.

To address the problem of insufficient main memory, we analysed the TGG rules to *identify a suitable partition* of the model, so that the pattern matcher could be restarted and executed for each partition independently. We also *decomposed* some rules to avoid an explosion of combinations. After the first phase of our test schedule generation process was executed successfully, the next challenge was solving the resulting ILP. To make this task tractable, we *split the problem* into sub-problems and solved these successively. In the following, we discuss these three general techniques in some more detail.

Identification of a Partition

A static analysis of the TGG rules can be used to identify sufficient conditions under which rules can be applied *independently*[4]. This information can be used to speed up the pattern matching process by running multiple instances of the pattern matcher in parallel. As our problem, however, was more related to limited memory, we used the dependency analysis to run the pattern matcher on multiple "parts" of the models, terminating and restarting the process after each part was completed.

The partition we identified (based on the structure of the TGG rules in use) and exploited for this was based on the areas and corresponding task area of the resource and task models, respectively. It is possible to restrict allocations to a single area and task area combination, and to compute all possible allocations that result from this correspondence on the level of areas. When all candidate allocations have been computed, the next area correspondence can be established and handled. This area-wise collection of candidates saves memory as the internal state of the pattern matcher can be freed after every area combination.

Rule Decomposition and Reduction

To reduce the number of candidate allocations that have to be determined by the pattern matcher, TGG rules can be *decomposed* to avoid an explosion of combinations. As a concrete example, *FirstExec* could be separated into two separate rules: one rule creating the correspondence link between availability and execution, and another rule creating a correspondence link for all possible environments. The pattern matcher thus only determines n candidates for the first rule, and m for the second, i. e., $n + m$ instead of $n \cdot m$ candidates for the combined rule *FirstExec*.

While this change has the positive effect of reducing the number of candidates determined by the pattern matcher (and consequently required memory), it also has a price: (i) additional ILP constraints have to be formulated to ensure that the separate variables

[4]For details about sequential and parallel independence, the interested reader is referred to Ehrig et al. [EEPT06, pp. 47-64]

are handled as a single variable, i. e., either all set to 0 or 1, and (ii) it becomes more difficult to rate the combined match as the rating function has to be decomposed as well. Another means of reducing the size of candidates is by removing context elements in rules: *FirstExec* can be reduced in this manner by omitting the week. Consequently, separate candidates are no longer collected for different weeks (saving memory) but can no longer be rated differently.

Splitting the Problem into Sub-Problems

As a final technique, now for addressing scalability problems of the ILP solver, we experimented with different ways of "splitting" the problem into sub-problems. Possibilities we considered included (i) using a maximal *time window*, i. e., creating a plan for two weeks, fixing the determined allocations, and then solving the next two weeks, etc., (ii) using an analogous *task window*, and (iii) using an analogous *people window*. While all three windows made the problem tractable for our ILP solver, this is done by sacrificing optimality, which can no longer be guaranteed for the entire problem. Although the use of heuristic solvers would be a possible alternative as soon as receiving an optimal solution is no hard requirement, the experiments of Sect. 8.5 indicate that this would probably not lead to the desired runtime improvements.

For our concrete use case, experiments showed that the time window was the best solution, i. e., still produced relatively good schedules compared to task and people windows. An explanation is that such a split requires a suitable ordering of weeks, tasks, or people. While weeks are naturally ordered, it is unclear how to order tasks or people in an optimal manner. Naïve attempts using single attribute values (risk, suitability, priority) yielded much worse schedules than the natural temporal order for the time window.

12.9 Evaluation

Our approach was successfully implemented as a fully automated test schedule generator, based on eMoflon::IBeX, which is now in productive use at dSPACE replacing manual test schedule creation and maintenance. The TGG consists of 10 rules in total, and the models describing the test schedule conform to an extended version of the metamodel depicted in Fig. 12.2.

To evaluate our approach, we conducted a semi-structured interview with the test manager at dSPACE, and were granted access to data from using the test schedule generator during one testing phase. We were also provided with two previous, manually generated test schedules, which we compared to corresponding schedules automatically generated using our tool. The three test schedule instances incorporate about 8-10 weeks, 250 testing tasks, and 20 – 30 people. This sections summarises the results of our qualitative and quantitative evaluation, which was already presented in more detail in previous work [AWO+20, Opp18]. We investigate the following research questions:

RQ1 How challenging is the task of validating, configuring, and adapting the test schedule generator as perceived by the test manager (the domain and integration expert)?

RQ2 Are generated schedules of comparable or even higher quality than manually created schedules? How much time and effort can be saved via automated test schedule generation?

RQ3 How time- and memory-consuming is the generation process for test schedules of realistic size?

Assessment by the test manager

The following summarises the main findings from our interview with the test manager. The complete interview is available online[5].

Adaptability (RQ1): The test manager can easily configure the ratings for allocation candidates using the configuration DSL, even without knowledge about technical and implementation details. The weighting function resembles the formula used for manual planning, such that most ratings could be directly transferred from past values used in the manual process. Resource and task models can be provided as Excel sheets, such that adapting the input artefacts is not required. After a brief introduction to TGGs comprising the syntax and semantics of rules and model instances, the rules defining the validity of test schedules can be understood and validated by the test manager. Changing existing or adding new rules is, however, a non-trivial task for the test manager as understanding the consequences of such changes requires knowledge of dependencies between rules and how such changes can affect, e. g., the efficiency of the entire process. An important requirement is being able to adapt the generated plan during the testing phase, taking current changes into account. This is well supported as additional, simple constraints can be added to fix all allocations in past weeks and thus allow a regeneration of the plan for the remaining weeks.

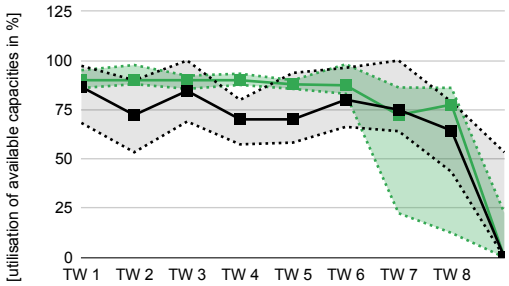
Quality (RQ2): The schedule generator creates valid test plans, guaranteeing that all hard constraints are fulfilled. In all experiments, it was always possible to schedule all tasks that could be scheduled manually (i.e., only tasks which lack suitable resources remain unplanned). Concerning other quality metrics: the workload was well-balanced among testers, their capacities were well-utilised with values near to the maximum especially in the early test weeks (cf. Fig. 12.6a), and the suitability of testers for tasks was remarkably well-considered. As desired, the schedule generator preferred time-consuming tasks with a high priority or risk (cf. Fig. 12.6b), and employees with more experience were assigned more complex tasks. The preferred distance between multiple executions is also well-considered. Overall, while the generated schedules are not perfect (neither are manually created schedules), they are of sufficiently good quality for productive use, such that the software solution will replace the manual process for future release cycles.

Time Consumption (RQ3): The generation of a test schedule with a duration of 9 weeks takes about 30 minutes on average, compared to an estimated workload of eight hours for manual creation. For both generated and manual schedules, approximately one hour per week is still required for manual adaptations (e.g., due to testers falling sick, or when tasks require much longer or shorter than expected). This means that the overall effort required for manual test schedule generation and maintenance is about $8 + 9 = 17$h, compared to $0.5 + 9 = 9.5$h when using the schedule generator. This is thus a reduction of about 50%. This is also a conservative estimate, as a human being will have increasing difficulty to cope with larger schedules, while the automated solution will remain predictable and consequent, especially during maintenance.

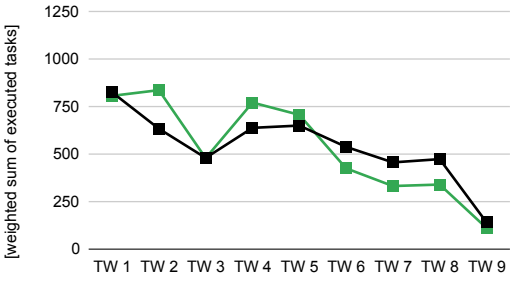
Resource Usage

To obtain quantitative data for RQ2, we considered the utilisation of testers' capacities in the course of the testing phase. The goals related to utilisation are threefold: First, as much of the testers' capacities as possible should be utilised so that as many tasks as possible can be completed. Second, efforts should be distributed equally among testers

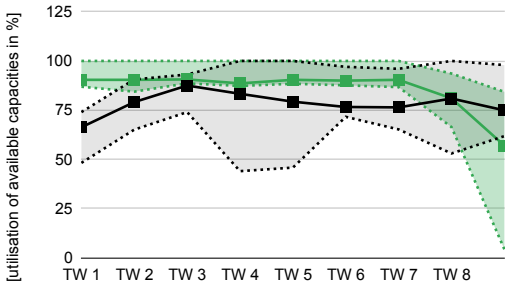
[5] https://drive.google.com/file/d/1sU6DYwdXQ3x4ennjM1PDW0883XOHZjIi



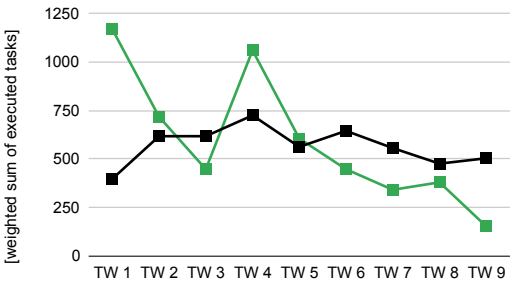
(a) Resource usage: Generated schedule



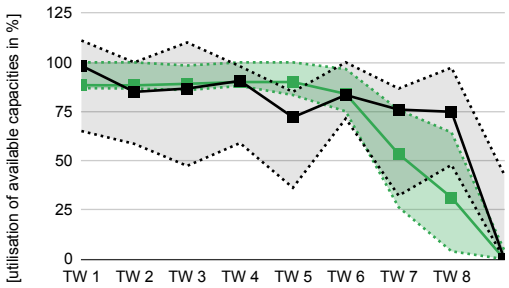
(b) Task priorities: Generated schedule



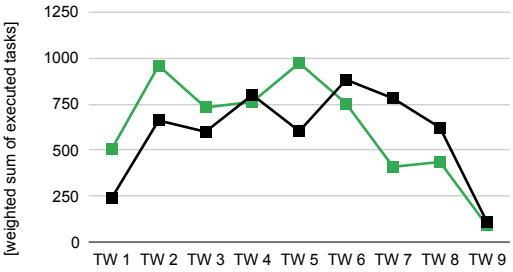
(c) Resource usage: Previous schedule A



(d) Task priorities: Previous schedule A



(e) Resource usage: Previous schedule B



(f) Task priorities: Previous schedule B

Figure 12.6: Evaluation Results: Schedule Quality

to ensure the process is as fair as possible. Third, to be able to react to unforeseeable circumstances, a time buffer should be left at the end of the schedule.

Figure 12.6a depicts the utilisation of testers' capacities in the initial plan generated by the tool (green), and the generated plan that includes all adaptations made during the testing phase (black). The solid lines represent the median value over all testers, whereas the dashed lines show the first and third quartile. One can observe a high concentration of tasks in the first six weeks of the generated plan. This is desirable because testers' capacities are well-utilised, leaving as much buffer as possible towards the end of the schedule. While the adapted plan shows less utilisation due to illness, unexpected bug fixing, and other unexpected problems, the workload still decreases towards the end of the testing phase.

For two previous testing phases, we compared generated schedules to the corresponding, manually created schedules (cf. Fig. 12.6c and 12.6e). As we only had access to (i) the final resource and task models, and (ii) the final state of the manually created and maintained schedule, our comparison is of course strongly biased towards the generated schedules,

which are optimal for the given pair of models. This threat to validity is further discussed towards the end of this section. Even if this is to be expected, one can still observe that the automated generation outperforms the manual process with respect to quality: The fact that the generated schedules are better than the manually created schedules is still a confirmation that the schedule generator produces schedules of high quality: Human resources are better utilised in the first six weeks in the generated plans, whereas this value noticeably drops in the last three weeks, leaving a buffer for unexpected effort or for releasing human resources for other projects. Furthermore, one can observe that the spread amongst testers (visualised via the dashed lines) is smaller in the generated plans, which means that a fairer task distribution is achieved.

Task Distribution by Priority and Risk

Another quantitative measure for schedule quality (RQ2) is the degree to which the schedule generator prefers tasks that are risky and/or highly prioritised. The test manager categorises tasks as *high*, *medium* or *low* for both risk and priority. The corresponding factor (cf. Sect. 12.7) is set to 5, 2.5 or 1, respectively. Figures 12.6b, 12.6d, and 12.6f depict the sum of testing tasks per test week weighted with these factors. One can observe that tasks with high priority and risk are executed in the first week. As there is a repetition typically after at least three weeks, many highly prioritised and risky tasks are also (re)executed in week 4 (explaining the second peak in the plots). Depending on how many such tasks are still left, there can be a further peak at week 6. The generated test plans clearly show a stronger preference for tasks with high priority and/or risk in the early weeks, with much higher peaks and a lower curve towards the ending of the test phase, indicating that all risky and prioritised tasks have already been allocated.

Performance

During test schedule generation, CPU usage and main memory consumption was measured to determine performance bottlenecks and assess the feasibility of schedule generation on a common laptop or desktop computer.

Setup: A notebook with an Intel Core i7-2600 CPU and 16 GB of memory of memory was used to generate a schedule for a test phase. Windows 7 in the 64-bit edition was used as operating system, with Eclipse Photon and JRE 10.2 as execution environment. The memory available for the JVM was limited to a maximum of 12GB.

CPU usage: During the pattern matching step, the CPU usage was comparably low (ca. 16%), with only a small increase visible after a set of determined matches have been processed and a new incremental pattern matching step starts. In contrast, the CPU is almost fully utilised when solving the ILP problem due to multi-threading. Using a graph pattern matcher that can parallelise the pattern matching step would thus speed up the process.

Memory consumption: During the pattern matching phase, strong fluctuations between pattern matching (high memory consumption, 1.8 – 2.8 GB) and match processing (low memory consumption 0.2 – 1.0 GB) can be observed. When the ILP solver is invoked, the memory consumption remains at a constant level of about 2.5 GB. This indicates that the assigned memory is sufficient to generate schedules of comparable size, observing a positive effect of the optimisations described in Sec. 12.8, and is therefore not a bottleneck.

Required time: The time required for test schedule generation was measured separately for pattern matching and ILP solving and repeated 20 times. As the Gurobi solver was

excluded by dSPACE due to its high amount of recurring cost, the CBC solver[6] was used in the concrete implementation. The average time for pattern matching was approximately 16.6 minutes with little spread. In contrast, the time for solving the ILP varied from 10 to 32.5 minutes, with a median of 16.5 minutes.

Summary

Revisiting our first research question (RQ1), we feel confident to claim that the problem definition with our approach can be fully validated by the test manager without requiring in depth knowledge of the solution domain. It can also be easily configured to a large extent by the test manager, even though changing and adding new TGG rules still requires assistance from the tool developer. As the validity of the allocations is not to be adapted as often as the rating functions for optimisation, this is not a severely limiting factor.

Concerning the quality of generated schedules and effort saved (RQ2), the quantitative quality metrics corroborate the qualitative feedback from the test manager, showing that generated schedules are comparable if not better than manually created schedules. The automation saves at least 50% of working time for the test manager per test phase; this value is probably higher as manual maintenance is bound to be more error prone, especially for larger schedules.

The performance measurements (RQ3) show strongly varying memory consumption (0.2–2.8 GB) and moderate CPU utilization (15-30 %) during the pattern matching phase. In the optimisation phase, a memory consumption of 2.5 GB and a CPU utilization of almost 100 % imply that the bottleneck is currently the CPU during the optimization phase. Probably, the runtime can be substantially improved by using multiple cores for the pattern matching step. The generation process takes about 35 minutes on average, whereby the longest test run took 50 minutes. This is acceptable, as test schedules are generated once a week on a single machine, and do not require user interaction.

Threats to Validity

We now discuss the most important threats to the validity of our evaluation and results and how we attempted to mitigate them:

Internal Validity: As the test manager was closely involved in the project, it is fair to assume a positive bias towards the final solution. Especially the qualitative assessment of our solution can be (very) different for a domain expert who is unwilling to learn and use a new DSL, or who feels threatened by the automation. The interview was conducted in German and translated to English. To avoid any mistakes that could have occurred during the translation, we asked the test manager to review the final transcript.

As we could only compare manual schedules containing adaptations made over the entire testing phase with generated schedules based on the final resource and task models, it is clear that the generated schedules should be superior. Our results have to be thus carefully interpreted; they only indicate that generated schedules are of comparable quality and can be productively used as a replacement for manual schedules. While the performance measurements indicate that real-world examples can be solved within an acceptable time frame, the effectiveness of the optimisation techniques proposed in Sect. 12.8 are not experimentally evaluated. Finally, to provide a high-level overview of how schedules take the risk and priority of tasks into account (cf. Fig. 12.6b, 12.6d, 12.6f), we have treated risks and priorities as interval and not as ordinal data. While we have clarified this with the domain expert, it is questionable if this makes sense for actual risks/priorities.

[6] projects.coin-or.org/Cbc

External Validity: As we have interviewed only a single domain expert, and applied our approach to a single project at a single company, we cannot claim that our positive results can be directly transferred to a different context. We were also only able to apply the schedule generator for the creation of three test schedules up until now. Additional tests would be useful to further validate the results. Finally, while we initially experimented with various ILP solvers, the performance results depend substantially on the choice of graph pattern matcher and ILP solver, which were not varied in the experimental setting.

12.10 Summary and Discussion

After having presented an industrial use case that leverages TGG-based forward transformations and backward synchronisations in Chap. 11, we propose a configurable, model-driven approach to optimal scheduling that is an application of the correspondence creation (CC) operation. Going beyond its use for consistency checking, the produced correspondence model has a content-related meaning in this case as well, as it can be regarded as an *allocation model*. A TGG that maps human resources to testing tasks is used for specifying the validity of test schedules in a rule-based, high-level manner.

In order to be able to not only assess the validity but also the quality of the test schedule, the original form of the correspondence creation operation is extended by a configurable rating function that assigns each rule application candidate a specific weight in the ILP's objective function. For rating these candidates, we established a new, additional DSL with a very narrow focus. We have shown that our approach not only produces test schedules of acceptable quality, but that the test manager is able to understand and validate the entire problem definition, and configure at least the rating functions with confidence. In this case study, the scalability of the hybrid approach was tested using a TGG of realistic size. The performance evaluation has shown that the implementation – depending on the concrete use case – requires further (heuristic) adaptions to scale sufficiently well for practical use, but also that our generic tool support offers good opportunities to do so.

With respect fault-tolerance, the case study discloses that a certain degree of flexibility is essential for solving allocation problems of this kind, because it is often not possible – and towards the end of a testing phase not even desired – to map each availability to a testing task execution. While it would be misleading to denote input models as inconsistent because they cannot be completed to a triple of the TGG's language in the general case, approaches which enforce strict consistency could not be used here. While the goal of this case study is to enable the test manager to produce an initial test schedule, it would be interesting to use the concurrent synchronisation operation (Chap. 7 and 8) to propagate updates on either of the two models, because the priority of tasks or the availability of testers likely changes during the testing phase.

Considering the use case at dSPACE, the Adosate TGG (Sect. 10.7) and the BX between SysML and Event-B (Chap. 11) as three case studies for applying (fault-tolerant) consistency management in practice, all of them have different characteristics and require situation-specific adaptations of the hybrid approach. Even this small range of examples emphasises that the requirements and challenges of potential use cases are very heterogeneous, such that there is an apparent need for further real-world examples in this regard. The last chapter of this thesis summarises the gathered results and sketches promising directions for further research.

13 Conclusion and Future Work

In this thesis, a hybrid approach to fault-tolerant consistency management in MDE was presented that synergetically combines Triple Graph Grammars (TGGs) as a declarative means of defining a consistency relation, and different optimisation techniques – especially Integer Linear Programming (ILP) – to determine a solution that is consistent to the largest possible extent. The conceptual framework was implemented as part of the eMoflon tool suite, and applied to two industrial case studies. This chapter concludes with a summary of this thesis' contributions in Sect. 13.1, structured on the basis of the requirements for a fault-tolerant consistency management system, which are posed in Sect. 1.2. Directions for future research are subsequently sketched in Sect. 13.2.

13.1 Requirements Revisited

This section maps the contributions of the thesis to the ten requirements of Sect. 1.2. In total, one can state that all requirements are adequately addressed, whereby manifold directions for further research have become apparent, which are discussed in Sect. 13.2 to conclude the thesis.

R1: Formal Basis

The central contribution of this thesis is the conceptual framework that combines TGGs and optimisation techniques such as ILP in a hybrid fashion. TGGs, as a declarative and rule-based BX approach, are based on algebraic graph transformation, inheriting its formal semantics. By means of graph pattern matching, a set of rule application candidates is formed, from which an optimisation problem is constructed. All construction steps are formally defined and guarantee language membership and schema-compliance for the computed solution. The well-defined combination of TGGs and optimisation techniques makes it possible to prove formal properties, such as correctness and completeness, for forward and backward transformations, as well as for consistency checks.

R2: Fault-Tolerance

While forward and backward transformations, model synchronisation and consistency checks have been specified in prior work already, the aspect of fault-tolerance has been insufficiently investigated. The requirement can therefore be regarded as the unique selling proposition of this thesis: By applying the operations of the hybrid framework, tolerance towards violations of intra- and inter-model consistency is achieved. Simultaneously, there is a guarantee that the output models are free of faults, i. e., comply to the given schema and are contained in the language of the underlying TGG. In case of inconsistent inputs, the largest consistent sub-triple is computed, and the set of remaining elements is returned separately. We claim that this output is valuable for the user, because model transformations in the presence of faults are possible, while both a consistent solution and additional information about open problems are returned.

R3: Multiple Operations

The hybrid framework offers a variety of consistency management operations, including FWD_OPT and BWD_OPT for forward and backward transformation, CO and CC for consistency checking with and without correspondence links, and CS for concurrent model synchronisation (which we drop for the rest of this paragraph, as R5 discusses this operation in detail). All operations share a common formal basis, including the representation of rule application candidates as variables, a set of constraint types that guarantee language membership and schema-compliance, and an objective function that strives to maximise the number of translated input elements. As a result, both the implementation and the proof of formal properties follow a common generic structure for all operations. Satisfactory scalability results could be observed for all operations in performance evaluations, with CO scaling best and CC scaling worst among them.

R4: Expressiveness

By adding further constraints to the definition of the optimisation problem, negative, positive and implication constraints are fully integrated into the conceptual framework. In Chap. 4, we have shown that graph constraints increase the expressive power of TGGs. In accordance with prior approaches, attribute conditions are supported by directly attaching them to TGG rules. For conducting the industrial case studies (Chap. 11 and 12), attribute conditions played an essential role to, e. g., encode whether a software tester is suitable for a specific task when creating test schedules. An experimental evaluation indicates that negative constraints are efficiently processable, whereas adding implication constraints leads to substantial increases in runtime.

R5: Concurrent Synchronisation

As one of only a few model-driven approaches, the task of synchronising concurrent updates on both models is supported by the hybrid framework. To the best of our knowledge, the additional support for graph constraints is a further novel contribution. The operation also encloses (one-sided) model synchronisation as a special case in which only one model is edited by a domain expert. Conflicts are resolved without defining them explicitly: Created, deleted and unchanged elements are assigned different weights in the objective function, which can be configured via parameters. This makes it possible to reject user edits in case other (more important) changes are prevented otherwise.

As optimality is less important and distinguishable for this operation, it seems promising at the first glance to replace the exact method of ILP solving by meta-heuristics in order to improve the overall runtime performance without larger efforts. Both single- and multi-objective optimisation heuristics were investigated, whereby the original problem definition could remain equal for the single-objective case. Splitting the objective function into multiple parts, in contrast, makes it even possible to omit the configuration parameters. Several problems became apparent, though: On the one hand, for the multi-objective case, the pareto-front consists of too many solutions to present them to the user, such that we strengthened the focus on single-objective heuristics. On the other hand, both the Genetic Algorithm (GA) and Simulated Annealing (SA) – as investigated single-objective heuristics – scale worse than ILP in comparable settings, which can be led back to the tiny portion of feasible solutions in the search space.

R6: Generic Tool Support

All concepts of the hybrid approach (Chap. 5 - 8) are implemented as part of eMoflon, which is a Java-based MDE tool suite supporting metamodelling, unidirectional and bidirectional model transformations based on graph transformations and TGGs, respectively. eMoflon is implemented as a set of Eclipse plug-ins and has a modular architecture, involving several external components via suitable adapters, of which the (incremental) graph pattern matcher and the ILP solver are of particular interest for implementing the hybrid approach. The tool suite itself consists of two main components, i. e., IBeX and Neo. While IBeX operates on EMF models, Neo uses the graph database Neo4j both as (meta-)model storage and pattern matching engine. Both components are under active development, all sources are available on GitHub[1]. Altogether, tool developers are provided with a generic tool suite to efficiently create transformation engines for specific use cases.

Motivated by Stevens' requirement of involving the user in consistency management processes [Ste14], the VICToRy debugger was developed as an eMoflon-independent add-on component for Java-based TGG and GT tools. The main capabilities of the debugger and its value for fault-tolerant consistency management will be discussed in connection with requirement R9.

R7: Full Automation

For all consistency management operations, including CS, the respective solution is computed in a fully automated manner (cf. Fig. 5.1, 6.1, 7.2 and 8.1). Implementation-wise, Java code is generated from rules and constraints at compile time, which is executed as part of the uniform consistency management algorithm at runtime (cf. Fig. 9.5 and 9.12). The inspection of inputs and outputs differs between IBeX and Neo: While in IBeX, different EMF editors are used to handle the resources, a combination of the textual eMSL language and the visual Neo4j browser is used in Neo. During the transformation, the integration expert is informed about the current state and detected faults via console outputs.

R8: Efficiency and Scalability

A main motivation for combining TGGs and optimisation techniques is to benefit from both the scalability of algorithmic approaches and the flexibility of search-based approaches. While a certain degree of flexibility is necessary to process faulty input models, efficiency and scalability are important to use the proposed framework in practice. In order to assess the runtime behaviour of the hybrid approach, several experimental evaluations with different settings have been conducted (cf. Sect. 5.8, 6.6, 7.7, 8.5, and 9.6). In general, most operations show a satisfactory runtime behaviour for growing model sizes. Especially the CO operation could be efficiently implemented in eMoflon::Neo, as only one database query per rule is necessary to determine all application candidates. Furthermore, the other three operations presented in Chap. 5 scale reasonably well, also in combination with negative constraints, whereas room for improvement is left for implication constraints.

For several reasons, the performance of the CS operation is an open problem: First, there are multiple rule variants for each declarative TGG rule, whereby the number of variants grows exponentially with the rule size. Second, in contrast to all other operations, the optimisation step appears to be even more costly than the pattern matching step, which indicates that the constructed optimisation problem is inherently complex. As there

[1] https://github.com/eMoflon

is no separate operation for one-sided model synchronisation (cf. R5), one can assume that scalability is also problematic for this operation. Compared to existing incremental implementations [LAF+17], the differences are substantial.

R9: User Interaction

As user involvement is one of the main requirements for fault-tolerant systems, the VICToRy debugger was developed in order to make consistency management processes transparent and understandable for integration experts. There existed several MDE debuggers prior to the development of VICToRy, which indeed have a different scope: While these debuggers enable the user to understand the pattern matching and rule application process, they are not intended for large-scale transformations, as they lack a suitable breakpoint concept. VICToRy, in contrast, treats a rule application as the smallest unit for debugging model transformations. It offers a differentiated breakpoint concept that enables a target-oriented search for faults, either in the input models or in the specification of rules and constraints. As determining suitable configuration parameters for concurrent model synchronisation (R5) turns out to involve enormous efforts, a reasonable implementation of this operation requires some user interaction in any way. Due to the complexity and novelty of this operation, we decided to spend some time on developing requirements together with experts from different subfields of MDE. The resulting UI prototype and process specification shall be integrated into VICToRy in the future to equip the integration expert with tool support for conflict detection and resolution.

R10: Applicability

In Chap. 11 and 12, we presented two comprehensive industrial case studies, in which real-world problems were solved by using different operations of the hybrid approach. At DB Netz AG, an infrastructure manager of the German railway system, a BX between the semi-formal language SysML and the formal language Event-B was automated to reduce human efforts and remove one potential source of error from the transformation process. While fault-tolerance was no explicit requirement, the future use of other operations of the hybrid framework (e. g., consistency checks) appears to be promising in the application context. The CC operation was used to create optimal test schedules at dSPACE GmbH, a software and hardware developer for testing mechatronic control units. For this use case, fault-tolerance (in a technical sense) is essential, because it is hardly possible and sometimes even undesired to utilise all availabilities of each software tester during the entire testing phase. The case study also involves a quantitative evaluation, whereas for the Adosate TGG (Sect. 10.7) and the previously mentioned use case at DB Netz AG, the conducted experiments are purely qualitative. It became evident that, depending on the TGG at hand and the case-specific requirements, further performance improvements can be necessary, while it was at the same time possible to adapt the tool support as required.

The overall feedback given by the practitioners at DB Netz and dSPACE was largely positive. They appreciate the functionality of the eMoflon tool suite, in particular that different consistency management tasks can be accomplished in a uniform manner. Therefore, the main argument for BX approaches, i. e., that different operations can be automatically derived based on a common consistency relation, is confirmed by practitioners from different domains. Suggestions for improvement are particularly made with respect to the usability and understandability of the approach. It is possible for inexperienced users to understand TGG rules that have been specified by experts, and they seem to be a suitable basis for discussing the intended system behaviour. Without further human or

tool support, it is hard for domain and integration experts to specify rules and constraints, and thereby control consistency management processes, though.

Summary

After having discussed how the contributions of this thesis address the (technology-independent) requirements, let us briefly shift the focus to ongoing research on the TGG formalism. In their roadmap for future research on TGGs, Anjorin et al. [ALS15] name five dimensions, according to which the contributions of this thesis shall be classified (cf. Fig. 13.1).

Two steps forward have been taken with respect to *(fault-)tolerance*, as the hybrid framework is able to handle both inconsistent input models and inconsistent user edits. An open question might be whether it is even beneficial to allow inconsistent output models in some cases, which is prevented completely by the developed framework.

By integrating implication constraints, the *expressiveness* of supported TGGs has been improved, comparable to the expressive power of positive application conditions. There are still (complex) constraint types that cannot be expressed yet. Also, it must be noted that multi-amalgamation (cf. Sect. 4.6), being formalised and implemented for TGGs in prior work [LAST15, LAS15], is not yet supported by the hybrid framework.

As a further consistency management operation, concurrent model synchronisation (denoted as *integration* in the diagram) was introduced by Orejas et al. [OPN20], Fritsche et al. [FKM+20], and proposed in a fault-tolerant version in Chap. 7 and 8 of this thesis. Model transformation across *multiple domains* as a last step on the concurrency axis is an active research field, that is elaborated in Sect. 13.2.

Further improvements with respect to scalability or reliability have not taken place yet and – in addition to the aspects discussed in the following section – indicate a direction for further research.

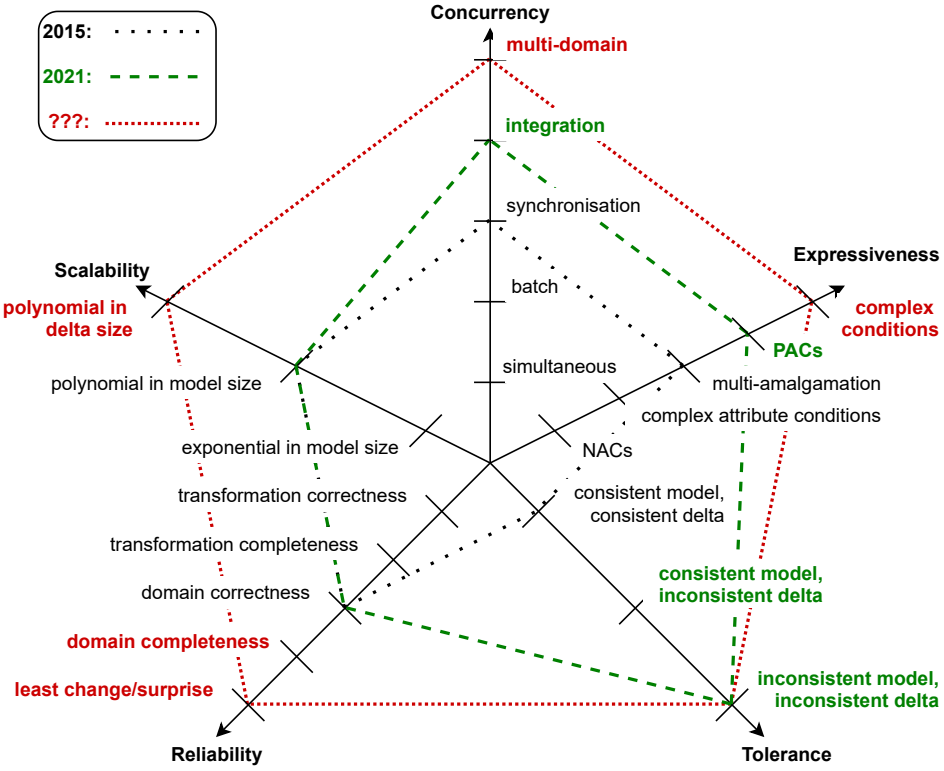


Figure 13.1: TGG-specific contributions

13.2 Future Work

After the previous Sect. 13.1 has given a summary of how the contributions of this thesis address the requirements for fault-tolerant consistency management systems, the remainder of this chapter is dedicated to possible directions for future research in the narrower and broader sense.

Extending the Hybrid Approach

With the consistency management operations of the hybrid approach, output models – which consist of a largest consistent sub-triple and the remaining elements of the input models – can be computed from inconsistent inputs. These sub-triples are indeed helpful for input models with faults at the end of the rule application sequence, e. g., models with incorrectly typed or disconnected leave nodes. As the largest consistent sub-triple must be the result of a correct rule application sequence, it is highly problematic if, in extreme cases, no matches for axiom rules can be determined. Consequently, the largest consistent sub-triple would be empty, which amounts to rejecting the input models. Instead, it would be helpful in such situations if a "smallest consistent super-triple" could be established that adds as few elements to the input models as possible. Alternatively, fuzzy mechanisms such as "island grammars" from the field of compiler construction could be used [ABvdB+12, EHSB13] in order to determine matches even though single context elements are missing.

Similarly, the handling of typing faults leaves room for improvement: In order to find a rule application candidate, a valid match must exist that maps each context element of the rule to the host graph. This means that a single typing fault, i. e., an incorrectly typed node or edge, can prevent the consistency management system from finding a match at all, resulting in rather useless solutions as described above. Additional mechanisms that tolerate typing faults could therefore improve the average solution quality of the hybrid approach.

Furthermore, the effects of the distribution of faults over the input models, i. e., the differences between an approximately equal distribution and a concentration of faults on a specific part of the input, are not known yet. This is because the search for an optimal solution is always global and maximises the total number of translated elements, independent of their location within the input models. It would be interesting to investigate the strengths and weaknesses of the approach, depending on the distribution of faults, especially from the user perspective.

Increasing Expressiveness

As already suggested in Sect. 13.1, the expressive power of the hybrid approach can be increased by adding further language features to the framework. For the running example, it was necessary to find a workaround for the variable initialisation which could be expressed by the multi-rule of the multi-amalgamated rule *PseudostateToActions* (cf. Fig. 4.13). The integration of this language feature seems promising and feasible: The substructure of one kernel rule and arbitrarily many multi-rules resembles the structure of a graph constraint with one premise pattern and arbitrarily many conclusion patterns.

This aspect leads over to open issues regarding constraint handling. First, the scalability experiments indicate that further performance improvements are necessary to efficiently handle implication constraints. Second, advanced constraint types could be a valuable extension of the hybrid approach, such as complex graph conditions [GEH11] or OCL constraints as an important standard of the OMG.

Improving Concurrent Synchronisation

Regarding the integration of the concurrent synchronisation into the hybrid framework, several points for future improvements can be identified. The most obvious problem is the concrete parametrisation of the objective function for this operation. In Chap. 8, we have seen that enormous efforts are required to determine suitable parameters for a small example. Furthermore, there can be situations in which changes in either source or target domain are preferred independent of the parameter values (cf. Sect. 7.6), which makes the need for separate parameters for source and target domain apparent.

According to the quantitative evaluation of Sect. 8.5, ILP outperforms all single-objective optimisation heuristics for the concrete application scenario due to the tiny portion of feasible solutions in the search space. This can be - at least partly - led back to the uniform penalty that is assigned to invalid solutions (cf. Sect. 8.4). The severity of constraint violations could be taken into account, e. g., by imposing a penalty for each violation, in order to distinguish between nearly correct and completely invalid solutions.

Another important point is the grouping of user edits into atomic units. In its current version, the concurrent synchronisation operation considers each creation or deletion of an element as a separate change, regardless of the user's intention behind it. Let us assume that a user wants to introduce a superclass `S` for classes `A` and `B`, because `A` and `B` share several attributes and methods. It would be possible for the CS operation, though, to introduce `S` but leave all common attributes and methods in `A` and `B`, which is worse than rejecting the change completely. The *atomicity* of user edits can be integrated by adding further constraints to the definition of the optimisation problem.

In contrast to the approach of Fritsche et al. [FKM+20], there is no support for explicitly handling attribute value changes yet. Currently, the operation can simulate these changes by deleting and adding nodes with the respective values, which can easily lead to rejecting the change in cases the new value must be propagated to other nodes of the same model.

Enhancing the Tool Support

The two components of the eMoflon tool suite fundamentally differ in their underlying technology, i. e., the EMF framework and the graph database Neo4j. While the use of graph databases makes metamodelling more flexible and makes it unnecessary to entirely load large models into the main memory, the EMF framework is still a de-facto standard for many modelling tools. The use of Ecore-compliant XMI files facilitated the implementation of the transformation tool chain at DB Netz by far (cf. Sect. 11.5), which was, besides the broader support for attribute conditions, the main argument for preferring IBeX over Neo for this use case. To improve the interoperability with EMF-based tools, import and export functionality for XMI files is currently added to eMoflon::Neo.

With the development of the VICToRy debugger, a remarkable step towards explainable BX was taken. According to the conducted expert interviews, fault detection for experts and exploration of a TGG for novices are promising use cases for the debugger. While the overall functionality received positive feedback, the participants stated that the question why a particular rule is *not* applicable in a specific situation cannot be answered with the tool. Likewise, practitioners at DB Netz and dSPACE stated that further tool support is required to enable integration experts to define TGG rules without assistance. They suggested to develop a domain-specific UI that guides inexperienced users through consistency management processes, which would increase the users' trust in the software system.

As discussed in the last subsection, the automated conflict resolution via the CS operation leaves room for improvement from several points of view. An alternative and presumably more promising line of research would be to shift the task of resolving conflicts to the integration expert. In this thesis, a UI prototype and process specification for an interactive concurrent synchronisation component were developed. The actual implementation is one of the next steps to make the debugger ready for practical use.

Further Case Studies

The practical applicability of the hybrid framework is demonstrated via two industrial case studies in Chap. 11 and 12. Fault-tolerance was only an implicit requirement in the latter case, though, which motivates us to conduct further studies for which fault-tolerant mechanisms are explicitly required. As eMoflon::IBeX was used in both cases, further studies based on eMoflon::Neo are especially important. Furthermore, the use of multiple consistency management operations in the same industrial context would underpin the unique selling proposition of BX approaches. In both of the existing case studies, the CS operation could be used to maintain consistency over a longer period of time.

Due to time and capacity constraints, the empirical studies which were conducted in the course of this thesis are of purely qualitative nature. Structured experiments with a sufficiently high number of test users are planned for the future. Furthermore, only different student groups, one key user each at dSPACE and DB Netz, and the eMoflon development team have hands-on experience with the tool support, whereas the expert interviews were conducted based on live demonstrations. To strengthen the acceptance of MDE approaches in industry, more practitioners must be directly involved into comparable experiments. Finally, a direct comparison of fault-tolerant and -intolerant approaches is necessary to assess how valuable the gained flexibility is perceived by domain and integration experts.

Follow-up Work: Leaving the Hybrid Approach

The topic of fault-tolerant consistency management in MDE is broadly diversified and has a larger scope than the proposed approach of this thesis can address. Only a small portion of the relevant papers that have been identified by the Systematic Literature Review (SLR) (cf. Chap. 2) can be classified as BX approaches. For instance, fault-tolerance plays an important role for model-to-text transformations (e. g., code generation) as well. Opportunities and risks of tolerating inconsistencies in this area need to be investigated in future work. While the hybrid approach is based on graph pattern matching and graph transformation rules, other concepts of fault-tolerance might be more suitable for textual transformation languages such as ATL.

Another problem that is not directly addressed in the scope of this thesis is fault-tolerance with respect to metamodel-conformance. The SLR has shown that software developers hesitate to apply MDE approaches because changing the metamodel at later stages of the development process involves substantial manual efforts as most tools enforce strict metamodel-conformance (cf. Sect. 2.6).

With help of the SLR, many commonalities and differences between the closely related concepts of fault-tolerance and uncertainty could be identified. A question that remains open is, however, which of the two concepts is more recommendable in which application scenarios, or whether it is even possible to combine them to utilise the strengths of both concepts. In total, flexible modelling is a practically relevant and complex topic, for which more research is needed.

Follow-up Work: Multi-directional Transformations

The last step on the concurrency axis of Fig. 13.1, i. e., transformation across *multiple domains*, is a topic that recently gained noticeable attention among MDE researchers. The motivation for extending the bidirectional to a multi-directional case is driven by practical considerations: Usually, more than two (teams of) domain experts work in parallel on larger software projects, such that the consistency relation is not binary, but n-ary, requiring respective consistency management frameworks. Building upon seminal work [KKS07, Ste17] and the Dagstuhl seminar on multi-directional transformations [CKSZ18], several approaches to maintaining consistency in software systems with arbitrarily many models have been recently proposed.

Trollmann et al. generalise TGGs to *graph diagrams* [TA16] and define semantics-preserving transformations for such graph diagrams, consisting of two or more graphs, which can be pair-wise connected by further correspondence graphs. Stünkel et al. [SKLR20, SKLR21] further generalise this idea by introducing *comprehensive systems*, in which the set of correspondence graphs is replaced by a single commonality structure that defines corresponding elements in more than two models, and thereby represents an n-ary consistency relation. It is shown that important formal properties for graph transformations also hold for comprehensive systems. Conceptual work on model transformation networks is proposed by Klare et al. [Kla21, SK21, GKB21], who define correctness properties (e. g., compatibility and a suitable orchestration) and quality properties for transformation networks.

While TGGs, graph diagrams and comprehensive systems share common ground with respect to their formalisation, there is still a substantial amount of work left regarding the operationalisation of rules for the last two concepts. Nonetheless, introducing fault-tolerance to networks of model transformations is a problem of practical relevance, for which it makes sense to develop concepts at this point already.

Follow-up Work: Low-Code Development

Another topic of interest that emerged in recent years and that is closely related to MDE is software development with *Low-Code Development Platforms (LCDPs)*. The idea behind low-code development is very similar to MDE concepts: With the help of a mix of visual and textual editors and configuration menus, users shall be enabled to develop simple software applications for different platforms, including smart phones, tablets and full desktop PCs. Besides rapidly growing interest in industry, the establishment of a workshop for low-code development co-located with the MODELS conference in 2020 [GI20] underpins the relevance for the MDE community.

First meta-studies come to the conclusion that low-code development is rather a combination of existing lines of research, such as rapid application development, MDE, or software as a service, than a fundamentally novel software development approach. As an opportunity for further research, Bock and Frank name the synchronisation of models and code [BF21], which requires BX mechanisms also for LCDPs. With all platforms we are aware of, only unidirectional code generation is possible, though. They further see great potential in introducing domain-specific abstractions to LCDPs, whereas users of current platforms are obliged to create generic data models for their software application.

Also in this context, fault-tolerance plays an important role for the software development process. LCDPs are intended to be used by so-called “citizen developers”, who can be considered as the counterparts to domain experts in MDE: Employees without (advanced) programming experience but profound domain knowledge shall be enabled to develop

simple software applications on their own. As these employees are not familiar with IDEs and compiler messages, it is particularly important to make the development process as flexible as possible. Instead of enforcing perfect consistency, LCDPs should present a partial solution to citizen developers, which can be iteratively completed to a fully functional low-code application.

Bibliography

[ABC+20] Arturo Amendola, Anna Becchi, Roberto Cavada, Alessandro Cimatti, Alberto Griggio, Giuseppe Scaglione, Angelo Susi, Alberto Tacchella, and Matteo Tessi. A Model-Based Approach to the Design, Verification and Deployment of Railway Interlocking System. In *International Symposium on Leveraging Applications of Formal Methods (ISoLA) 2020, Proceedings, Part III*, pages 240–254. Springer, 2020. Cited on page 201.

[ABGM09] Aldeida Aleti, Stefan Björnander, Lars Grunske, and Indika Meedeniya. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES) 2009, Proceedings*, pages 61–71. IEEE, 2009. Cited on page 218.

[ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010. Cited on page 204.

[ABH+10] Jean-Raymond Abrial, Michael J Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010. Cited on pages 205 and 206.

[ABvdB+12] Ali Afroozeh, Jean-Christophe Bach, Mark van den Brand, Adrian Johnstone, Maarten Manders, Pierre-Etienne Moreau, and Elizabeth Scott. Island Grammar-Based Parsing Using GLL and Tom. In *Software Language Engineering (SLE) 2012, Proceedings*, pages 224–243. Springer, 2012. Cited on page 240.

[ABW17] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. The Families to Persons Case. In *Transformation Tool Contest (TTC) 2017, Proceedings*, pages 27–34. CEUR-WS.org, 2017. Cited on pages 97, 98, 160, 197, and 283.

[ABW+20] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling*, 19(3):647–691, sep 2020. Cited on pages 54, 97, 104, and 203.

[AC19] Anthony Anjorin and James Cheney. Provenance Meets Bidirectional Transformations. In *International Workshop on Theory and Practice of Provenance (TaPP) 2019, Proceedings*. USENIX Association, 2019. Cited on page 199.

[ACG+14] Anthony Anjorin, Alcino Cunha, Holger Giese, Frank Hermann, Arend Rensink, and Andy Schürr. BenchmarX. In *International Workshop on*

Bidirectional Transformations (Bx) 2014, Proceedings, pages 82–86. CEUR-WS.org, 2014. Cited on page 54.

[ADAA17] Lujain Al-Dakheel and Issam Al-Azzoni. Model-to-Model based Approach for Software Component Allocation in Embedded Systems. In *International Conference on Model-Driven Engineering and Software Development (MODELSWARD) 2017, Proceedings*, pages 320–328. SciTePress, 2017. Cited on page 218.

[ADJ+17] Anthony Anjorin, Zinovy Diskin, Frédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. Benchmarx Reloaded: A Practical Benchmark Framework for Bidirectional Transformations. In *International Workshop on Bidirectional Transformations (Bx) 2017, Proceedings*, pages 15–30. CEUR-WS.org, 2017. Cited on page 54.

[AGF15] Colin Atkinson, Ralph Gerbig, and Mathias Fritzsche. A multi-level approach to modeling language extension in the Enterprise Systems Domain. *Information Systems*, 54, 2015. Cited on page 33.

[AH07] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007. Cited on pages 4, 41, and 202.

[AH18] Abdullah Alqahtani and Reiko Heckel. Model Based Development of Data Integration in Graph Databases Using Triple Graph Grammars. In *Software Technologies: Applications and Foundations (STAF) 2018, Workshop Proceedings*, pages 399–414. Springer, 2018. Cited on page 166.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA) 2006, Proceedings*, pages 361–375. Springer, 2006. Cited on page 164.

[ALK+15] Anthony Anjorin, Erhan Leblebici, Roland Kluge, Andy Schürr, and Perdita Stevens. A Systematic Approach and Guidelines to Developing a Triple Graph Grammar. In *International Workshop on Bidirectional Transformations (Bx) 2015, Proceedings*, pages 81–95. CEUR-WS.org, 2015. Cited on pages 54 and 101.

[ALPS11] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *Jahrestagung der Gesellschaft für Informatik (INFORMATIK) 2011, Abstract Proceedings*, page 281. GI, 2011. Cited on page 164.

[ALS15] Anthony Anjorin, Erhan Leblebici, and Andy Schürr. 20 Years of Triple Graph Grammars: A Roadmap for Future Research. *Electronic Communication of the European Association of Software Science and Technology*, 73(1):1–20, 2015. Cited on pages 13, 44, 55, 81, 83, and 239.

[AM01] Javier Alcaraz and Concepción Maroto. A Robust Genetic Algorithm for Resource Allocation in Project Scheduling. *Annals of Operations Research*, 102(1-4):83–109, 2001. Cited on page 218.

[AN16] Afef Awadid and Selmin Nurcan. A Systematic Literature Review of Consistency Among Business Process Models. In *Workshop on Business Process Modelling, Development, and Support (BPMDS) 2016, Proceedings*, volume 248, pages 175–195. Springer, 2016. Cited on page 19.

[AN19] Afef Awadid and Selmin Nurcan. Consistency requirements in business process modeling: a thorough overview. *Software and Systems Modeling*, 18(2):1097–1115, 2019. Cited on page 19.

[Aqu09] Nathalie Aquino. Adding flexibility in the model-driven engineering of user interfaces. In *Symposium on Engineering Interactive Computing Systems (EICS) 2009, Proceedings*, pages 329–332. ACM, 2009. Cited on page 33.

[ASCG+16] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Introduction to Bidirectional Transformations. In *Bidirectional Transformations*, pages 1–28. Springer, 2016. Cited on page 4.

[ASLS14] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In *Fundamental Approaches to Software Engineering (FASE) 2014, Proceedings*, pages 340–354. Springer, 2014. Cited on pages 54, 195, and 284.

[AST12] Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Construction of Integrity Preserving Triple Graph Grammars. In *International Conference on Graph Transformation (ICGT) 2012, Proceedings*, pages 356–370. Springer, 2012. Cited on pages 67 and 103.

[AVLH17] Wesley K G Assunção, Silvia R Vergilio, and Roberto E Lopez-Herrejon. Discovering Software Architectures with Search-Based Merge of UML Model Variants. In *International Conference on Software Reuse (ICSR) 2017, Proceedings*, pages 95–111. Springer, 2017. Cited on page 146.

[AVS12] Anthony Anjorin, Gergely Varró, and Andy Schürr. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. In *International Workshop on Bidirectional Transformations (Bx) 2012, Proceedings*, pages 1–16. EASST, 2012. Cited on pages 53, 54, and 59.

[AVS+14] Hani Abdeen, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-Objective Optimization in Rule-based Design Space Exploration. In *International Conference on Automated Software Engineering (ASE) 2014, Proceedings*, pages 289–300. ACM, 2014. Cited on page 146.

[AWO+20] Anthony Anjorin, Nils Weidmann, Robin Oppermann, Lars Fritsche, and Andy Schürr. Automating test schedule generation with domain-specific languages: a configurable, model-driven approach. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2020, Proceedings*, pages 320–331. ACM, 2020. Cited on pages 16 and 228.

[AY15] Shaukat Ali and Tao Yue. Evolving, Modelling and Testing Realistic Uncertain Behaviours of Cyber-Physical Systems. In *International Conference on Software Testing, Verification, and Validation (ICST) 2015, Proceedings*. IEEE, 2015. Cited on page 32.

[AYK+14] Takahiro Ando, Hirokazu Yatsu, Weiqiang Kong, Kenji Hisazumi, and Akira Fukuda. Translation rules of SysML state machine diagrams into CSP# toward formal model checking. *International Journal of Web Information Systems*, 10(2):151–169, 2014. Cited on page 204.

[AYL+18] Anthony Anjorin, Enes Yigitbas, Erhan Leblebici, Andy Schürr, Marius Lauder, and Martin Witte. Description Languages for Consistency Management Scenarios Based on Examples from the Industry Automation Domain. *The Art, Science, and Engineering of Programming*, 2(3):7, 2018. Cited on page 4.

[Bal91] Robert Balzer. Tolerating Inconsistency. In *International Conference on Software Engineering (ICSE) 1991, Proceedings*, pages 158–165. IEEE, 1991. Cited on pages 7, 18, 23, 25, 28, and 31.

[BCC+15] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting efficient and advanced omniscient debugging for xDSMLs. In *Software Language Engineering (SLE) 2015, Proceedings*, pages 137–148. ACM, 2015. Cited on page 184.

[BCC+16] Alessio Bucaioni, Antonio Cicchetti, Federico Ciccozzi, Saad Mubeen, Alfonso Pierantonio, and Mikael Sjödin:. Handling Uncertainty in Automatically Generated Implementation Models in the Automotive Domain. In *EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA) 2016, Proceedings*, pages 173–180. IEEE, 2016. Cited on pages 25 and 33.

[BEP+17] Marco Brambilla, Romina Eramo, Alfonso Pierantonio, Gianni Rosa, and Eric Umuhoza. Enhancing Flexibility in User Interaction Modeling by Adding Design Uncertainty to IFML. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Proceedings*, pages 435–440. IEEE, 2017. Cited on pages 25, 29, and 33.

[BET12] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation. *Software and System Modeling*, 11(2):227–250, 2012. Cited on page 165.

[Béz05] Jean Bézivin. On the Unification Power of Models. *Software and Systems Modeling*, 4(2):171–188, 2005. Cited on page 3.

[BF21] Alexander Bock and Ulrich Frank. In Search of the Essence of Low-Code: An Exploratory Study of Seven Development Platforms. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2021, Companion Proceedings*. IEEE, 2021. Cited on page 243.

[BFH85] Paul Boehm, Harald-Reto Fonio, and Annegret Habel. Amalgamation of Graph Transformations with Applications to Synchronization. In *Theory and Practice of Software Development (TAPSOFT) 1985, Proceedings*, pages 267–283. Springer, 1985. Cited on page 69.

[BFH87] Paul Boehm, Harald-Reto Fonio, and Annegret Habel. Amalgamation of graph transformations: A synchronization mechanism. *Journal of Computer and System Sciences*, 34(2-3):377–408, 1987. Cited on page 54.

[bFKLW12] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. Search-based detection of high-level model changes. In *International Conference on Software Maintenance (ICSM) 2012, Proceedings*, pages 212–221. IEEE, 2012. Cited on page 146.

[BFP+08] Aaron Bohannon, John Nathan Foster, Benjamin C Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang : Resourceful Lenses for String Data. In *Symposium on Principles of Programming Languages (POPL) 2008, Proceedings*, pages 407–419. ACM, 2008. Cited on pages 81 and 83.

[BFT+19] Robert Bill, Martin Fleck, Javier Troya, Tanja Mayerhofer, and Manuel Wimmer. A local and global tour on MOMoT. *Software and Systems Modeling*, 18(2):1017–1046, 2019. Cited on pages 146, 147, 151, 153, and 158.

[BG07] Ebrahim Bagheri and Ali A Ghorbani. On the Collaborative Development of Para-Consistent Conceptual Models. In *International Conference on Quality Software (QSIC) 2007, Proceedings*, pages 336–341. IEEE, 2007. Cited on page 25.

[BG16] Thomas Buchmann and Sandra Greiner. Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code. In *International Joint Conference on Software Technologies (ICSOFT) 2016, Proceedings (Vol. 2)*, pages 27–38. SciTePress, 2016. Cited on page 125.

[BGN+04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool Integration at the Meta-model Level: The Fujaba Approach. *International Journal on Software Tools for Technology Transfer*, 6(3):203–218, 2004. Cited on pages 164 and 165.

[BGR+17] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. CASE Tool Support for Variability Management in Software Product Lines. *ACM Computing Surveys*, 50(1):14:1–14:45, 2017. Cited on page 19.

[BH07] Michael J Butler and Stefan Hallerstede. The Rodin formal modelling tool. In *Christmas Workshop: Formal Methods in Industry (FMI) 2007, Proceedings*, pages 1–5. eWiC, 2007. Cited on page 202.

[BHS05] Leopoldo Bertossi, Anthony Hunter, and Torsten Schaub. Introduction to Inconsistency Tolerance. In *Inconsistency Tolerance [result from a Dagstuhl seminar]*, pages 1–14. Springer, 2005. Cited on page 8.

[BLC18] Oluwaseun Bamgboye, Xiaodong Liu, and Peter Cruickshank. Towards Modelling and Reasoning About Uncertain Data of Sensor Measurements for Decision Support in Smart Spaces. In *International Computer Software and Applications Conference (COMPSAC) 2018, Proceedings*, pages 744–749. IEEE, 2018. Cited on pages 26 and 32.

[BLL+07] Christopher Brooks, Edward A Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains). Technical report, University of California, Berkeley, CA, 2007. Cited on page 217.

[BM14] Jorge Barreiros and Ana Moreira. Flexible Modeling and Product Derivation in Software Product Lines. In *International Conference on Software Engineering and Knowledge Engineering (SEKE) 2013, Proceedings*, pages 67–70. Knowledge Systems Institute Graduate School, 2014. Cited on pages 24 and 33.

[BMMM08] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *International Conference on Software Engineering (ICSE) 2008, Proceedings*, pages 511–520. ACM, 2008. Cited on pages 23, 24, and 32.

[BPD+14] Dominique Blouin, Alain Plantec, Pierre Dissaux, Frank Singhoff, and Jean-Philippe Diguet. Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report. In *International Conference on Model Transformation (ICMT) 2014, Proceedings*, pages 106–121. Springer, 2014. Cited on pages 4 and 195.

[Bro18] Manfred Broy. Yesterday, Today, and Tomorrow: 50 Years of Software Engineering. *IEEE Software*, 35(5):38–43, 2018. Cited on page 3.

[BRST05] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model Transformations in Practice Workshop. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2005, Workshop Proceedings*. Springer, 2005. Cited on page 96.

[BSE10] Nils Bandener, Christian Soltenborn, and Gregor Engels. Extending DMM Behavior Specifications for Visual Execution and Debugging. In *Software Language Engineering (SLE) 2010, Proceedings*, pages 357–376. Springer, 2010. Cited on page 185.

[BSV20] Aren A Babikian, Oszkár Semeráth, and Dániel Varró. Automated Generation of Consistent Graph Models with First-Order Logic Theorem Provers. In *Fundamental Approaches to Software Engineering (FASE) 2020, Proceedings*, pages 441–461. Springer, 2020. Cited on pages 23, 24, and 103.

[Buc18] Thomas Buchmann. BXtend - A Framework for (Bidirectional) Incremental Model Transformations. In *International Conference on Model-Driven Engineering and Software Development (MODELSWARD) 2018, Proceedings*, pages 336–345. SciTePress, 2018. Cited on page 98.

[BvBG+10] Twan Basten, Emiel van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian de Smet, Lou J Somers, and Egbert Teeselink. Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In *International Symposium on Leveraging Applications (ISoLA) 2010, Proceedings, Part I*, pages 90–105. Springer, 2010. Cited on page 218.

[BYWE21] Kai Biermeier, Enes Yigitbas, Nils Weidmann, and Gregor Engels. Ensuring User Interface Adaptation Consistency through Triple Graph Grammers. In *International Workshop on Human-Centered Software Engineering for Changing Contexts of Use (HCSE) 2021, Proceedings (to appear)*, 2021. Cited on page 16.

[BZJ19] Alexandru Burdusel, Steffen Zschaler, and Stefan John. Automatic Generation of Atomic Consistency Preserving Search Operators for Search-Based Model Engineering. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2019, Proceedings*, pages 106–116. IEEE, 2019. Cited on pages 23 and 24.

[CBGS18] Matteo Camilli, Carlo Bellettini, Angelo Gargantini, and Patrizia Scandurra. Online Model-Based Testing under Uncertainty. In *International Symposium on Software Reliability Engineering (ISSRE) 2018, Proceedings*, pages 36–46. IEEE, 2018. Cited on pages 26 and 32.

[CCGdL10] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010. Cited on page 102.

[CdM19] Jarbele C S Coutinho, Wilkerson de L. Andrade, and Patricia D L Machado. Requirements Engineering and Software Testing in Agile Methodologies: a Systematic Mapping. In *Brazilian Symposium on Software Engineering (SBES) 2019, Proceedings*, pages 322–331. ACM, 2019. Cited on page 19.

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Cambridge University Press, 2012. Cited on page 62.

[CESG17] Jonathan Corley, Brian P Eddy, Eugene Syriani, and Jeff Gray. Efficient and scalable omniscient debugging for model transformations. *Software Quality Journal*, 25(1):7–48, 2017. Cited on page 184.

[CFH+09] Krzysztof Czarnecki, John Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *International Conference on Model Transformation (ICMT) 2009, Proceedings*, pages 260–283. Springer, 2009. Cited on pages 44 and 54.

[CGdL+17] Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara, Robert Clarisó, and Jordi Cabot. Translating Target to Source Constraints in Model-to-Model Transformations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Proceedings*, pages 12–22. IEEE, 2017. Cited on page 102.

[CGMS15] James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Towards a Principle of Least Surprise for Bidirectional Transformations. In *International Workshop on Bidirectional Transformations (Bx) 2015, Proceedings*, pages 66–80. CEUR-WS.org, 2015. Cited on page 8.

[CGSB17] Matteo Camilli, Angelo Gargantini, Patrizia Scandurra, and Carlo Bellettini. Towards Inverse Uncertainty Quantification in Software Development (Short Paper). In *International Conference on Software Engineering and Formal Methods (SEFM) 2017, Proceedings*, pages 375–381. Springer, 2017. Cited on pages 26 and 32.

[CK13] Glenn Callow and Roy Kalawsky. A Satisficing Bi-Directional Model Transformation Engine using Mixed Integer Linear Programming. *Journal of Object Technology*, 12(1):1: 1–43, 2013. Cited on pages 24, 33, and 83.

[CKSZ18] Anthony Cleve, Ekkart Kindler, Perdita Stevens, and Vadim Zaytsev. Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491). *Dagstuhl Reports*, 8(12):1–48, 2018. Cited on page 243.

[CLFLW16] Georgiana Caltais, Florian Leitner-Fischer, Stefan Leue, and Jannis Weiser. SysML to NuSMV Model Transformation via Object-Orientation. In *International Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy) 2016, Proceedings*, pages 31–45. Springer, 2016. Cited on page 204.

[CLS20] Georgiana Caltais, Stefan Leue, and Hargurbir Singh. Correctness of an ATL Model Transformation from SysML State Machine Diagrams to Promela. In *International Conference on Model-Driven Engineering and Software Development (MODELSWARD) 2020, Proceedings*, pages 360–372. SciTePress, 2020. Cited on page 204.

[CLvV07] Carlos Artemio Coello Coello, Gary B Lamont, and David A van Veldhuizen. *Evolutionary algorithms for solving multi-objective problems, Second Edition*. Springer, 2007. Cited on page 149.

[Cou90] Bruno Courcelle. The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs. *Information and Computation*, 85(1):12–75, 1990. Cited on page 62.

[CR20] Matteo Camilli and Barbara Russo. Model-Based Testing Under Parametric Variability of Uncertain Beliefs. In *International Conference on Software Engineering and Formal Methods (SEFM) 2020, Proceedings*, pages 175–192. Springer, 2020. Cited on pages 30 and 32.

[CRE+10] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, Alfonso Pierantonio, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL : A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering (SLE) 2010, Proceedings*, pages 183–202. Springer, 2010. Cited on pages 23, 81, 82, and 125.

[CSBW09] Betty H C Cheng, Peter Sawyer, Nelly Bencomo, and Jon Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2009, Proceedings*, pages 468–483. Springer, 2009. Cited on pages 26, 32, and 33.

[CST12] Selim Ciraci, Hasan Sözer, and Bedir Tekinerdogan. An Approach for Detecting Inconsistencies between Behavioral Models of the Software Architecture and the Code. In *International Computer Software and Applications Conference (COMPSAC) 2012, Proceedings*, pages 257–266. IEEE, 2012. Cited on page 24.

[DAPM02] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. Cited on page 146.

[DC16] Byron DeVries and Betty H C Cheng. Automatic detection of incomplete requirements via symbolic analysis. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2016, Proceedings*, pages 385–395. ACM, 2016. Cited on page 32.

[DC17] Byron DeVries and Betty H C Cheng. Using Models at Run Time to Detect Incomplete and Inconsistent Requirements. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings*, pages 201–209. CEUR-WS.org, 2017. Cited on page 32.

[DDG+13] Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. metroII: A design environment for cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 12(1s):49:1–49:31, 2013. Cited on page 217.

[DDGV16] István Dávid, Joachim Denil, Klaas Gadeyne, and Hans Vangheluwe. Engineering Process Transformation to Manage (In)consistency. In *International Workshop on Collaborative Modelling in MDE (COMMitMDE) 2016, Proceedings*, pages 7–16. CEUR-WS.org, 2016. Cited on page 146.

[Dec11] Hendrik Decker. Data Quality Maintenance by Integrity-Preserving Repairs that Tolerate Inconsistency. In *International Conference on Quality Software (QSIC) 2011, Proceedings*, pages 192–197. IEEE, 2011. Cited on pages 23, 25, 31, and 32.

[Dec17] Hendrik Decker. Inconsistency-Tolerant Database Repairs and Simplified Repair Checking by Measure-Based Integrity Checking. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 34:153–183, 2017. Cited on pages 8 and 18.

[DEW+16] Hoa Khanh Dam, Alexander Egyed, Michael Winikoff, Alexander Reder, and Roberto E Lopez-Herrejon. Consistent merging of model versions. *Journal of Systems and Software*, 112:137–155, 2016. Cited on page 146.

[DG08] Duc-Hanh Dang and Martin Gogolla. On Integrating OCL and Triple Graph Grammars. In *Models in Software Engineering (MiSE) 2008, Proceedings*, pages 124–137. Springer, 2008. Cited on page 166.

[Dij72] Edsger W Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. Cited on page 3.

[DJSC17] Gwendal Daniel, Frédéric Jouault, Gerson Sunyé, and Jordi Cabot. Gremlin-ATL: a scalable model transformation framework. In *International Conference on Automated Software Engineering (ASE) 2017, Proceedings*, pages 462–472. IEEE, 2017. Cited on pages 166 and 173.

[DJVV14] Joachim Denil, Maris Jukss, Clark Verbrugge, and Hans Vangheluwe. Search-Based Model Optimization Using Model Transformations. In *System Analysis and Modeling (SAM) 2014, Proceedings*, pages 80–95. Springer, 2014. Cited on pages 81 and 83.

[DKEM16] Andreas Demuth, Roland Kretschmer, Alexander Egyed, and Davy Maes. Introducing Traceability and Consistency Checking for Change Impact Analysis across Engineering Tools in an Automation Solution Company: An Experience Report. In *International Conference on Software Maintenance and*

Evolution (ICSME) 2016, Proceedings, pages 529–538. IEEE, 2016. Cited on page 24.

[DM08] Hendrik Decker and Davide Martinenghi. Classifying integrity checking methods with regard to inconsistency tolerance. In *International Conference on Principles and Practice of Declarative Programming (PPDP) 2008, Proceedings*, pages 195–204. ACM, 2008. Cited on pages 25, 31, and 32.

[DM11] Hendrik Decker and Davide Martinenghi. Inconsistency-Tolerant Integrity Checking. *IEEE Transactions on Knowledge and Data Engineering*, 23(2):218–234, 2011. Cited on pages 8 and 18.

[DMV+17] István Dávid, Bart Meyers, Ken Vanherpen, Yentl Van Tendeloo, Kristof Berx, and Hans Vangheluwe. Modeling and Enactment Support for Early Detection of Inconsistencies in Engineering Processes. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings*, pages 145–154. CEUR-WS.org, 2017. Cited on pages 23 and 24.

[DRE14] Hoa Khanh Dam, Alexander Reder, and Alexander Egyed. Inconsistency Resolution in Merging Versions of Architectural Models. In *Working Conference on Software Architecture, (WICSA) 2014, Proceedings*, pages 153–162. IEEE, 2014. Cited on page 146.

[DRV+16] Csaba Debreceni, István Ráth, Dániel Varró, Xabier De Carlos, Xabier Mendialdua, and Salvador Trujillo. Automated Model Merge by Design Space Exploration. In *Fundamental Approaches to Software Engineering (FASE) 2016, Proceedings*, pages 104–121. Springer, 2016. Cited on page 146.

[DSB+16] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. NeoEMF: A Multi-database Model Persistence Framework for Very Large Models. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2016, Companion Proceedings*, pages 1–7. CEUR-WS.org, 2016. Cited on page 166.

[DSB+17] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. NeoEMF: A multi-database model persistence framework for very large models. *Science of Computer Programming*, 149:9–14, 2017. Cited on page 166.

[dSOdR+03] Cleidson R B de Souza, Hamilton L R Oliveira, Cleber R P da Rocha, Kléder Miranda Gonçalves, and David F Redmiles. Using Critiquing Systems for Inconsistency Detection in Software Engineering Models. In *International Conference on Software Engineering and Knowledge Engineering (SEKE) 2003, Proceedings*, pages 196–203, 2003. Cited on page 31.

[dtC12] Pedro da Silva Hack and Carla Schwengber ten Caten. Measurement Uncertainty: Literature Review and Research Trends. *IEEE Transactions on Instrumentation and Measurement*, 61(8):2116–2124, 2012. Cited on page 20.

[EC07] Ali Ebnenasir and Betty H C Cheng. Pattern-Based Modeling and Analysis of Failsafe Fault-Tolerance in UML. In *International Symposium on High*

Assurance Systems Engineering (HASE) 2007, Proceedings, pages 275–282. IEEE, 2007. Cited on page 25.

[ECK06] Ali Ebnenasir, Betty H C Cheng, and Sascha Konrad. Use Case-Based Modeling and Analysis of Failsafe Fault-Tolerance. In *International Requirements Engineering Conference (RE) 2006, Proceedings*, pages 336–337. IEEE, 2006. Cited on page 32.

[EDG+11] Alexander Egyed, Andreas Demuth, Achraf Ghabi, Roberto E Lopez-Herrejon, Patrick Mäder, Alexander Nöhrer, and Alexander Reder. Fine-Tuning Model Transformation: Change Propagation in Context of Consistency, Completeness, and Human Guidance. In *International Conference on Model Transformation (ICMT) 2011, Proceedings*, pages 1–14. Springer, 2011. Cited on pages 23, 28, 31, and 32.

[EEE+07] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information Preserving Bidirectional Model Transformations. In *Fundamental Approaches to Software Engineering (FASE) 2007, Proceedings*, pages 72–86. Springer, 2007. Cited on pages 115 and 117.

[EEEP08] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, and Ulrike Prange. Consistent Integration of Models Based on Views of Visual Languages. In *Fundamental Approaches to Software Engineering (FASE) 2008, Proceedings*, pages 62–76. Springer, 2008. Cited on page 24.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006. Cited on pages 44, 50, 59, 67, 78, and 227.

[EET11] Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer. A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In *Fundamental Approaches to Software Engineering (FASE) 2011, Proceedings*, pages 202–216. Springer, 2011. Cited on page 54.

[EGT16] Michael Elberfeld, Martin Grohe, and Till Tantau. Where First-Order and Monadic Second-Order Logic Coincide. *ACM Transactions on Computational Logic*, 17(4):25:1–25:18, 2016. Cited on page 62.

[Egy06] Alexander Egyed. Instant consistency checking for the UML. In *International Conference on Software Engineering (ICSE) 2006, Proceedings*, pages 381–390. ACM, 2006. Cited on pages 18, 23, 25, 28, 32, and 33.

[Egy07a] Alexander Egyed. Fixing Inconsistencies in UML Design Models. In *International Conference on Software Engineering (ICSE) 2007, Proceedings*, pages 292–301. IEEE, 2007. Cited on pages 23, 28, 32, 124, and 125.

[Egy07b] Alexander Egyed. UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models. In *International Conference on Software Engineering (ICSE) 2007, Proceedings*, pages 793–796. IEEE, 2007. Cited on pages 25, 28, 31, and 32.

[Egy11] Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011. Cited on pages 23, 25, 28, 31, and 32.

[EHGB12] Claudia Ermel, Frank Hermann, Jürgen Gall, and Daniel Binanzer. Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars. *Electronic Communication of the European Association of Software Science and Technology*, 54:1–12, 2012. Cited on pages 81, 83, and 165.

[EHS09] Hartmut Ehrig, Frank Hermann, and Christoph Sartorius. Completeness and Correctness of Model Transformations Based on Triple Graph Grammars with Negative Application Conditions. *Electronic Communication of the European Association of Software Science and Technology*, 18, 2009. Cited on pages 54 and 103.

[EHSB13] Hartmut Ehrig, Frank Hermann, Hanna Schölzel, and Christoph Brandt. Propagation of constraints along model transformations using triple graph grammars and borrowed context. *Journal of Visual Languages and Computing*, 24(5):365–388, 2013. Cited on page 240.

[ELF08] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *Automated Software Engineering Conference (ASE) 2008, Proceedings*, pages 99–108. ACM, 2008. Cited on pages 23, 25, 28, 31, and 32.

[EM10] Naeem Esfahani and Sam Malek. Uncertainty in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems II, 2010 Revised Selected and Invited Papers*, pages 214–238. Springer, 2010. Cited on pages 26, 32, and 33.

[EPR14] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. Uncertainty in bidirectional transformations. In *Models in Software Engineering (MiSE) 2014, Proceedings*, pages 37–42. ACM, 2014. Cited on pages 30, 32, and 33.

[EPR15] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. Managing uncertainty in bidirectional model transformations. In *Software Language Engineering (SLE) 2015, Proceedings*, pages 49–58. ACM, 2015. Cited on pages 25 and 33.

[EPT18] Romina Eramo, Alfonso Pierantonio, and Michele Tucci. Enhancing the JTL tool for bidirectional transformations. In *International Conference on Art, Science, and Engineering of Programming (PROGRAMMING) 2018, Proceedings*, pages 36–41. ACM, 2018. Cited on page 103.

[EZV+17] Johannes Eder, Sergey Zverlov, Sebastian Voss, Maged Khalil, and Alexandru Ipatiov. Bringing DSE to Life: Exploring the Design Space of an Industrial Automotive Use Case. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Proceedings*, pages 270–280. IEEE, 2017. Cited on page 218.

[FBDD+15] Michalis Famelis, Naama Ben-David, Alessio Di Sandro, Rick Salay, and Marsha Chechik. Mu-Mmint: An IDE for Model Uncertainty. In *International Conference on Software Engineering (ICSE) 2015, Proceedings*, pages 697–700. IEEE, 2015. Cited on pages 25 and 32.

[FGdL12] Kleinner Farias, Alessandro Garcia, and Carlos José Pereira de Lucena. Evaluating the Impact of Aspects on Inconsistency Detection Effort: A Controlled Experiment. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2012, Proceedings*, pages 219–234. Springer, 2012. Cited on pages 24 and 33.

[FGH06] Peter Feiler, David Gluch, and John Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Carnegie Mellon University, Pittsburgh, PA, 2006. Cited on page 217.

[FHA17] Imen Ben Fraj, Yousra Bendaly Hlaoui, and Leila Jemni Ben Ayed. A Modeling Approach for Flexible Workflow Applications of Cloud Services. In *International Computer Software and Applications Conference (COMPSAC) 2017, Proceedings*, pages 175–180. IEEE, 2017. Cited on page 33.

[FKM+20] Lars Fritsche, Jens Kosiol, Adrian Möller, Andy Schürr, and Gabriele Taentzer. A precedence-driven approach for concurrent model synchronization scenarios using triple graph grammars. In *Software Language Engineering (SLE) 2020, Proceedings*, pages 39–55. ACM, 2020. Cited on pages 117, 126, 128, 142, 170, 178, 183, 194, 199, 239, 241, and 286.

[FKST18] Lars Fritsche, Jens Kosiol, Andy Schürr, and Gabriele Taentzer. Short-Cut Rules - Sequential Composition of Rules Avoiding Unnecessary Deletions. In *Software Technologies: Applications and Foundations (STAF) 2018, Workshop Proceedings*, pages 415–430. Springer, 2018. Cited on pages 128 and 134.

[FLAS17] Lars Fritsche, Erhan Leblebici, Anthony Anjorin, and Andy Schürr. A Look-Ahead Strategy for Rule-Based Model Transformations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings*, pages 45–53. CEUR-WS.org, 2017. Cited on page 90.

[FM18] Adel Ferdjoukh and Jean-Marie Mottu. Towards an Automated Fault Localizer while Designing Meta-models. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Workshop Proceedings*, pages 547–552. CEUR-WS.org, 2018. Cited on page 185.

[FPP08] John Nathan Foster, Alexandre Pilkiewicz, and Benjamin C Pierce. Quotient Lenses. In *International Conference on Functional programming (ICFP) 2008*, pages 383–396. ACM, 2008. Cited on page 83.

[Fre12] Eva Freund. IEEE Standard for System and Software Verification and Validation (IEEE Std 1012-2012). *Software Quality Professional*, 15(1):43, 2012. Cited on page 201.

[Fri18] Jan Friedrich. Declarative project planning and controlling: a formal model to support the handling of unavoidable inconsistencies. In *International Conference on Software and Systems Process (ICSSP) 2018*, pages 61–69. ACM, 2018. Cited on pages 24 and 31.

[Fri21] Lars Fritsche. *Local Consistency Restoration Methods for Triple Graph Grammars*. PhD thesis, Darmstadt University of Technology, Germany, 2021. Cited on page 142.

[FS10] Marc Förster and Daniel Schneider. Flexible, Any-Time Fault Tree Analysis with Component Logic Models. In *International Symposium on Software Reliability Engineering (ISSRE) 2010*, pages 51–60. IEEE, 2010. Cited on pages 26 and 32.

[FS13] Michalis Famelis and Stephanie Santosa. MAV-Vis: A notation for model uncertainty. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) 2013, Proceedings*, pages 7–12. IEEE, 2013. Cited on pages 25, 26, 32, and 33.

[FSBR10] Ana M Fernández-Sáez, Marcela Genero Bocco, and Francisco P Romero. SLR-Tool - A Tool for Performing Systematic Literature Reviews. In *International Conference on Software and Data Technologies (ICSOFT) 2010, Proceedings*, pages 157–166. SciTePress, 2010. Cited on page 20.

[FSC12a] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *International Conference on Software Engineering (ICSE) 2012, Proceedings*, pages 573–583. IEEE, 2012. Cited on pages 18, 25, 32, and 33.

[FSC12b] Michalis Famelis, Rick Salay, and Marsha Chechik. The semantics of partial model transformations. In *Models in Software Engineering (MiSE) 2012, Proceedings*, pages 64–69. IEEE, 2012. Cited on pages 26, 29, and 32.

[FSSC13] Michalis Famelis, Rick Salay, Alessio Di Sandro, and Marsha Chechik. Transformation of Models Containing Uncertainty. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2013, Proceedings*, pages 673–689. Springer, 2013. Cited on pages 26, 30, and 32.

[FTW16] Martin Fleck, Javier Troya, and Manuel Wimmer. Search-Based Model Transformations. *Journal of Software: Evolution and Process*, 28(12):1081–1117, 2016. Cited on pages 81 and 83.

[FYCL09] Guisheng Fan, Huiqun Yu, Liqiong Chen, and Dongmei Liu. A Method for Modeling and Analyzing Fault-Tolerant Service Composition. In *Asia-Pacific Software Engineering Conference (APSEC) 2009, Proceedings*, pages 507–514. CEUR-WS.org, 2009. Cited on page 31.

[GAM16] Miguel Goulão, Vasco Amaral, and Marjan Mernik. Quality in model-driven engineering: a tertiary study. *Software Quality Journal*, 24(3):601–633, 2016. Cited on page 19.

[Gar10] David Garlan. Software engineering in an uncertain world. In *Workshop on Future of Software Engineering Research (FoSER) 2010, Proceedings*, pages 125–128. ACM, 2010. Cited on pages 26 and 32.

[GBT+19] José A Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. Automated analysis of feature models: Quo vadis? *Computing*, 101(5):387–433, 2019. Cited on page 19.

[GC08] Heather Goldsby and Betty H C Cheng. Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In *International*

Conference on Model Driven Engineering Languages and Systems (MoDELS) 2008, Proceedings, pages 568–583. Springer, 2008. Cited on pages 26, 29, and 32.

[GdL06] Esther Guerra and Juan de Lara. Model View Management with Triple Graph Transformation Systems. In *International Conference on Graph Transformation (ICGT) 2006, Proceedings*, pages 351–366. Springer, 2006. Cited on page 39.

[GdL18] Esther Guerra and Juan de Lara. On the Quest for Flexible Modelling. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Proceedings*, pages 23–33. ACM, 2018. Cited on pages 18, 23, 24, 25, and 31.

[GEH10] Ulrike Golas, Hartmut Ehrig, and Annegret Habel. Multi-Amalgamation in Adhesive Categories. In *International Conference on Graph Transformation (ICGT) 2010, Proceedings*, pages 346–361. Springer, 2010. Cited on pages 54 and 69.

[GEH11] Ulrike Golas, Hartmut Ehrig, and Frank Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. *Electronic Communication of the European Association of Software Science and Technology*, 39, 2011. Cited on pages 53, 54, 103, and 240.

[GH09] Holger Giese and Stephan Hildebrandt. Efficient Model Synchronization of Large-Scale Models. Technical report, Hasso-Plattner Institute at the University of Potsdam, Germany, 2009. Cited on pages 81 and 83.

[GHE14] Ulrike Golas, Annegret Habel, and Hartmut Ehrig. Multi-amalgamation of rules with application conditions in M-adhesive categories. *Mathematical Structures in Computer Science*, 24(04):1–68, jun 2014. Cited on page 54.

[GHHS15] Martin Gogolla, Lars Hamann, Frank Hilken, and Matthias Sedlmeier. Checking UML and OCL Model Consistency: An Experience Report on a Middle-Sized Case Study. In *International Conference of Tests and Proofs (TAP) 2015, Proceedings*, pages 129–136. Springer, 2015. Cited on pages 23 and 24.

[GHL14] Holger Giese, Stephan Hildebrandt, and Leen Lambers. Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars - Ensuring Conformance of Relational Model Transformation Specifications and Implementations. *Software and Systems Modeling*, 13(1):273–299, 2014. Cited on pages 54, 165, and 185.

[GHN10] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In *Graph Transformations and Model-Driven Engineering*, pages 555–579. Springer, 2010. Cited on pages 4, 39, and 203.

[GHN+13] Susann Gottmann, Frank Hermann, Nico Nachtigall, Benjamin Braatz, Claudia Ermel, Hartmut Ehrig, and Thomas Engel. Correctness and Completeness of Generalised Concurrent Model Synchronisation Based on Triple Graph Grammars. In *Workshop on the Analysis of Model Transformations (AMT) 2013, Proceedings*. CEUR-WS.org, 2013. Cited on page 126.

[GHYZ11] Xin Gao, Wenhui Hu, Wei Ye, and Shikun Zhang. Data Uncertainty Model for Mashup. In *International Conference on Software Engineering and Knowledge Engineering (SEKE) 2011, Proceedings*, pages 503–508. KKnowledge Systems Institute Graduate School, 2011. Cited on pages 25 and 32.

[GI20] Esther Guerra and Ludovico Iovino, editors. *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2020, Companion Proceedings*. ACM, 2020. Cited on page 243.

[GK10] Joel Greenyer and Ekkart Kindler. Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. *Software and Systems Modeling*, 9(1):21–46, 2010. Cited on page 166.

[GKB21] Joshua Gleitze, Heiko Klare, and Erik Burger. Finding a Universal Execution Strategy for Model Transformation Networks. In *Fundamental Approaches to Software Engineering (FASE) 2021, Proceedings*, pages 87–107. Springer, 2021. Cited on page 243.

[GKH09] Martin Gogolla, Mirco Kuhlmann, and Lars Hamann. Consistency, Independence and Consequences in UML and OCL Models. In *International Conference of Tests and Proofs (TAP) 2009, Proceedings*, pages 90–104. Springer, 2009. Cited on page 24.

[GLO09] Esther Guerra, Juan De Lara, and Fernando Orejas. Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions. In *International Conference on Model Transformation (ICMT) 2009, Proceedings*, pages 83–99. Springer, 2009. Cited on page 59.

[Göt18] Sebastian Götz. Supporting systematic literature reviews in computer science: the systematic literature review toolkit. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Companion Proceedings*, pages 22–26. ACM, 2018. Cited on pages 18 and 20.

[GPR11] Joel Greenyer, Sebastian Pook, and Jan Rieke. Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In *European Conference on Modelling Foundations and Applications (ECMFA) 2011, Proceedings*, pages 144–159. Springer, 2011. Cited on page 54.

[GPST13] Carlo Ghezzi, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *International Conference on Software Engineering (ICSE) 2013, Proceedings*, pages 33–42. IEEE, 2013. Cited on pages 26, 29, and 33.

[GR16] Jeff Gray and Bernhard Rumpe. How to write a successful SoSyM submission. *Software and Systems Modeling*, 15(4):929–931, 2016. Cited on page 19.

[GSS14] Christine M Gerpheide, Ramon R H Schiffelers, and Alexander Serebrenik. A Bottom-Up Quality Model for QVTo. In *International Conference on the Quality of Information and Communications Technology (QUATIC) 2014, Proceedings*, pages 85–94. IEEE, 2014. Cited on page 19.

[GSS16] Christine M Gerpheide, Ramon R H Schiffelers, and Alexander Serebrenik. Assessing and improving quality of QVTo model transformations. *Software Quality Journal*, 24(3):797–834, 2016. Cited on page 19.

[Gue16] Daniela Guericke. *Routing and Scheduling for Home Care Services : Solution Approaches for Static and Dynamic Settings*. PhD thesis, Paderborn University, Germany, 2016. Cited on page 216.

[GW06] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2006, Proceedings*, volume 4199, pages 543–557. Springer, 2006. Cited on pages 141 and 165.

[GW09] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Journal of Software and Systems Modeling*, 8(1):21–43, 2009. Cited on pages 124, 125, and 126.

[GWT+14] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. Variability in Software Systems - A Systematic Literature Review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014. Cited on page 19.

[GZJ16] Wafa Gabsi, Bechir Zalila, and Mohamed Jmaiel. EMA2AOP: From the AADL Error Model Annex to aspect language towards fault tolerant systems. In *International Conference on Software Engineering Research and Applications (SERA) 2016, Proceedings*, pages 155–162. IEEE, 2016. Cited on page 31.

[GZN+14] Liangpeng Guo, Qi Zhu, Pierluigi Nuzzo, Roberto Passerone, Alberto L Sangiovanni-Vincentelli, and Edward A Lee. Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2014, Proceedings*, pages 24:1–24:10. ACM, 2014. Cited on page 217.

[HDH10] Thorsten Höllrigl, Jochen Dinger, and Hannes Hartenstein. A Consistency Model for Identity Information in Distributed Systems. In *International Computer Software and Applications Conference (COMPSAC) 2010, Proceedings*, pages 252–261. IEEE, 2010. Cited on pages 25, 31, and 32.

[HDM05] Hong Wang, Dan Lin, and Min-Qiang Li. A competitive genetic algorithm for resource-constrained project scheduling problem. In *International Conference on Machine Learning and Cybernetics (ICMLC) 2005, Proceedings*, volume 5, pages 2945–2949. IEEE, 2005. Cited on page 218.

[HEC+14] Mahmoud El Hamlaoui, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, and Adil Anwar. Heterogeneous models matching for consistency management. In *Research Challenges in Information Science (RCIS) 2014, Proceedings*, pages 1–12. IEEE, 2014. Cited on page 23.

[HEEO12] Frank Hermann, Hartmut Ehrig, Claudia Ermel, and Fernando Orejas. Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In *Fundamental Approaches to Software Engineering (FASE) 2012, Proceedings*, pages 178–193. Springer, 2012. Cited on page 126.

[HFS+15] Xiao He, Yanmei Fu, Chang-ai Sun, Zhiyi Ma, and Weizhong Shao. Towards Model-Driven Variability-Based Flexible Service Compositions. In *International Computer Software and Applications Conference (COMPSAC) 2015, Proceedings*, pages 298–303. IEEE, 2015. Cited on page 33.

[HGN+13] Frank Hermann, Susann Gottmann, Nico Nachtigall, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, and Thomas Engel. On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In *International Conference on Model Transformation (ICMT) 2013, Proceedings*, volume 7909, pages 50–51. Springer, 2013. Cited on page 39.

[HGN+14] Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, and Claudia Ermel. Triple Graph Grammars in the Large for Translating Satellite Procedures. In *International Conference on Model Transformation (ICMT) 2014, Proceedings*, pages 122–137. Springer, 2014. Cited on page 203.

[HHR+11] Ábel Hegedüs, Ákos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. Quick fix generation for DSMLs. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC) 2011, Proceedings*, pages 17–24. IEEE, 2011. Cited on pages 23, 24, and 32.

[Hil16] Nicolas Hili. A Metamodeling Framework for Promoting Flexibility and Creativity Over Strict Model Conformance. In *Workshop on Flexible Model Driven Engineering (FlexMDE) 2016, Proceedings*, pages 2–11. CEUR-WS.org, 2016. Cited on pages 24 and 33.

[HKB16] Regina Hebig, Djamel Khelladi, and Reda Bendraou. Approaches to Co-Evolution of Metamodels and Models: A Survey. *IEEE Transactions on Software Engineering*, PP:1, 2016. Cited on page 33.

[HKC+14] Sebastian J I Herzig, Benjamin Kruse, Federico Ciccozzi, Joachim Denil, Rick Salay, and Dániel Varró. Towards an Approach for Orchestrating Design Space Exploration Problems to Fix Multi-Paradigm Inconsistencies. In *Workshop on Multi-Paradigm Modeling (MPM) 2014, Proceedings*, pages 61–66. CEUR-WS.org, 2014. Cited on page 146.

[HLBG12] Stephan Hildebrandt, Leen Lambers, Basil Becker, and Holger Giese. Integration of Triple Graph Grammars and Constraints. *Electronic Communication of the European Association of Software Science and Technology*, 54, 2012. Cited on pages 54 and 103.

[HLG+11] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick, and Ingo Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE) 2011, Proceedings*, pages 238–253. Springer, 2011. Cited on page 165.

[HLG+13] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A Survey of Triple Graph Grammar Tools. In *International Workshop on Bidirectional Transformations (Bx) 2013 Proceedings*, volume 57. ECEASST, 2013. Cited on pages 54 and 166.

[HLR08] Thomas Hettel, Michael Lawley, and Kerry Raymond. Model Synchronisation: Definitions for Round-Trip Engineering. In *International Conference on Model Transformation (ICMT) 2008, Proceedings*, pages 31–45. Springer, 2008. Cited on pages 124 and 126.

[HM05] Brahim Hamid and Mohamed Mosbah. A Formal Model for Fault-Tolerance in Distributed Systems. In *International Conference on Computer Safety, Reliability, and Security (SAFECOMP) 2005, Proceedings*, pages 108–121. Springer, 2005. Cited on pages 25 and 31.

[HMM20] Edward Huang, Leon F McGinnis, and Steven W Mitchell. Verifying SysML activity diagrams using formal transformation to Petri nets. *Systems Engineering*, 23(1):118–135, 2020. Cited on page 204.

[HO19] Russ Harmer and Eugenia Oshurko. Knowledge Representation and Update in Hierarchies of Graphs. In *International Conference on Graph Transformation (ICGT) 2019, Proceedings*, pages 141–158. Springer, 2019. Cited on page 44.

[Hoa13] Thai Son Hoang. An introduction to the Event-B modelling method. *Industrial Deployment of System Engineering Methods*, pages 211–236, 2013. Cited on pages 4, 42, and 202.

[Hol92] John H Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992. Cited on page 146.

[Hor17] Tassilo Horn. Solving the TTC Families to Persons Case with FunnyQT. In *Transformation Tool Contest (TTC) 2017, Proceedings*, pages 47–51. CEUR-WS.org, 2017. Cited on pages 83 and 98.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of High-Level Transformation Systems Relative to Nested Conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009. Cited on pages 78 and 101.

[HP19] Jon Holt and Simon Perry. *SysML for Systems Engineering: A model-based approach*. Institution of Engineering and Technology, 2019. Cited on page 39.

[HPP+14] Fernando Herrera, Héctor Posadas, Pablo Peñil, Eugenio Villar, Francisco Ferrero, Raúl Valencia, and Gianluca Palermo. The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems. *Journal of Systems Architecture*, 60(1):55–78, 2014. Cited on page 217.

[HRW11] John Edward Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *International Conference on Software Engineering (ICSE) 2011, Proceedings*, pages 633–642. ACM, 2011. Cited on page 3.

[HS15] Bernhard Hoisl and Stefan Sobernig. Consistency Rules for UML-based Domain-specific Language Models: A Literature Review. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2015, Workshop Proceedings*, volume 1508, pages 29–36. CEUR-WS.org, 2015. Cited on page 19.

[HS17] Nicolas Hili and Jean-Sébastien Sottet. The Conformance Relation Challenge: Building Flexible Modelling Frameworks. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings*, pages 418–423. CEUR-WS.org, 2017. Cited on pages 24 and 33.

[HT99] Klaus Marius Hansen and Michael Thomsen. The "Domain Model Concealer" and "Application Moderator" Patterns: Addressing Architectural Uncertainty in Interactive Systems. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS) 1999, Proceedings*, pages 177–190. IEEE, 1999. Cited on pages 25, 26, 32, and 33.

[HTJ92] Jin-Kao Hao, F Trousset, and Jean Jacques. Prototyping an Inconsistency Checking Tool for Software Process Models. In *International Conference on Software Engineering and Knowledge Engineering (SEKE) 1992, Proceedings*, pages 227–234. IEEE, 1992. Cited on pages 24 and 32.

[HWF13] Kiyoshi Honda, Hironori Washizaki, and Yoshiaki Fukazawa. A Generalized Software Reliability Model Considering Uncertainty and Dynamics in Development. In *Product-Focused Software Process Improvement (PROFES) 2013, Proceedings*, pages 342–346. Springer, 2013. Cited on pages 32 and 33.

[IFED09] Hamdy Ibrahim, Behrouz Homayoun Far, Armin Eberlein, and Y Daradkeh. Uncertainty management in software engineering: Past, present, and future. In *Canadian Conference on Electrical and Computer Engineering (CCECE) 2009, Proceedings*, pages 7–12. IEEE, 2009. Cited on pages 26, 32, and 33.

[IKJ19] Asif Iqbal, Iftikhar Ahmed Khan, and Sadaqat Jan. A Review and Comparison of the Traditional Collaborative and Online Collaborative Techniques for Software Requirement Elicitation. In *International Conference on Advancements in Computational Sciences (ICACS) 2019, Proceedings*, pages 1–8. IEEE, 2019. Cited on page 19.

[JBEM10] Ivan Jureta, Alexander Borgida, Neil A Ernst, and John Mylopoulos. Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling. In *International Requirements Engineering Conference (RE) 2010, Proceedings*, pages 115–124. IEEE, 2010. Cited on page 32.

[JKT16] Azadeh Jahanbanifar, Ferhat Khendek, and Maria Toeroe. Runtime Adjustment of Configuration Models for Consistency Preservation. In *International Symposium on High Assurance Systems Engineering (HASE) 2016, Proceedings*, pages 102–109. IEEE, 2016. Cited on pages 23 and 31.

[JLC+18] Ruihua Ji, Zhong Li, Shouyu Chen, Minxue Pan, Tian Zhang, Shaukat Ali, Tao Yue, and Xuandong Li. Uncovering Unknown System Behaviors in Uncertain Networks with Model and Search-Based Testing. In *International Conference on Software Testing, Verification, and Validation (ICST) 2018, Proceedings*, pages 204–214. IEEE, 2018. Cited on page 29.

[Jos21] Jane Jose. Large-Scale Model Transformation Debugging with Configurable Breakpoints. Master thesis, Paderborn University, Germany. 2021. Cited on pages 15 and 194.

[JVV17] Maris Jukss, Clark Verbrugge, and Hans Vangheluwe. Transformations Debugging Transformations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings*, pages 449–454. CEUR-WS.org, 2017. Cited on page 184.

[JWY+20] Ivan Jovanovikj, Nils Weidmann, Enes Yigitbas, Anthony Anjorin, Stefan Sauer, and Gregor Engels. A Model-Driven Mutation Framework for Validation of Test Case Migration. In *International Conference on Systems Modelling and Management (ICSMM) 2020, Proceedings*, pages 21–29. Springer, 2020. Cited on page 16.

[Kan20] Suganya Kannan. Systematic Literature Review on Tolerance in Model-Driven Engineering. Master thesis, Paderborn University, Germany. 2020. Cited on page 15.

[KBB+09] Barbara A Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen G Linkman. Systematic literature reviews in software engineering - A systematic literature review. *Information and Software Technology*, 51(1):7–15, 2009. Cited on page 20.

[KBB+16] Rudolf Kruse, Christian Borgelt, Christian Braune, Sanaz Mostaghim, and Matthias Steinbrecher. *Computational Intelligence - A Methodological Introduction, Second Edition*. Springer, 2016. Cited on page 151.

[KBSB10] Marouane Kessentini, Arbi Bouchoucha, Houari A Sahraoui, and Mounir Boukadoum. Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search. In *European Conference on Modelling Foundations and Applications (ECMFA) 2010, Proceedings*, pages 156–172. Springer, 2010. Cited on page 147.

[KCT+15] Christos Kyrkou, Eftychios Christoforou, Theocharis Theocharides, Christoforos Panayiotou, and Marios M Polycarpou. A camera uncertainty model for collaborative visual sensor network applications. In *International Conference on Distributed Smart Cameras (ICDSC) 2015, Proceedings*, pages 86–91. ACM, 2015. Cited on pages 24, 26, and 32.

[KEK+15] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger, and Johannes Schönböck. Consistent co-evolution of models and transformations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2015, Proceedings*, pages 116–125. IEEE, 2015. Cited on pages 24 and 30.

[KG19] Heiko Klare and Joshua Gleitze. Commonalities for Preserving Consistency of Multiple Models. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2019, Companion Proceedings*, pages 371–378. IEEE, 2019. Cited on page 55.

[KGV05] Aneesh Krishna, Aditya K Ghose, and Sergiy A Vilkomir. Loosely-coupled Consistency between Agent-oriented Conceptual Models and Z Specifications. In *International Conference on Software Engineering and Knowledge Engineering (SEKE) 2005, Proceedings*, pages 455–460, 2005. Cited on page 23.

[KH06] Shinya Kawanaka and Haruo Hosoya. biXid: A Bidirectional Transformation Language for XML. In *International Conference on Functional Programming (ICFP) 2006, Proceedings*, pages 201–214. ACM, 2006. Cited on pages 81 and 83.

[KHJS14] Hasan Koç, Erik Hennig, Stefan Jastram, and Christoph Starke. State of the Art in Context Modelling - A Systematic Literature Review. In *International Conference on Advanced Information Systems Engineering (CAiSE) 2014, Workshop Proceedings*, volume 178, pages 53–64. Springer, 2014. Cited on page 19.

[KHW12] A Krasnogolowy, S Hildebrandt, and S Wätzoldt. Flexible Debugging of Behavior Models. In *International Conference on Industrial Technology (ICIT) 2012, Proceedings*, pages 331–336. IEEE, 2012. Cited on page 185.

[Kit04] Barbara A Kitchenham. Procedures for Performing Systematic Reviews. Technical report, Keele University, UK, 2004. Cited on pages 11 and 19.

[KJS10] Eunsuk Kang, Ethan K Jackson, and Wolfram Schulte. An Approach for Effective Design Space Exploration. In *Monterey Workshop 2010, Proceedings*, pages 33–54. Springer, 2010. Cited on page 218.

[KJV83] Scott Kirkpatrick, D Gelatt Jr., and Mario P Vecchi. Optimization by Simmulated Annealing. *Science*, 220(4598):671–680, 1983. Cited on page 146.

[KKD+17] Roland Kretschmer, Djamel Eddine Khelladi, Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. From Abstract to Concrete Repairs of Model Inconsistencies: An Automated Approach. In *Asia-Pacific Software Engineering Conference (APSEC) 2017, Proceedings*, pages 456–465. CEUR-WS.org, 2017. Cited on pages 23, 28, and 32.

[KKE18] Roland Kretschmer, Djamel Eddine Khelladi, and Alexander Egyed. An automated and instant discovery of concrete repairs for model inconsistencies. In *International Conference on Software Engineering (ICSE) 2018, Proceedings*, pages 298–299. ACM, 2018. Cited on pages 23 and 31.

[KKE19] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. Detecting and exploring side effects when repairing model inconsistencies. In *Software Language Engineering (SLE) 2019, Proceedings*, pages 113–126. ACM, 2019. Cited on pages 23, 24, 28, 31, and 32.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC-FSE) 2007, Proceedings*, pages 285–294. ACM, 2007. Cited on pages 39, 165, and 243.

[Kla21] Heiko Klare. *Building Transformation Networks for Consistent Evolution of Interrelated Models*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2021. Cited on pages 55 and 243.

[KLKS10] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In *Graph Transformations and Model Driven Engineering*, pages 141–174. Springer, 2010. Cited on pages 103 and 134.

[KN18] Barbara König and Dennis Nolte. CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers (Tool Presentation Paper). In *International Conference on Graph Transformation (ICGT) 2018, Proceedings*. Springer, 2018. Cited on page 82.

[KP14] Stefan Kugele and Gheorghe Pucea. Model-based optimization of automotive E/E-architectures. In *International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA) 2014, Proceedings*, pages 18–29. ACM, 2014. Cited on page 218.

[KPP08a] Dimitrios S Kolovos, Richard F Paige, and Fiona Polack. Detecting and Repairing Inconsistencies across Heterogeneous Models. In *International Conference on Software Testing, Verification, and Validation (ICST) 2008, Proceedings*, pages 356–364. IEEE, 2008. Cited on pages 25 and 32.

[KPP08b] Dimitrios S Kolovos, Richard F Paige, and Fiona Polack. The Grand Challenge of Scalability for Model Driven Engineering. In *Models in Software Engineering (MiSE) 2008, Proceedings*, pages 48–53. Springer, 2008. Cited on pages 124, 125, and 126.

[KPP+15] Stefan Kugele, Gheorghe Pucea, Ramona Popa, Laurent Dieudonné, and Horst Eckardt. On the deployment problem of embedded systems. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE) 2015, Proceedings*, pages 158–167. IEEE, 2015. Cited on page 218.

[KR07] Jochen Malte Küster and Ksenia Ryndina. Improving Inconsistency Resolution with Side-Effect Evaluation and Costs. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2007, Proceedings*, pages 136–150. Springer, 2007. Cited on pages 24 and 32.

[KSB08] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model Transformation as an Optimization Problem. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2008, Proceedings*, pages 159–173. Springer, 2008. Cited on pages 81, 83, and 147.

[KSBB12] Marouane Kessentini, Houari A Sahraoui, Mounir Boukadoum, and Omar Benomar. Search-based model transformation by example. *Software and Systems Modeling*, 11(2):209–226, 2012. Cited on page 147.

[KSG+17] Roland Kluge, Michael Stein, David Giessing, Andy Schürr, and Max Mühlhäuser. cMoflon: Model-Driven Generation of Embedded C Code for Wireless Sensor Networks. In *European Conference on Modelling Foundations and Applications (ECMFA) 2017, Proceedings*, pages 109–125. Springer, 2017. Cited on page 164.

[KSTZ20] Jens Kosiol, Daniel Strüber, Gabriele Taentzer, and Steffen Zschaler. Graph Consistency as a Graduated Property - Consistency-Sustaining and -Improving Graph Transformations. In *International Conference on Graph Transformation (ICGT) 2020, Proceedings*, volume 12150, pages 239–256. Springer, 2020. Cited on page 103.

[KW07] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report, Paderborn University, Germany, 2007. Cited on page 54.

[KW12] Lilija Klassen and Robert Wagner. EMorF - A Tool for Model Transformations. *Electronic Communication of the European Association of Software Science and Technology*, 54, 2012. Cited on page 165.

[KWLW13] Marouane Kessentini, Wafa Werda, Philip Langer, and Manuel Wimmer. Search-based model merging. In *Genetic and Evolutionary Computation Conference (GECCO) 2013, Proceedings*, pages 1453–1460. ACM, 2013. Cited on page 146.

[KZH16] Hsiang-shang Ko, Tao Zan, and Zhenjiang Hu. BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming. In *Workshop on Partial Evaluation and Program Manipulation (PEPM) 2016, Proceedings*, pages 61–72. ACM, 2016. Cited on pages 81, 83, 98, and 203.

[LAF+17] Erhan Leblebici, Anthony Anjorin, Lars Fritsche, Gergely Varro, and Andy Schürr. Leveraging Incremental Pattern Matching Techniques for Model Synchronisation. In *International Conference on Graph Transformation (ICGT) 2017, Proceedings*, pages 179–195. Springer, 2017. Cited on pages 103, 141, and 238.

[LAS14a] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Developing eMoflon with eMoflon. In *International Conference on Model Transformation (ICMT) 2014, Proceedings*, pages 138–145. Springer, 2014. Cited on pages 54, 83, 164, and 165.

[LAS+14b] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A Comparison of Incremental Triple Graph Grammar Tools. *Electronic Communication of the European Association of Software Science and Technology*, 67, 2014. Cited on pages 54 and 166.

[LAS15] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Tool Support for Multi-amalgamated Triple Graph Grammars. In *International Conference on Graph Transformation (ICGT) 2015, Proceedings*, pages 257–265. Springer, 2015. Cited on pages 54 and 239.

[LAS17] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. Inter-model Consistency Checking using Triple Graph Grammars and Linear Optimization Techniques. In *Fundamental Approaches to Software Engineering (FASE) 2017, Proceedings*, pages 191–207. Springer, 2017. Cited on pages 13, 23, 24, 82, 84, 97, and 99.

[LAST15] Erhan Leblebici, Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Multi-amalgamated Triple Graph Grammars. In Francesco Parisi-Presicce and Bernhard Westfechtel, editors, *International Conference on Graph Transformation (ICGT) 2015, Proceedings*, pages 87–103. Springer, 2015. Cited on pages 54 and 239.

[LAST17] Erhan Leblebici, Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. Multi-amalgamated triple graph grammars: Formal foundation and application to visual language translation. *Journal of Visual Language and Computing*, 42:99–121, 2017. Cited on pages 53, 54, 69, 70, 72, and 73.

[Lau13] Marius Lauder. *Incremental Model Synchronization with Precedence-Driven Triple Graph Grammars*. PhD thesis, Darmstadt University of Technology, Germany, 2013. Cited on page 96.

[LBG13] Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais. Executing and debugging UML models: an fUML extension. In *Symposium on Applied Computing (SAC) 2013, Proceedings*, pages 1095–1102. ACM, 2013. Cited on page 184.

[Leb16] Erhan Leblebici. Towards a Graph Grammar-Based Approach to Inter-Model Consistency Checks with Traceability Support. In *International Workshop on Bidirectional Transformations (Bx) 2016, Proceedings*, pages 35–39. CEUR-WS.org, 2016. Cited on pages 82 and 97.

[Leb18] Erhan Leblebici. *Inter-Model Consistency Checking and Restoration with Triple Graph Grammars*. PhD thesis, Darmstadt University of Technology, Germany, 2018. Cited on pages 13, 14, 82, 84, 100, 102, 112, and 117.

[LHGO12] Leen Lambers, Stephan Hildebrandt, Holger Giese, and Fernando Orejas. Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case. *Electronic Communication of the European Association of Software Science and Technology*, 49, 2012. Cited on pages 53, 54, 59, and 83.

[LKV11] Ricky T Lindeman, Lennart C L Kats, and Eelco Visser. Declaratively Defining Domain-Specific Language Debuggers. In *International Conference on Generative Programming and Component Engineering (GPCE) 2011, Proceedings*, pages 127–136. ACM, 2011. Cited on page 184.

[LTZ12] Ioanna Lytra, Huy Tran, and Uwe Zdun. Constraint-Based Consistency Checking between Design Decisions and Component Models for Supporting Software Architecture Evolution. In *European Conference on Software Maintenance and Reengineering (CSMR) 2012, Proceedings*, pages 287–296. IEEE, 2012. Cited on page 23.

[LTZ13] Ioanna Lytra, Huy Tran, and Uwe Zdun. Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations. In *European Conference on Software Architecture (ECSA) 2013, Proceedings*, pages 224–239. Springer, 2013. Cited on page 23.

[MC13] Nuno Macedo and Alcino Cunha. Implementing QVT-R Bidirectional Model Transformations using Alloy. In *Fundamental Approaches to Software Engineering (FASE) 2013, Proceedings*, pages 297–311. Springer, 2013. Cited on pages 23, 81, 82, 83, and 125.

[MCK+20] Gunter Mussbacher, Benoît Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari A Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow. Opportunities in intelligent modeling assistance. *Software and Systems Modeling*, 19(5):1045–1053, 2020. Cited on page 3.

[MdddM15] Marcelo Luiz Monteiro Marinho, Suzana Cândido de Barros Sampaio, Telma Lúcia de Andrade Lima, and Hermano Perrelli de Moura. Uncertainty Management in Software Projects. *Journal of Software*, 10(3):288–303, 2015. Cited on page 20.

[MKKJ10] Brice Morin, Jacques Klein, Jörg Kienzle, and Jean-Marc Jézéquel. Flexible Model Element Introduction Policies for Aspect-Oriented Modeling. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2010, Proceedings*, pages 63–77. Springer, 2010. Cited on page 24.

[MKL+15] Usman Mansoor, Marouane Kessentini, Philip Langer, Manuel Wimmer, Slim Bechikh, and Kalyanmoy Deb. MOMM: Multi-Objective Model Merging. *Journal of Systems and Software*, 103:423–439, 2015. Cited on page 146.

[MM19] Mahyar Tourchi Moghaddam and Henry Muccini. Fault-Tolerant IoT - A Systematic Mapping Study. In *Software Engineering for Resilient Systems (SERENE) 2019, Proceedings*, pages 67–84. Springer, 2019. Cited on page 20.

[MSBV20] Kristóf Marussy, Oszkár Semeráth, Aren A Babikian, and Dániel Varró. A Specification Language for Consistent Model Generation based on Partial Models. *Journal of Object Technology*, 19(3):3:1–22, 2020. Cited on page 103.

[MSD06] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2006, Proceedings*, pages 200–214. Springer, 2006. Cited on pages 23 and 32.

[MSdM18] Marcelo Luiz Monteiro Marinho, Suzana Sampaio, and Hermano Perrelli de Moura. Managing uncertainty in software projects. *Innovations in Systems and Software Engineering*, 14(3):157–181, 2018. Cited on page 20.

[MTV17] Simon Van Mierlo, Yentl Van Tendeloo, and Hans Vangheluwe. Debugging Parallel DEVS. *Simulation*, 93(4):285–306, 2017. Cited on page 185.

[MTV18] Simon Van Mierlo, Yentl Van Tendeloo, and Hans Vangheluwe. A Generalized Stepping Semantics for Model Debugging. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Workshop Proceedings*, pages 541–546. CEUR-WS.org, 2018. Cited on page 184.

[MTZ17] Faiz Ul Muram, Huy Tran, and Uwe Zdun. Systematic Review of Software Behavioral Model Consistency Checking. *ACM Computing Surveys*, 50(2):17:1–17:39, 2017. Cited on page 19.

[MV17] Simon Van Mierlo and Hans Vangheluwe. Debugging Non-determinism: a Petrinets Modelling, Analysis, and Debugging Tool. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings*, pages 460–462. CEUR-WS.org, 2017. Cited on page 184.

[MVD08] Ricardo Martinho, João Varajão, and Dulce Domingos. A Two-Step Approach for Modelling Flexibility in Software Processes. In *Automated*

Software Engineering Conference (ASE) 2008, Proceedings, pages 427–430. ACM, 2008. Cited on pages 25 and 31.

[MWV16] Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. Adding uncertainty and units to quantity types in software models. In *Software Language Engineering (SLE) 2015, Proceedings*, pages 118–131. ACM, 2016. Cited on pages 26, 29, and 32.

[NCEF01] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2001. Cited on pages 23 and 25.

[NEF03] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency Management with Repair Actions. In *International Conference on Software Engineering (ICSE) 2003, Proceedings*, pages 455–464. IEEE, 2003. Cited on page 23.

[NER01] Bashar Nuseibeh, Steve M Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, 2001. Cited on pages 23, 25, and 31.

[NGTS10] Florian Noyrit, Sébastien Gérard, François Terrier, and Bran Selic. Consistent Modeling Using Multiple UML Profiles. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2010, Proceedings*, pages 392–406. Springer, 2010. Cited on page 24.

[NRB+14] Amanda Sávio Nascimento, Cecilia M F Rubira, Rachel Burrows, Fernando Castor, and Patrick H S Brito. Designing fault-tolerant SOA based on design diversity. *Journal of Software Engineering Research and Development*, 2:13, 2014. Cited on page 19.

[NSKB03] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-Based Design-Space Exploration and Model Synthesis. In *International Conference on Embedded Software (EMSOFT) 2003, Proceedings*, pages 290–305. Springer, 2003. Cited on page 218.

[OBE+13] Fernando Orejas, Artur Boronat, Hartmut Ehrig, Frank Hermann, and Hanna Schölzel. On Propagation-Based Concurrent Model Synchronization. *Electronic Communication of the European Association of Software Science and Technology*, 57, 2013. Cited on pages 126 and 140.

[OLVV18] Bentley James Oakes, Levi Lucio, Clark Verbrugge, and Hans Vangheluwe. Debugging of Model Transformations and Contracts in SyVOLT. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Workshop Proceedings*, pages 532–537. CEUR-WS.org, 2018. Cited on page 185.

[OPN20] Fernando Orejas, Elvira Pino, and Marisa Navarro. Incremental Concurrent Model Synchronization using Triple Graph Grammars. In *Fundamental Approaches to Software Engineering (FASE) 2020, Proceedings*, pages 273–293. Springer, 2020. Cited on pages 14, 124, 125, 128, and 239.

[Opp18] Robin Oppermann. A Configurable, Model-Driven Approach to Optimal Scheduling using Triple Graph Grammars and Linear Programming. Master thesis, Paderborn University, Germany. 2018. Cited on pages 15, 169, and 228.

[ORS09] Xinming Ou, Siva Raj Rajagopalan, and Sakthiyuvaraja Sakthivelmurugan. An Empirical Approach to Modeling Uncertainty in Intrusion Analysis. In *Annual Computer Security Applications Conference (ACSAC) 2009, Proceedings*, pages 494–503. IEEE, 2009. Cited on page 33.

[PBBT09] Gilles Perrouin, Erwan Brottier, Benoit Baudry, and Yves Le Traon. Composing Models for Detecting Inconsistencies: A Requirements Engineering Perspective. In *Requirements Engineering: Foundation for Software Quality (REFSQ) 2009, Proceedings*, pages 89–103. Springer, 2009. Cited on pages 24, 28, 31, and 32.

[PBG14] Rui Pais, João Paulo Barros, and Luís Gomes. From SysML state machines to Petri nets using ATL transformations. In *Doctoral Conference on Computing, Electrical and Industrial Systems*, pages 227–236. Springer, 2014. Cited on page 204.

[PFMM08] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic Mapping Studies in Software Engineering. In *International Conference on Evaluation and Assessment in Software Engineering (EASE) 2008, Proceedings*. BCS, 2008. Cited on page 19.

[PH19] Uwe Pohlmann and Marcus Hüwe. Model-Driven Allocation Engineering: Specifying and Solving Constraints based on the Example of Automotive Systems. *Automated Software Engineering*, 26(2):315–378, 2019. Cited on pages 216 and 218.

[PK04] Amit M Paradkar and Tim Klinger. Automated Consistency and Completeness Checking of Testing Models for Interactive Systems. In *International Computer Software and Applications Conference (COMPSAC) 2004, Proceedings*, pages 342–348. IEEE, 2004. Cited on page 31.

[PK19] I S W B Prasetya and Rick Klomp. Test Model Coverage Analysis Under Uncertainty. In *International Conference on Software Engineering and Formal Methods (SEFM) 2019, Proceedings*, pages 222–239. Springer, 2019. Cited on pages 25 and 32.

[PP14] Christopher M Poskitt and Detlef Plump. Verifying Monadic Second-Order Properties of Graph Programs. In *International Conference on Graph Transformation (ICGT) 2014, Proceedings*, pages 33–48. Springer, 2014. Cited on page 78.

[PSG03] Benjamin C Pierce, A Schmitt, and Michael Greenwald. Bringing harmony to optimism: a synchronization framework for heterogeneous tree-structured data. *Technical Report MS-CIS-03-42*, 2003. Cited on pages 124 and 125.

[PST+97] Karin Petersen, Mike Spreitzer, Douglas B Terry, Marvin Theimer, and Alan J Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Symposium on Operating Systems Principles (SOSP) 1997, Proceedings*, pages 288–301. ACM, 1997. Cited on page 24.

[RCBS12] Andres J Ramirez, Betty H C Cheng, Nelly Bencomo, and Pete Sawyer. Relaxing Claims: Coping with Uncertainty While Evaluating Assumptions at Run Time. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2010, Proceedings*, pages 53–69. Springer, 2012. Cited on pages 26 and 32.

[RE12a] Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In *International Conference on Automated Software Engineering (ASE) 2012, Proceedings*, pages 220–229. ACM, 2012. Cited on pages 23 and 32.

[RE12b] Alexander Reder and Alexander Egyed. Incremental Consistency Checking for Complex Design Rules and Larger Model Changes. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2012, Proceedings*, pages 202–218. Springer, 2012. Cited on pages 24, 28, and 31.

[RE13] Alexander Reder and Alexander Egyed. Determining the Cause of a Design Model Inconsistency. *IEEE Transactions on Software Engineering*, 39(11):1531–1548, 2013. Cited on pages 23, 24, and 33.

[Red11] Alexander Reder. Inconsistency management framework for model-based development. In *International Conference on Software Engineering (ICSE) 2011, Proceedings*, pages 1098–1101. ACM, 2011. Cited on pages 23, 25, and 32.

[REDE14] Markus Riedl-Ehrenleitner, Andreas Demuth, and Alexander Egyed. Towards Model-and-Code Consistency Checking. In *International Computer Software and Applications Conference (COMPSAC) 2014, Proceedings*, pages 85–90. IEEE, 2014. Cited on page 23.

[RET11] Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 - New Features for Specifying and Analyzing Algebraic Graph Transformations. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE) 2011, Proceedings*, pages 81–88. Springer, 2011. Cited on page 184.

[Rie15] Jan Rieke. *Model Consistency Management for Systems Engineering.* PhD thesis, Paderborn University, Germany, 2015. Cited on page 185.

[RKPP09] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, and Fiona A C Polack. Enhanced Automation for Managing Model and Metamodel Inconsistency. In *Automated Software Engineering Conference (ASE) 2009, Proceedings*, pages 545–549. ACM, 2009. Cited on pages 24, 31, and 33.

[Rob18] Patrick Robrecht. Incremental Unidirectional Model Transformation via Graph Transformation with eMoflon::IBeX. Master thesis, Paderborn University, Germany. 2018. Cited on pages 15 and 180.

[Roq16] Pascal Roques. MBSE with the ARCADIA Method and the Capella Tool. In *European Congress on Embedded Real Time Software and Systems (ERTS) 2016, Proceedings*, 2016. Cited on page 192.

[Sal20] Shubhangi Salunkhe. Automatic Transformation of SysML Model To Event-B Model. Master thesis, Paderborn University, Germany. 2020. Cited on pages 15 and 208.

[SB06] Colin F Snook and Michael J Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, 2006. Cited on page 202.

[SB16] Jean-Sébastien Sottet and Nicolas Biri. JSMF: a Javascript Flexible Modelling Framework. In *Workshop on Flexible Model Driven Engineering (FlexMDE) 2016, Proceedings*, pages 42–51. CEUR-WS.org, 2016. Cited on pages 24, 30, and 33.

[SBL+20] Oszkár Semeráth, Aren A Babikian, Anqi Li, Kristóf Marussy, and Dániel Varró. Automated generation of consistent models with structural and attribute constraints. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2020, Proceedings*, pages 187–199. ACM, 2020. Cited on page 103.

[SBN+14] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan K Jackson. OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems. In *From Programs to Systems (FPS) 2014, Proceedings*, pages 235–248. Springer, 2014. Cited on page 218.

[SCG12] Rick Salay, Marsha Chechik, and Jan Gorzny. Towards a Methodology for Verifying Partial Model Refinements. In *International Conference on Software Testing, Verification, and Validation (ICST) 2012, Proceedings*, pages 938–945. IEEE, 2012. Cited on pages 26, 30, 32, and 33.

[Sch89] Andy Schürr. Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG) 1989, Proceedings*, pages 151–165. Springer, 1989. Cited on page 164.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG) 1994, Proceedings*, pages 151–163. Springer, 1994. Cited on pages 11 and 44.

[Sch05] Nicole Schweikardt. Arithmetic, first-order logic, and counting quantifiers. *ACM Transactions on Computational Logic*, 6(3):634–671, 2005. Cited on page 62.

[SCH12] Rick Salay, Marsha Chechik, and Jennifer Horkoff. Managing requirements uncertainty with partial models. In *International Requirements Engineering Conference (RE) 2012, Proceedings*, pages 1–10. IEEE, 2012. Cited on pages 26 and 32.

[SCV13] Ivan Svogor, Ivica Crnkovic, and Neven Vrcek. An Extended Model for Multi-Criteria Software Component Allocation on a Heterogeneous Embedded Platform. *Journal of Computing and Information Technology*, 21(4):211–222, 2013. Cited on page 218.

[SFC12] Rick Salay, Michalis Famelis, and Marsha Chechik. Language Independent Refinement Using Partial Modeling. In *Fundamental Approaches to Software Engineering (FASE) 2012, Proceedings*, volume 7212, pages 224–239. Springer, 2012. Cited on pages 26 and 30.

[SHNS13] Hajer Saada, Marianne Huchard, Clémentine Nebut, and Houari A Sahraoui. Recovering model transformation traces using multi-objective optimization. In *International Conference on Automated Software Engineering (ASE) 2013, Proceedings*, pages 688–693. ACM, 2013. Cited on page 146.

[SK08] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In *International Conference on Graph Transformation (ICGT) 2008, Proceedings*, pages 411–425. Springer, 2008. Cited on page 54.

[SK21] Timur Saglam and Heiko Klare. Classifying and Avoiding Compatibility Issues in Networks of Bidirectional Transformations. In *International Workshop on Bidirectional Transformations (Bx) 2021, Proceedings*, pages 34–53. CEUR-WS.org, 2021. Cited on page 243.

[SKE+14] Johannes Schönböck, Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Wieland Schwinger, Manuel Wimmer, and Martin Wischenbart. CARE - A Constraint-Based Approach for Re-Establishing Conformance-Relationships. In *Asia-Pacific Conference on Conceptual Modelling (APCCM) 2014, Proceedings*, volume 154 of *CRPIT*, pages 19–28. Australian Computer Society, 2014. Cited on pages 24 and 31.

[SKLR20] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. Towards Multiple Model Synchronization with Comprehensive Systems. In *Fundamental Approaches to Software Engineering (FASE) 2020, Proceedings*, pages 335–356. Springer, 2020. Cited on pages 55 and 243.

[SKLR21] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*, jul 2021. Cited on pages 55 and 243.

[SKW+13] Johannes Schönböck, Gerti Kappel, Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. TETRABox - A Generic White-Box Testing Framework for Model Transformations. In *Asia-Pacific Software Engineering Conference (APSEC) 2013, Proceedings*, pages 75–82. CEUR-WS.org, 2013. Cited on page 185.

[SMSJ03] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using Description Logic to Maintain Consistency between UML Models. In *International Conference on the Unified Modeling Language (UML) 2003, Proceedings*, pages 326–340. Springer, 2003. Cited on page 24.

[SNEC08] Mehrdad Sabetzadeh, Shiva Nejati, Steve M Easterbrook, and Marsha Chechik. Global consistency checking of distributed models with TReMer+. In *International Conference on Software Engineering (ICSE) 2008, Proceedings*, pages 815–818. ACM, 2008. Cited on page 23.

[SNL+07] Mehrdad Sabetzadeh, Shiva Nejati, Sotirios Liaskos, Steve M Easterbrook, and Marsha Chechik. Consistency Checking of Conceptual Models via Model Merging. In *International Requirements Engineering Conference (RE) 2007, Proceedings*, pages 221–230. IEEE, 2007. Cited on page 23.

[SOY17] Ahmad M Salih, Mazni Omar, and Azman Yasin. Understanding Uncertainty of Software Requirements Engineering: A Systematic Literature

Review Protocol. In *Asia Pacific Requirements Engineering Symposium (APRES) 2017, Proceedings*, volume 809, pages 164–171. Springer, 2017. Cited on page 20.

[SPV20] Alex Serban, Erik Poll, and Joost Visser. Towards Using Probabilistic Models to Design Software Systems with Inherent Uncertainty. In *European Conference on Software Architecture (ECSA) 2020, Proceedings*, pages 89–97. Springer, 2020. Cited on pages 26 and 30.

[Sri21] Ankita Srivastava. Visualization of Concurrent Synchronization Processes based on Triple Graph Grammars. Master thesis, Paderborn University, Germany. 2021. Cited on pages 15 and 194.

[ST15] Faezeh Siavashi and Dragos Truscan. Environment modeling in model-based testing: concepts, prospects and research challenges: a systematic literature review. In *International Conference on Evaluation and Assessment in Software Engineering (EASE) 2015, Proceedings*, pages 30:1–30:6. ACM, 2015. Cited on page 19.

[Ste14] Perdita Stevens. Bidirectionally Tolerating Inconsistency: Partial Transformations. In *Fundamental Approaches to Software Engineering (FASE) 2014, Proceedings*, pages 32–46. Springer, 2014. Cited on pages 5, 17, 18, 23, 25, 31, 38, and 237.

[Ste17] Perdita Stevens. Bidirectional Transformations in the Large. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Proceedings*, pages 1–11. IEEE, 2017. Cited on pages 4, 23, 25, and 243.

[Ste18a] Perdita Stevens. Is Bidirectionality Important? In *European Conference on Modelling Foundations and Applications (ECMFA) 2018, Proceedings*, pages 1–11. Springer, 2018. Cited on page 23.

[Ste18b] Perdita Stevens. Towards sound, optimal, and flexible building from megamodels. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Proceedings*, pages 301–311. ACM, 2018. Cited on pages 23 and 31.

[SV17] Oszkár Semeráth and Dániel Varró. Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In *International Conference on Model Transformation (ICMT) 2017, Proceedings*, volume 10374, pages 138–154. Springer, 2017. Cited on page 103.

[SVL15] Eugene Syriani, Hans Vangheluwe, and Brian Lashomb. T-Core: A Framework for Custom-built Model Transformation Engines. *Software and Systems Modeling*, 14(3):1215–1243, 2015. Cited on page 83.

[SZ04] Lijun Shan and Hong Zhu. Consistency Check in Modelling Multi-Agent Systems. In *International Computer Software and Applications Conference (COMPSAC) 2004, Proceedings*, pages 114–119. IEEE, 2004. Cited on pages 23 and 33.

[SZ06] Lijun Shan and Hong Zhu. Specifying Consistency Constraints for Modelling Languages. In *International Conference on Software Engineering and*

Knowledge Engineering (SEKE) 2006, Proceedings, pages 578–583, 2006. Cited on page 23.

[SZ16] Michael Szvetits and Uwe Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software and Systems Modeling*, 15(1):31–69, 2016. Cited on page 19.

[TA16] Frank Trollmann and Sahin Albayrak. Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models. In *International Conference on Model Transformation (ICMT) 2016, Proceedings*, pages 91–106. Springer, 2016. Cited on pages 55 and 243.

[TA17] Frank Trollmann and Sahin Albayrak. Decision Points for Non-determinism in Concurrent Model Synchronization with Triple Graph Grammars. In *International Conference on Model Transformation (ICMT) 2017, Proceedings*, pages 35–50. Springer, 2017. Cited on pages 55, 124, and 126.

[TBK17] Matthias Tichy, Luis Beaucamp, and Stefan Kögel. Towards Debugging the Matching of Henshin Model Transformations Rules. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings*, pages 455–456. CEUR-WS.org, 2017. Cited on page 184.

[TBSA11] Frank Trollmann, Marco Blumendorf, Veit Schwartze, and Sahin Albayrak. Formalizing model consistency based on the abstract syntax. In *Symposium on Engineering Interactive Computing Systems (EICS) 2011, Proceedings*, pages 79–84. ACM, 2011. Cited on page 24.

[TLWS18] Stefan Tomaszek, Erhan Leblebici, Lin Wang, and Andy Schürr. Virtual Network Embedding: Reducing the Search Space by Model Transformation Techniques. In *International Conference on Model Transformation (ICMT) 2018, Proceedings*, pages 59–75. Springer, 2018. Cited on pages 216 and 218.

[TM14] Le Minh Sang Tran and Fabio Massacci. An Approach for Decision Support on the Uncertainty in Feature Model Evolution. In *International Requirements Engineering Conference (RE) 2014, Proceedings*, pages 93–102. IEEE, 2014. Cited on page 26.

[Tom21] Stefan Tomaszek. *Modellbasierte Einbettung von virtuellen Netzwerken in Rechenzentren*. PhD thesis, Darmstadt University of Technology, Germany, 2021. Cited on page 218.

[Tra08] Laurence Tratt. A change propagating model transformation Language. *Journal of Object Technolology*, 7(3):107–124, 2008. Cited on pages 124 and 125.

[TWV+19] Tarik Terzimehic, Monika Wenger, Sebastian Voss, Sten Grüner, and Haitham Elfaham. SMT-Based Deployment Calculation in Industrial Automation Domain. In *International Conference on Emerging Technologies and Factory Automation (ETFA) 2019, Proceedings*. IEEE, 2019. Cited on page 218.

[TZY+08] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. ArnetMiner: Extraction and Mining of Academic Social Networks. In *International Conference on Knowledge Discovery and Data Mining (KDD) 2008, Proceedings*, pages 990–998. ACM, 2008. Cited on page 23.

[VAS12] Gergely Varró, Anthony Anjorin, and Andy Schürr. Unification of Compiled and Interpreter-Based Pattern Matching Techniques. In *European Conference on Modelling Foundations and Applications (ECMFA) 2012, Proceedings*, pages 368–383. Springer, 2012. Cited on page 164.

[VBH+16] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling*, 15(3):609–629, 2016. Cited on page 165.

[VBNE15] Jean-Luc Voirin, Stéphane Bonnet, Véronique Normand, and Daniel Exertier. Model-Driven IVV Management with Arcadia and Capella. In *International Conference on Complex Systems Design & Management (CSD&M) 2015, Proceedings*, pages 83–94. Springer, 2015. Cited on page 192.

[VD13] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In *International Conference on Model Transformation (ICMT) 2013, Proceedings*, pages 125–140. Springer, 2013. Cited on page 165.

[VDT18] Vuk Vukovic, Jovica Djurkovic, and Jelica Trninic. A Business Software Testing Process-Based Model Design. *International Journal of Software Engineering and Knowledge Engineering*, 28(5):701–750, 2018. Cited on page 19.

[Ver20] Surbhi Verma. Consistency Management based on Triple Graph Grammars with Graph Constraints. Master thesis, Paderborn University, Germany. 2020. Cited on page 15.

[VGE+10] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. Flexible and scalable consistency checking on product line variability models. In *Automated Software Engineering Conference (ASE) 2010, Proceedings*, pages 63–72. ACM, 2010. Cited on page 31.

[VGH+12] Michael Vierhauser, Paul Grünbacher, Wolfgang Heider, Gerald Holl, and Daniela Lettner. Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2012, Proceedings*, pages 531–545. Springer, 2012. Cited on pages 23, 28, and 31.

[VMO16] Antonio Vallecillo, Carmen Morcillo, and Priscill Orue. Expressing Measurement Uncertainty in Software Models. In *International Conference on the Quality of Information and Communications Technology (QUATIC) 2016, Proceedings*, pages 15–24. IEEE, 2016. Cited on pages 26 and 32.

[VPM90] Mladen A Vouk, Amit M Paradkar, and David F McAllister. Modeling execution time of multi-stage N-version fault-tolerant software. In *International Computer Software and Applications Conference (COMPSAC) 1990, Proceedings*, pages 505–511. IEEE, 1990. Cited on pages 24, 25, and 31.

[WA20] Nils Weidmann and Anthony Anjorin. Schema Compliant Consistency Management via Triple Graph Grammars and Integer Linear Programming. In *Fundamental Approaches to Software Engineering (FASE) 2020, Proceedings*, pages 315–334. Springer, 2020. Cited on pages 16 and 121.

[WA21a] Nils Weidmann and Anthony Anjorin. eMoflon::Neo - Consistency and Model Management with Graph Databases. In *International Workshop on Bidirectional Transformations (Bx) 2021, Proceedings*, pages 54–64. CEUR-WS.org, 2021. Cited on pages 16 and 178.

[WA21b] Nils Weidmann and Anthony Anjorin. Schema Compliant Consistency Management via Triple Graph Grammars and Integer Linear Programming. *Formal Aspects of Computing*, aug 2021. Cited on pages 16, 119, 283, and 286.

[WAC20] Nils Weidmann, Anthony Anjorin, and James Cheney. VICToRy: Visual Interactive Consistency Management in Tolerant Rule-based Systems. In *International Workshop on Graph Computation Models (GCM) 2020, Proceedings*, pages 1–12. EPTCS, 2020. Cited on pages 16 and 194.

[WAF+19] Nils Weidmann, Anthony Anjorin, Lars Fritsche, Gergely Varró, Andy Schürr, and Erhan Leblebici. Incremental Bidirectional Model Transformation with eMoflon: : IBeX. In *International Workshop on Bidirectional Transformations (Bx) 2019, Proceedings*, pages 45–55. CEUR-WS.org, 2019. Cited on page 16.

[Wag95] Annika Wagner. On the Expressive Power of Algebraic Graph Grammars with Application Conditions. In *Theory and Practice of Software Development (TAPSOFT) 1995, Proceedings*, pages 409–423. Springer, 1995. Cited on page 69.

[WALS19] Nils Weidmann, Anthony Anjorin, Erhan Leblebici, and Andy Schürr. Consistency management via a combination of triple graph grammars and linear programming. In *Software Language Engineering (SLE) 2019, Proceedings*, pages 29–41. ACM, 2019. Cited on pages 16 and 97.

[WARV19] Nils Weidmann, Anthony Anjorin, Patrick Robrecht, and Gergely Varró. Incremental (Unidirectional) Model Transformation with eMoflon: : IBeX. In *International Conference on Graph Transformation (ICGT) 2019, Proceedings*, pages 131–140. Springer, 2019. Cited on pages 16 and 180.

[WASK19] Nils Weidmann, Anthony Anjorin, Florian Stolte, and Florian Kraus. From Pattern Invocation Networks to Rule Preconditions. In *International Conference on Graph Transformation (ICGT) 2019, Proceedings*, pages 195–211. Springer, 2019. Cited on page 16.

[WBCW20] Andreas Wortmann, Olivier Barais, Benoît Combemale, and Manuel Wimmer. Modeling languages in Industry 4.0: an extended systematic mapping study. *Software and Systems Modeling*, 19(1):67–94, 2020. Cited on page 19.

[WC09] Chen-Wei Wang and Alessandra Cavarra. Checking Model Consistency Using Data-Flow Testing. In *Asia-Pacific Software Engineering Conference (APSEC) 2009, Proceedings*, pages 414–421. CEUR-WS.org, 2009. Cited on page 24.

[WCB17] Andreas Wortmann, Benoît Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Proceedings*, pages 281–291. IEEE, 2017. Cited on page 19.

[WE21] Nils Weidmann and Gregor Engels. Concurrent model synchronisation with multiple objectives. In *Genetic and Evolutionary Computation Conference (GECCO) 2021, Proceedings*, pages 1097–1105. ACM, 2021. Cited on pages 16 and 154.

[Web17] Jens H Weber. GRAPE – A Graph Rewriting and Persistence Engine. In *International Conference on Graph Transformation (ICGT) 2017, Proceedings*, pages 209–220. Springer, 2017. Cited on pages 166 and 173.

[Wei18] Nils Weidmann. Tolerant Consistency Management in Model-Driven Engineering. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Companion Proceedings*, pages 192–197. ACM, 2018. Cited on page 16.

[WFA20] Nils Weidmann, Lars Fritsche, and Anthony Anjorin. A search-based and fault-tolerant approach to concurrent model synchronisation. In *Software Language Engineering (SLE) 2020, Proceedings*, pages 56–71. ACM, 2020. Cited on pages 16, 117, 141, and 286.

[WGP09] Christopher Wolfe, T C Nicholas Graham, and W Greg Phillips. An Incremental Algorithm for High-Performance Runtime Model Consistency. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS) 2009, Proceedings*, pages 357–371. Springer, 2009. Cited on pages 23 and 33.

[WKA21] Nils Weidmann, Suganya Kannan, and Anthony Anjorin. Tolerance in Model-Driven Engineering: A Systematic Literature Review with Model-Driven Tool Support. *Computing Research Repository*, abs/2106.0, 2021. Cited on page 16.

[WKK+11] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitrios S Kolovos, Richard F Paige, Marius Lauder, and Andy Schürr. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In *International Conference on Model Transformation (ICMT) 2011, Proceedings*, pages 31–46. Springer, 2011. Cited on page 55.

[WKK+12] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitrios S Kolovos, Richard F Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. Surveying Rule Inheritance in Model-to-Model Transformation Languages. *Journal of Object Technology*, 2012. Cited on page 55.

[WMC+20] Sabine Wolny, Alexandra Mazak, Christine Carpella, Verena Geist, and Manuel Wimmer. Thirteen years of SysML: a systematic mapping study. *Software and Systems Modeling*, 19(1):111–169, 2020. Cited on page 19.

[WOR19] Nils Weidmann, Robin Oppermann, and Patrick Robrecht. A feature-based classification of triple graph grammar variants. In *Software Language Engineering (SLE) 2019, Proceedings*, pages 1–14. ACM, 2019. Cited on page 16.

[WRT+13] Martin Walker, Mark-Oliver Reiser, Sara Tucci Piergiovanni, Yiannis Papadopoulos, Henrik Lönn, Chokri Mraidha, David Parker, De-Jiu Chen, and David Servat. Automatic optimisation of system architectures using EAST-ADL. *Journal of Systems and Software*, 86(10):2467–2487, 2013. Cited on page 217.

[WS20] Nils Weidmann and Stefan Sauer. Applying Bidirectional Transformations in Industrial Contexts: Challenges and Solutions. In *Workshop Software-Reengineering und -Evolution (WSRE) 2020, Proceedings*, 2020. Cited on page 16.

[WSA+21] Nils Weidmann, Shubhangi Salunkhe, Anthony Anjorin, Enes Yigitbas, and Gregor Engels. Automating Model Transformations for Railway Systems Engineering. *Journal of Object Technology*, 20(3):10:1–14, 2021. Cited on pages 16 and 208.

[WXH+10] Bo Wang, Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Wei Zhang, and Hong Mei. A Dynamic-Priority Based Approach to Fixing Inconsistent Feature Models. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2010, Proceedings*, pages 181–195. Springer, 2010. Cited on pages 24 and 33.

[XHZ+08] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Song Hui, Hong Mei, Yingfei Xiong, Haiyan Zhao, Zhenjiang Hu, Masato Takeichi, Song Hui, and Hong Mei. Beanbag: Operation-based Synchronization with IntraRelations. In *Grace Technical Reports, GRACE-TR-2008–04*. National Institute of Informatics, 2008. Cited on pages 124 and 125.

[XHZ+09] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. Supporting automatic model inconsistency fixing. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE) 2009, Proceedings*, pages 315–324. ACM, 2009. Cited on pages 23, 25, 28, and 31.

[XSHT09] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. Supporting Parallel Updates with Bidirectional Model Transformations. In *International Conference on Model Transformation (ICMT) 2009, Proceedings*, pages 213–228. Springer, 2009. Cited on pages 124 and 125.

[XSHT13] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. Synchronizing concurrent model updates based on bidirectional transformation. *Software and System Modeling*, 12(1):89–104, 2013. Cited on pages 124 and 126.

[YALG18] Enes Yigitbas, Anthony Anjorin, Erhan Leblebici, and Marvin Grieger. Bidirectional Method Patterns for Language Editor Migration. In *European Conference on Modelling Foundations and Applications (ECMFA) 2018, Proceedings*, pages 97–114. Springer, 2018. Cited on page 165.

[YGWE21] Enes Yigitbas, Simon Gorissen, Nils Weidmann, and Gregor Engels. Collaborative Software Modeling in Virtual Reality. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2021, Proceedings (to appear)*. IEEE, 2021. Cited on page 16.

[YV00] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Symposium on Operating Systems Design and Implementation (OSDI) 2000, Proceedings*, pages 305–318. USENIX, 2000. Cited on pages 25 and 31.

[ZAY+19] Man Zhang, Shaukat Ali, Tao Yue, Roland Norgren, and Oscar Okariz. Uncertainty-Wise Cyber-Physical System test modeling. *Software and Systems Modeling*, 18(2):1379–1418, 2019. Cited on pages 26 and 30.

[ZAYN17] Man Zhang, Shaukat Ali, Tao Yue, and Roland Norgren. Uncertainty-wise evolution of test ready models. *Information and Software Technology*, 87:140–159, 2017. Cited on pages 26 and 30.

[ZCM+16] Athanasios Zolotas, Robert Clarisó, Nicholas Matragkas, Dimitrios S Kolovos, and Richard F Paige. Constraint Programming for Type Inference in Flexible Model-Driven Engineering. *Computer Languages, Systems & Structures*, 49, 2016. Cited on pages 23 and 33.

[ZKC16] Sergey Zverlov, Maged Khalil, and Mayank Chaudhary. Pareto-efficient deployment synthesis for safety-critical applications in seamless model-based development. In *European Congress on Embedded Real Time Software and Systems (ERTS) 2016, Proceedings*, 2016. Cited on page 218.

[ZRH+20] Athanasios Zolotas, Horacio Hoyos Rodriguez, Stuart Hutchesson, Beatriz Sanchez Piña, Alan Grigg, Mole Li, Dimitrios S Kolovos, and Richard F Paige. Bridging proprietary modelling and open-source model management tools: the case of PTC Integrity Modeller and Epsilon. *Software and Systems Modeling*, 19(1):17–38, 2020. Cited on pages 205 and 207.

Appendix A

Example TGGs

In this chapter, two further example TGGs are introduced, which were used to evaluate the hybrid approach. Section A.1 presents the *FamiliesToPersons* example that was used for the experimental evaluations of Sect. 5.8 and 6.6. In Sect. A.2, the TGG *JavaToDoc* is introduced, which served as an example for the experiments of Sect. 6.6, 7.7, and 8.5. For both TGGs, metamodels, rules and constraints are presented.

A.1 FamiliesToPersons

In this section, the example TGG *FamiliesToPersons* is briefly introduced. The used metamodel and rules follow the *FamiliesToPersons* benchmark by Allilaire and Jouault[1], extended by minor modifications as in [ABW17]. Graph constraints, which are not part of the original benchmark, were added to the example in subsequent work [WA21b].

The triple metamodel is depicted in Fig. A.1.[2] The Families model represents family structures with the relations between family members. A `FamilyRegister` contains a set of arbitrary many `Families`, which consists of family members having a role: father, mother, son or daughter. Families have a family `name`, and each family member has their own first `name`. The Persons model has a `PersonRegister` containing arbitrary many `Persons` which are either `Male` or `Female`. `Persons` have their full `name` and their `birthday` as attributes.

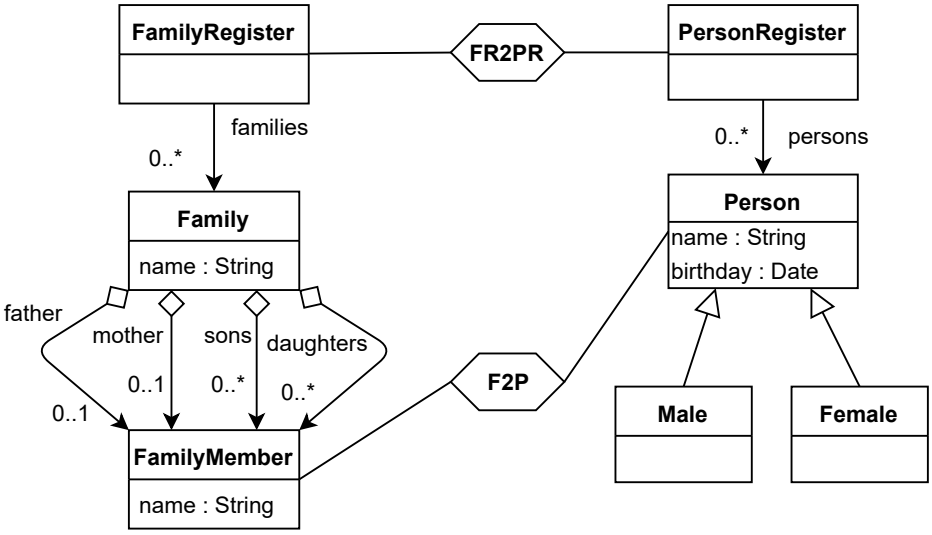


Figure A.1: FamiliesToPersons: Triple metamodel

[1] https://www.eclipse.org/atl/documentation/basicExamples_Patterns/

[2] Of course, these metamodels cannot represent all possible family structures. It was chosen due to its simplicity while being able to show all relevant concepts and its acceptance as a benchmark example.

Figure A.2 shows the (abstract) rules of the TGG. The rule *FamiliesToPersons* creates a family register `fr`, a person register `pr` and links them via a correspondence node. A new family `f` with a first family member `fm` is added to a family register `fr` with the rule *FamilyMemberToPerson*, while a person `p` that corresponds to `fm` is added to the person register `pr`. Thereby, `p`'s name is formed out of the forename of `fm` and the surname of `f`. The rule *MemberOfExistingFamilyToPerson* works the same way, but requires an existing family `f` as context instead of creating it anew.

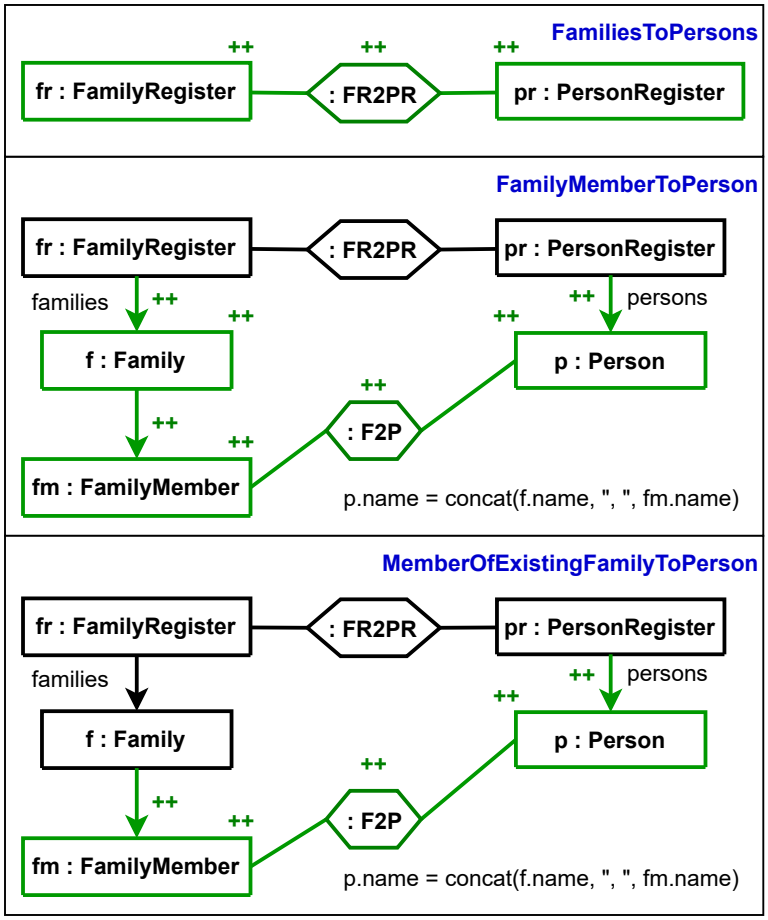


Figure A.2: FamiliesToPersons: TGG rules

The *FamiliesToPersons* example uses the rule refinement feature of TGGs [ASLS14]. Rules with abstract types (e. g., `Person` in the target model) can be refined with concrete types to avoid multiple definitions of structurally equivalent rules. The two abstract rules of Fig. A.2 are refined to concrete rules as stated in Tab. A.1. The rule *MotherToFemale*, e. g., refines *FamilyMemberToPerson* with a `mother` edge pointing from the family `f` to the family member `fm`, and a corresponding female person `p` is added to the target model. Its version without creating a new family `f` refines the rule *MemberOfExistingFamilyToPerson*, respectively.

To evaluate the effects of integrating graph constraints on the scalability of the hybrid approach (cf. Chap. B), the example was enriched with two negative constraints and one implication constraint as shown in Fig. A.3. The negative constraints *NoTwoFathers* and *NoTwoMothers* forbid the existence of two family members with the roles father and mother, respectively. These constraints guarantee that the upper bounds of the respective associations in the source metamodel (cf. Fig. A.1) are respected.

FamiliesToPersons	Family → FamilyMember	Person
MotherToFemale	mother	Female
MotherOfExistingFamilyToFemale	mother	Female
FatherToMale	father	Male
FatherOfExistingFamilyToMale	father	Male
DaughterToFemale	daughters	Female
DaughterOfExistingFamilyToFemale	daughters	Female
SonToMale	sons	Male
SonOfExistingFamilyToMale	sons	Male

Table A.1: FamiliesToPersons: Rule refinement

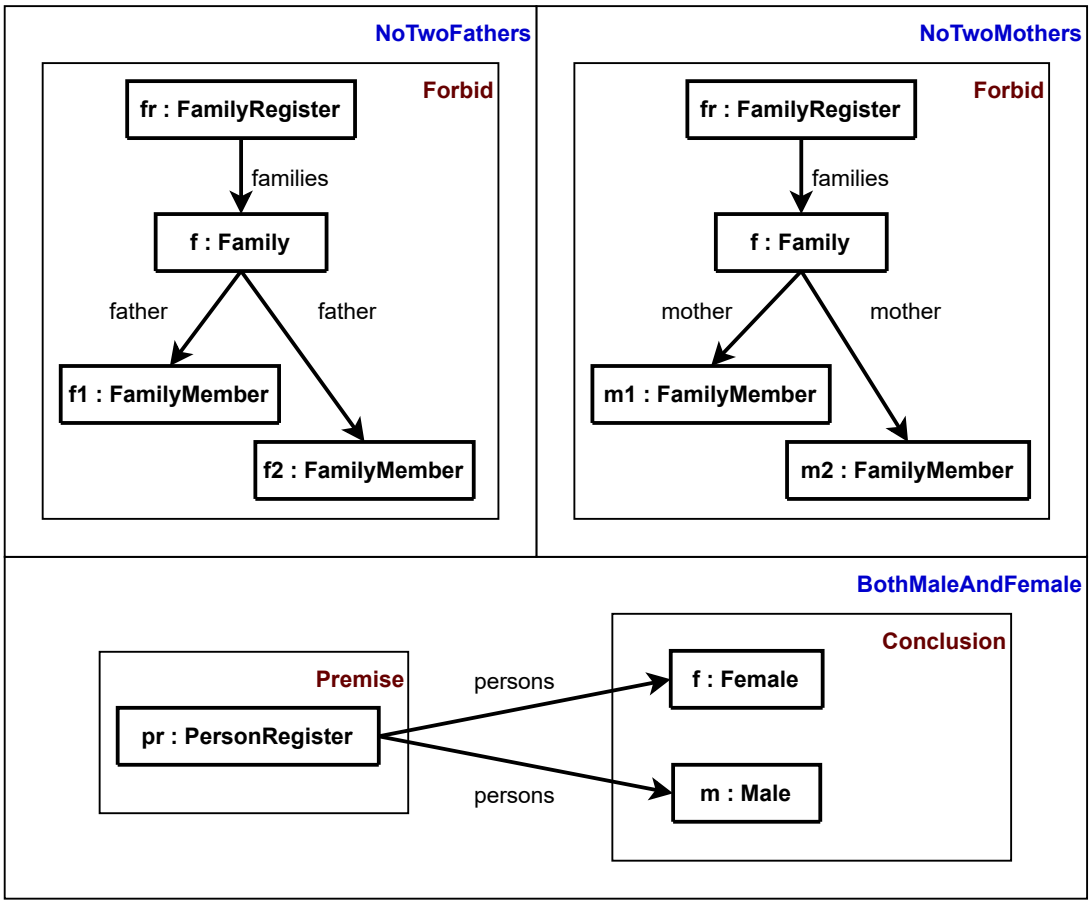


Figure A.3: FamiliesToPersons: Graph constraints

A.2 JavaToDoc

The second TGGs formalises a consistency relation between (simplified) abstract syntax trees describing Java code (source model) and its documentation (target model). The metamodels and rules of *JavaToDoc* originate from prior work by Fritsche et al. [FKM+20] as well as Weidmann et al. [WFA20], graph constraints were added subsequently [WA21b].

The triple metamodel is depicted in Fig. A.4. As the root of the Java metamodel, `Classes` can form an inheritance hierarchy. Each class can have arbitrarily many `Methods` and `Fields`, and each `Method` can have a set of `Parameters`. The documentation metamodel, in contrast, consists of `Documents` that can reference each other via hyper-references. A document is structured as a set of `Entries`. To provide an overview of important terms, a `Glossary` is contained in the documentation model. It consists of `GlossaryEntries`, which are referred to from documentation entries. The correspondence model is structured as follows: Classes are associated with documents, while methods, their parameters, and fields are represented as entries in the respective document. The glossary and its entries do not have a corresponding structural element in the source model, therefore they are not linked to a node of the correspondence model.

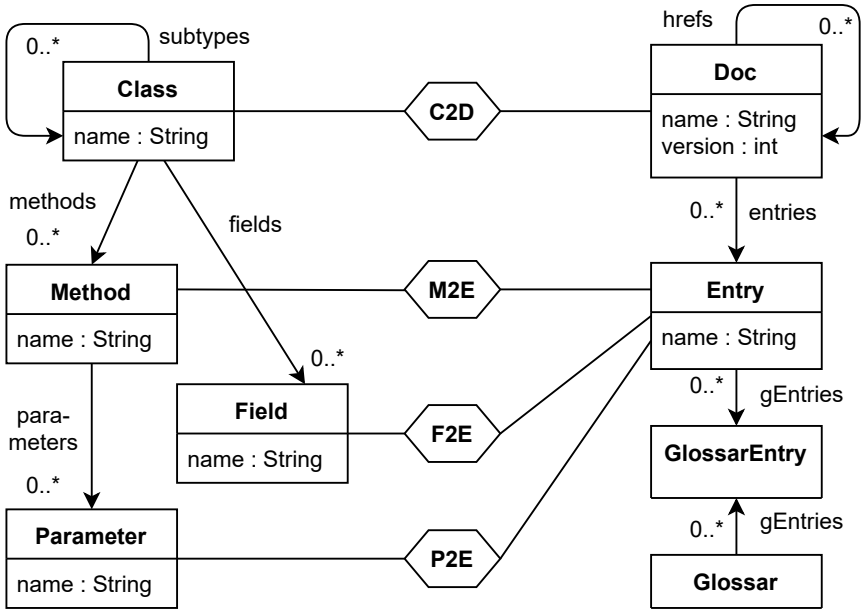


Figure A.4: JavaToDoc: Triple metamodels

The rules *ClassToDoc* and *AddGlossary* are the axioms of the *JavaToDoc* TGG. While the latter only creates a glossary `g` in the target model, the former creates a class `c` and a document `d` that correspond to each other. The rule *SubClassToDoc* adds a subclass `sc` to an existing class `c` and links it to a document `sd` that is referenced from the document `d` corresponding to `c`. The creation of the rule together with the inheritance links ensures that no multiple inheritance relations can be created as Java does not support them. Methods (`m`) and fields (`f`) are added to a class `c` by the rules *MethodToEntry* and *FieldToEntry*, respectively, while corresponding entries (`e`) are added to the document `d` which belongs to `c`. *AddParameter* creates a new parameter `p` for a method `m`, whereby the correct entry `e` in the documentation model is identified by the existing correspondence link between `m` and `e`. Affecting only the documentation model, the rules *AddGlossaryEntry*

and *LinkGlossaryEntry* add entries (`ge`) to a glossary `g` and link them to a document entry `e`, respectively.

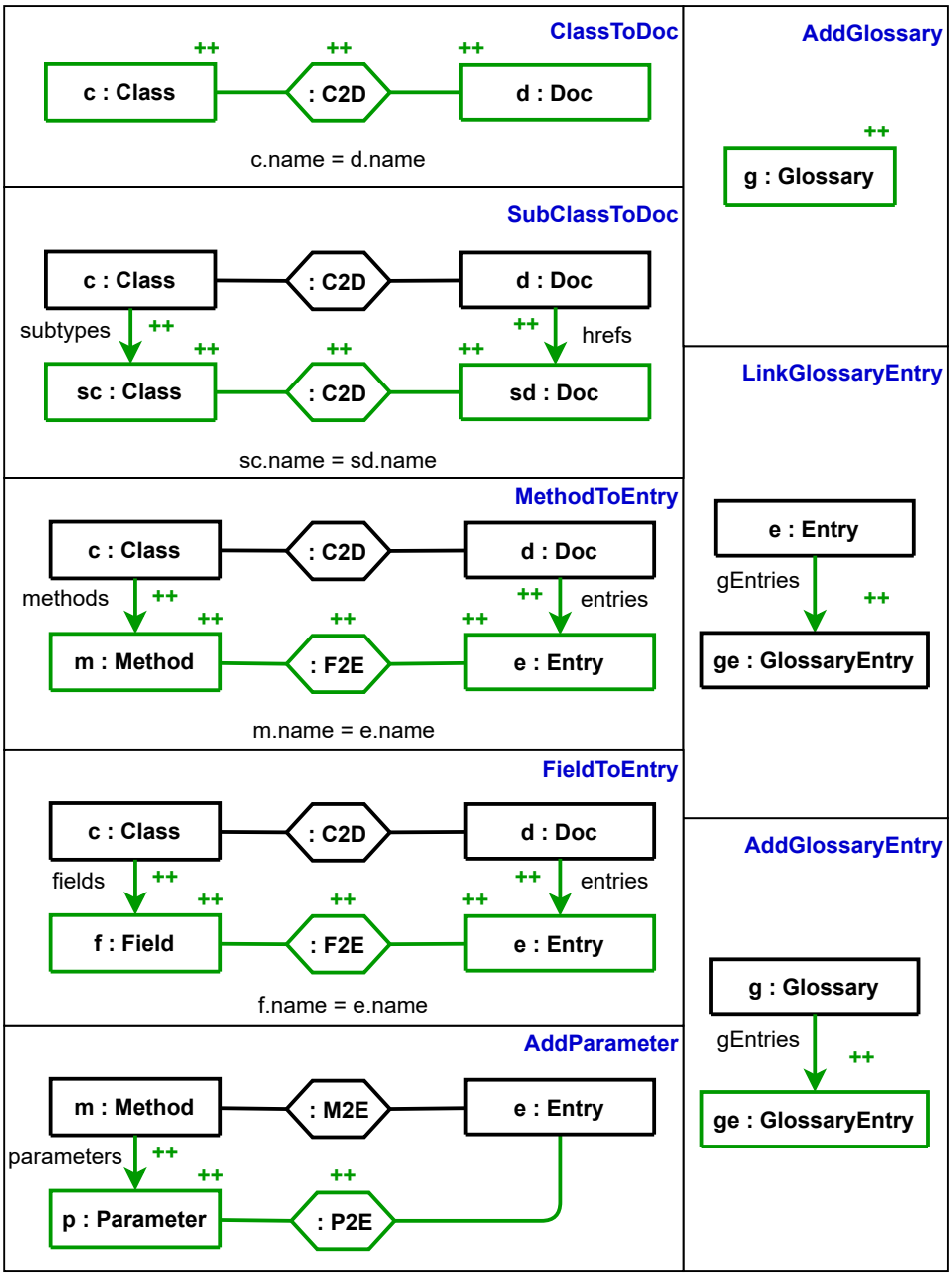


Figure A.5: JavaToDoc: TGG rules

Via two negative constraints and two implication constraints (Fig. A.6), we restrict the set of consistent triples for the *JavaToDoc* TGG. We *forbid* that there are two or more glossaries (`g1`, `g2`) in the documentation model with the constraint *NoTwoGlossaries*. Creating multiple links from an entry `e` to a glossary entry `ge` is forbidden by the constraint *NoDoubleLink*.

In the Java model, we enforce every class `c` to be non-empty. To specify this, we use an *implication* constraint expressing that each class `c` (premise) must be connected to either a method `m` (Conclusion 1) or a field `f` (Conclusion 2), forming the constraint *NoEmptyClass*.

Implication constraints can also be more complex and affect multiple models. With a fourth constraint *SameNameSameGlossaryEntry*, we ensure that methods `m1` and `m2` of the same class `c` with the same name (overloaded methods), correspond to entries `e1` and `e2` in the documentation model that point to a common glossary entry `ge`.

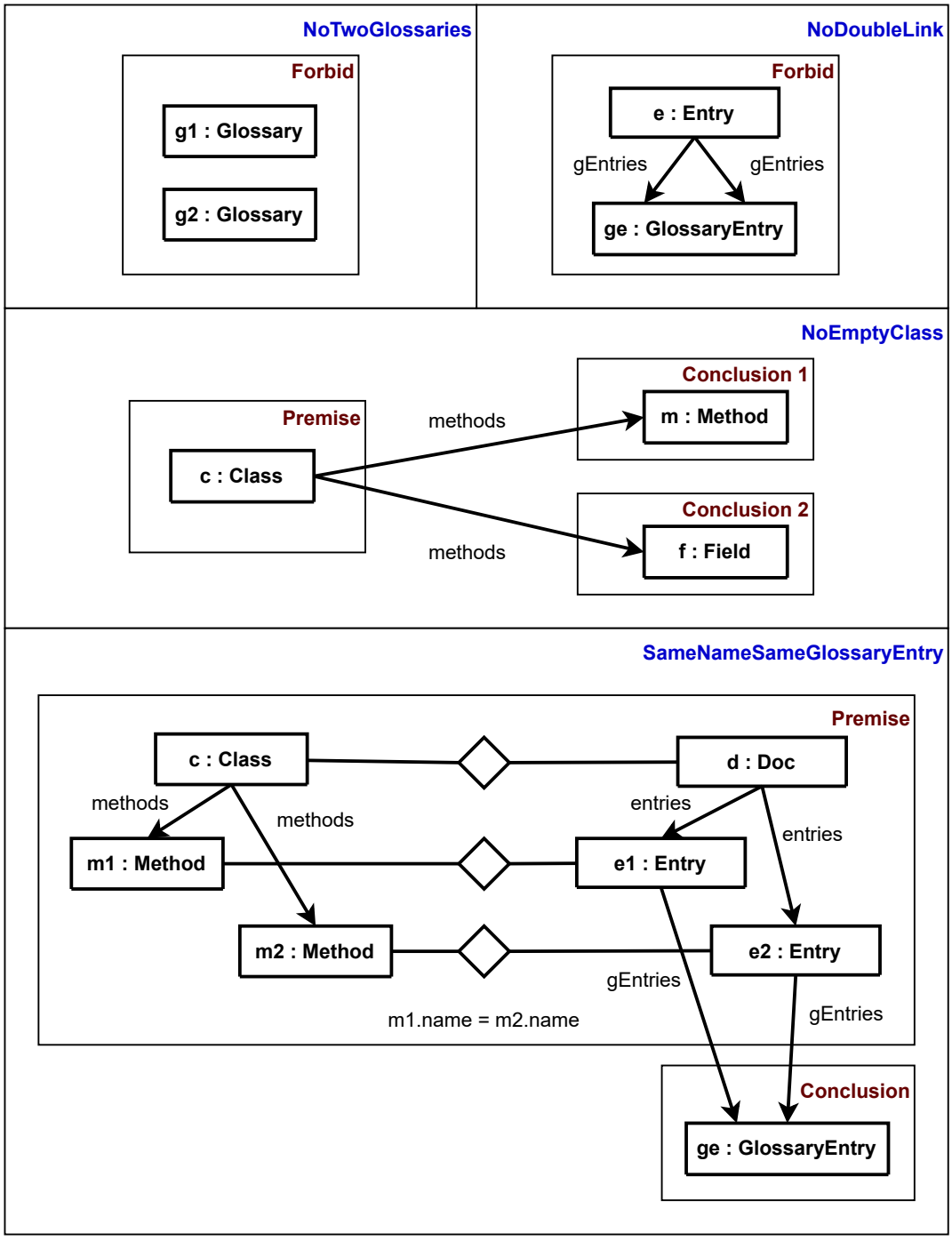


Figure A.6: JavaToDoc: Graph constraints

Appendix B

Runtime Measurements

As an extension of the evaluation results described in Sect. 6.6, this chapter presents the detailed runtime measurements for the example TGGs *FamiliesToPersons* (Sect. A.1) and *JavaToDoc* (Sect. A.2) in settings with and without graph constraints. In Fig. B.1 - B.24, the detailed performance measurements for the different operations, TGGs, and constraint types are presented.

In contrast to the plots of Sect. 6.6, distinct time measurements for the phases "ILP Generation", "Pattern Matching", "Rule Application", and "ILP Solving" of the hybrid consistency management work-flow are shown. In the following, *FamiliesToPersons* is abbreviated with *F2P*, while *J2D* stands for *JavaToDoc*.

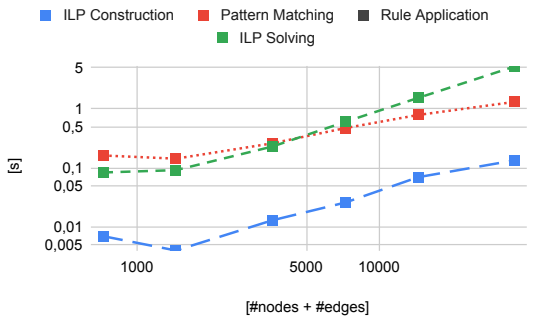


Figure B.1: F2P: CO without constraints

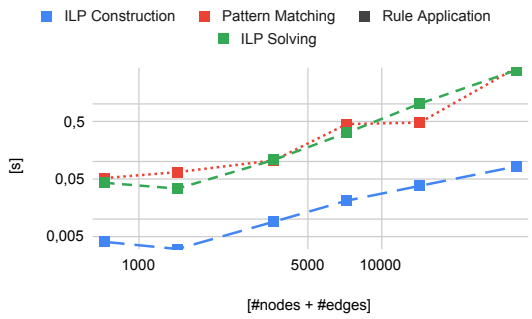


Figure B.2: J2D: CO without constraints

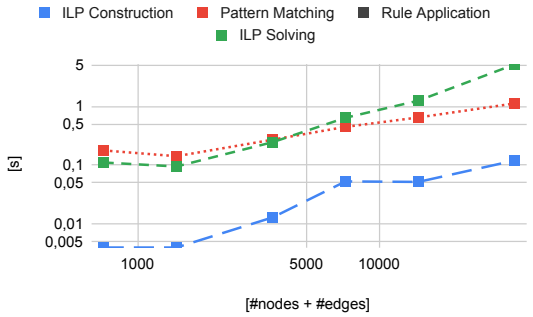


Figure B.3: F2P: CO with negative constraints

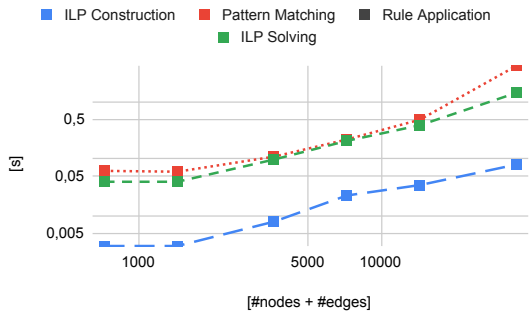


Figure B.4: J2D: CO with negative constraints

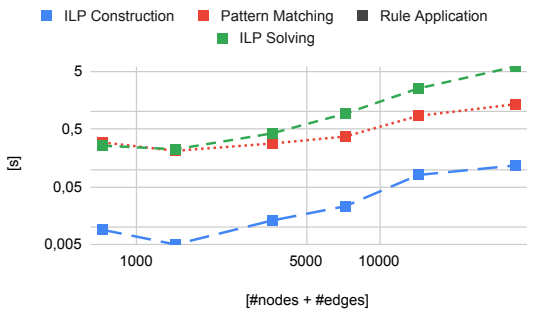


Figure B.5: F2P: CO with implications constraints

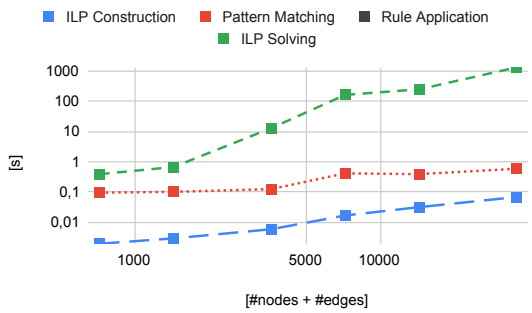


Figure B.6: J2D: CO with implications constraints

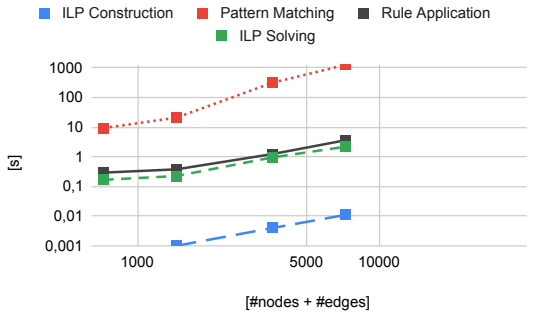


Figure B.7: F2P: CC without constraints

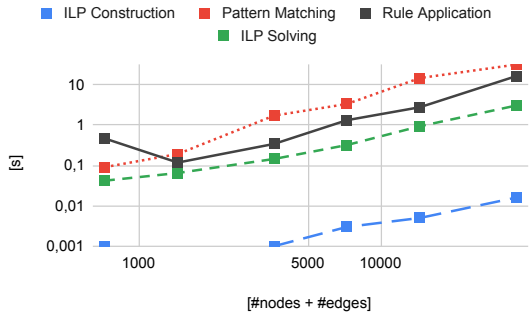


Figure B.8: J2D: CC without constraints

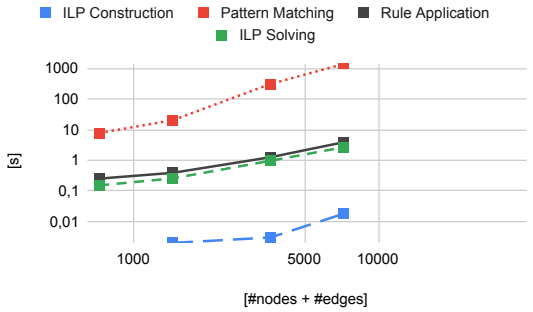


Figure B.9: F2P: CC with negative constraints

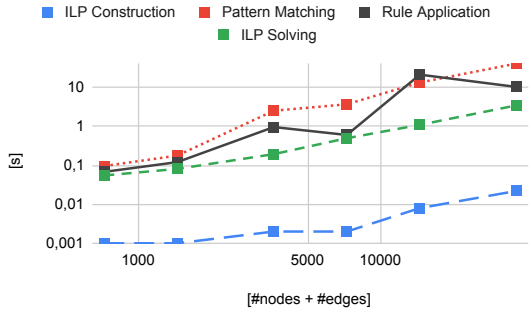


Figure B.10: J2D: CC with negative constraints

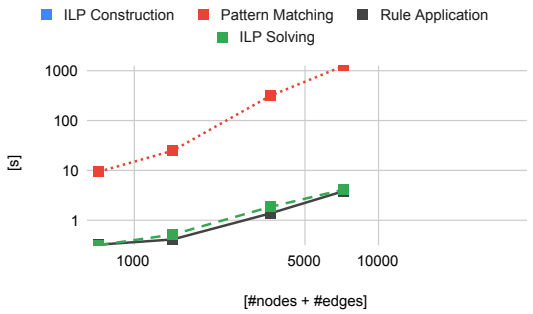


Figure B.11: F2P: CC with implications constraints

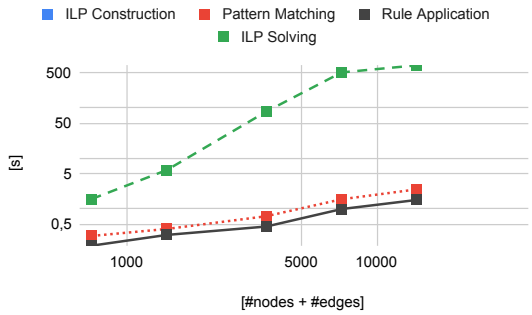


Figure B.12: J2D: CC with implications constraints

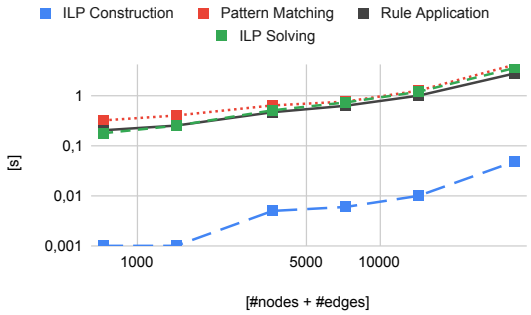


Figure B.13: F2P: FWD_OPT without constraints

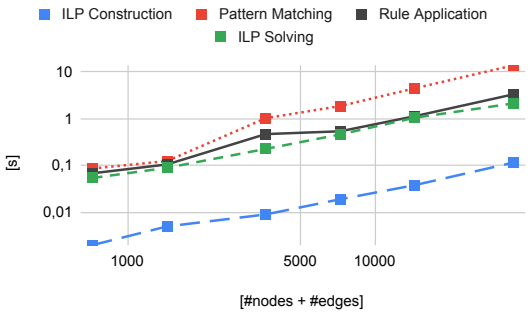


Figure B.14: J2D: FWD_OPT without constraints

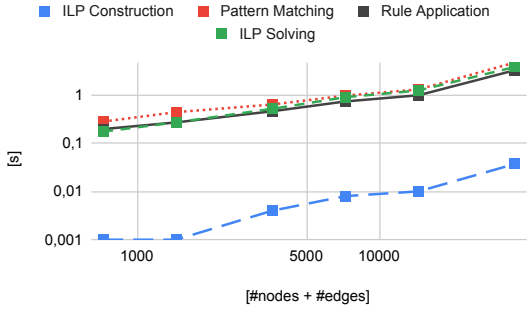


Figure B.15: F2P: FWD_OPT with negative constraints

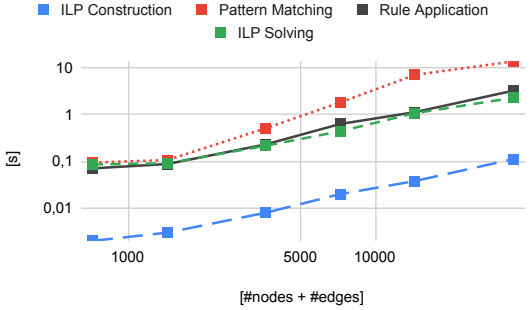


Figure B.16: J2D: FWD_OPT with negative constraints

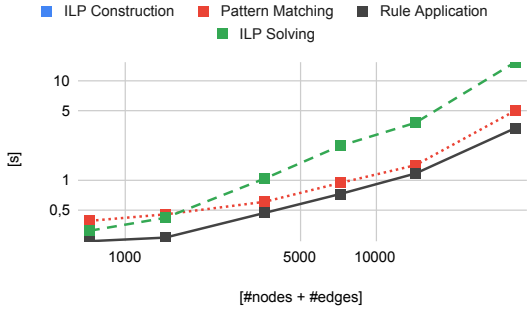


Figure B.17: F2P: FWD_OPT with implications constraints

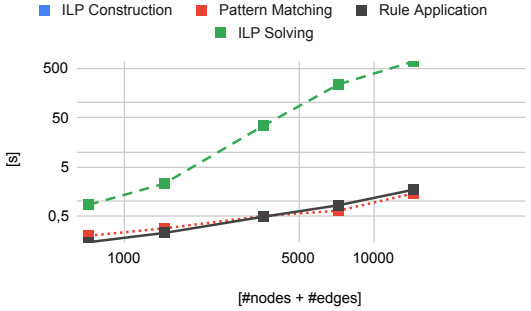


Figure B.18: J2D: FWD_OPT with implications constraints

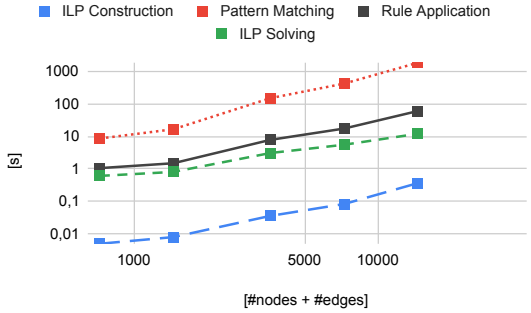


Figure B.19: F2P: BWD_OPT without constraints

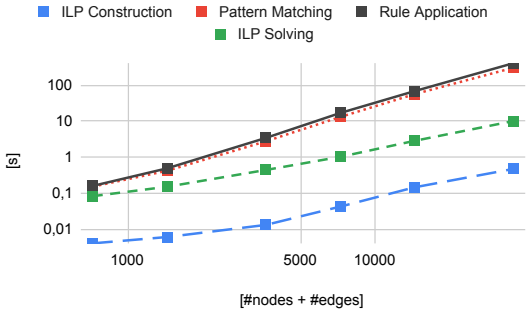


Figure B.20: J2D: BWD_OPT without constraints

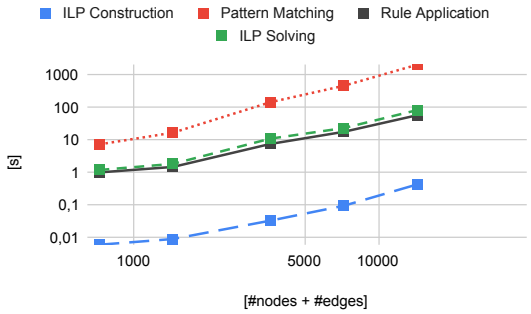


Figure B.21: F2P: BWD_OPT with negative constraints

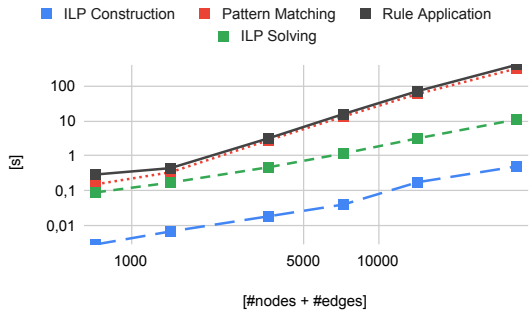


Figure B.22: J2D: BWD_OPT with negative constraints

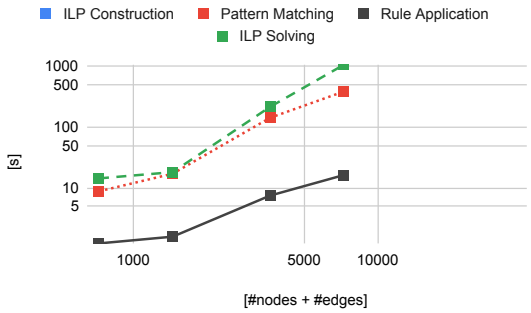


Figure B.23: F2P: BWD_OPT with implications constraints

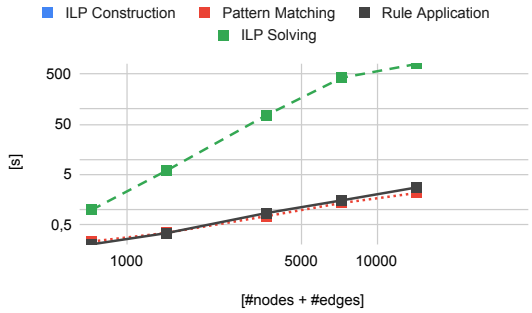


Figure B.24: J2D: BWD_OPT with implications constraints