Jan Dominik Rieke

Model Consistency Management for Systems Engineering

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar

ISSN 2195-5239

Satz und Gestaltung: Jan Dominik Rieke

Hersteller: Verlagshaus Monsenstein und Vannerdat OHG
Druck · Buch · Verlag
Münster

Printed in Germany

Preface

Today's technical systems, from home appliances to transportation means, exhibit innovative functionality and provide only comfort for the user, if they consist of a smart configuration of hardware and software components. As a consequence, the necessary engineering process has become very complex, because it requires a close interaction between the different involved disciplines, in particular mechanical, electrical and software engineering.

Traditionally and even today, these disciplines and their corresponding engineering processes and models are not really aligned to each other. Rather, the so-called "throw-it-over-the-wall" approach is the dominant guiding principle for many of these processes. "Throw-it-over-the-wall" means that usually a CAD-model is designed first by the mechanical engineers, then given to the electrical engineers to design the wirings, sensors, actuators etc. and, finally, the software engineers have to build the software "on top" what is given to them. Of course, this picture is a vast simplification, because many more models exist which have to be aligned to each other. Still this picture illustrates the two major weaknesses, namely (1) that the engineers of one discipline have to live with built in (resource) constraints as defined by the engineers of another discipline who could not oversee all consequences of their design and (2) that a tedious and error-prone manual process is often in place to redesign the different models if such constraints are detected during the engineering process. s

The core of Jan Rieke's approach is a method for an automatic model-to-model transformation which is combined with smart means for guaranteeing model consistency. In principle, the approach guaranties automatic updates of all models across all disciplines of an engineering process, when one model is changed by an engineer. However, such updates should not happen always immediately after a change. These changes are often very fine-grained ones and may even be reconsidered later on. Immediate updates would corrupt the whole process and would not really support a rigorous and controllable engineering process. As a particular advantage compared with other similar approaches, Jan Rieke's approach provides and integrates various ideas to enable so-called continuous engineering, i.e. the concurrent construction of different models. This requires to include version- and configuration management of models as well as provisions to delay updates. Delaying updates means to define appropriate milestones when updates are performed and appropriate processes to reestablish model consistency. As a result, no unexpected and hampering interferences with the work of individual engineers should happen during the engineering process.

In summary, the approach provides significant advantages in comparison with others, especially concerning model consistency and synchronization management. It represents a significant step towards making systems engineering, i.e. the close cooperation of different engineering disciplines, applicable in industry. Considering the current hype about cyber physical systems and Industry 4.0, such an approach comes exactly at the right time and marks a cornerstone in providing sophisticated automatic support for highly intertwined and aligned engineering processes. The thesis' results will surely boost the quality of resulting products and the cost-effectiveness of its development and engineering processes.

Paderborn, January 2015 *Prof. Dr. Wilhelm Schäfer*

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Heinz Nixdorf Institute
Software Engineering Group
Zukunftsmeile 1
33102 Paderborn

Model Consistency Management for Systems Engineering

by

Jan Dominik Rieke

`jan.rieke@uni-paderborn.de`

PhD Thesis

in partial fulfilment of the requirements for the degree of
doctor rerum naturalium (Dr. rer. nat.)

supervised by

Prof. Dr. Wilhelm Schäfer

Paderborn, January 23, 2015

Abstract

The development of complex mechatronic systems requires the close collaboration of different disciplines, like mechanical engineering, electrical engineering, control engineering, and software engineering. To tackle the complexity of such systems, such a development is heavily based on models. Engineers use several models on different abstraction levels, for different purposes and with different view-points. Usually, a discipline-spanning system model is developed during the first, interdisciplinary system design phase. For the implementation phase, the disciplines use different models and tools to develop the discipline-specific aspects of the system.

During such a model-based development, inconsistencies between the different discipline-specific models and the discipline-spanning system model are likely to occur, because changes to discipline-specific models may affect the discipline-spanning system model and models of other disciplines. These inconsistencies lead to increased development time and costs if they remain unresolved.

Model transformation and synchronization are promising techniques to detect and resolve such inconsistencies. However, existing model synchronization solutions are not powerful enough to support the complex consistency relations of such an application scenario. In this thesis, we present an novel model synchronization technique that allows for synchronized models with multiple views and abstraction levels. To minimize the information loss and improve automation during the synchronization, it employs metrics to encode expert knowledge. The approach can be customized to allow different amounts of user interaction, from full automation to fine-grained manual decisions.

Zusammenfassung

Die Entwicklung komplexer mechatronischer Systeme erfordert die Zusammenarbeit verschiedenster Fachdisziplinen wie beispielsweise Mechanik, Elektronik/Elektrotechnik, Regelungstechnik und Softwaretechnik. Um der Komplexität solcher Systeme während der Entwicklung Herr zu werden, findet die Entwicklung heutzutage meist modellbasiert statt. Dabei existieren zahlreiche verschiedene Modelle, die jeweils verschiedenen Zwecken dienen, unterschiedliche Gesichtspunkte berücksichtigen und sich auf verschiedenen Abstraktionsebenen befinden. Typischerweise wird zunächst ein fachdiziplinübergreifendes Systemmodell in einer ersten, interdiziplinären Entwicklungsphase erstellt. In der darauf folgenden Implementierungsphase verwenden die Fachdisziplinen jeweils eigene Modelle und Werkzeuge, um ihre disziplinspezifischen Aspekte des Systems zu entwickeln.

Während dieser Implementierungsphase kommt es häufig zu Inkonsistenzen zwischen den fachdisziplinspezifischen Modellen und dem disziplinübergreifenden Systemmodell, da sich Änderungen an den Fachdisziplinmodellen auch auf das Systemmodell und andere Disziplinen auswirken können. Wenn solche Inkonsistenzen ungelöst bleiben, führt dies zu einer verlängerten Entwicklungszeit und steigenden Kosten.

Modelltransformations- und -synchronisationstechniken sind ein vielversprechender Ansatz, um solche Inkonsistenzen zwischen Modellen zu erkennen und aufzulösen. Existierende Modellsynchronisationstechniken sind allerdings nicht mächtig genug, um die komplexen Beziehungen in so einem Entwicklungsszenario zu unterstützen. In dieser Arbeit wird eine neue Modellsynchronisationstechnik präsentiert, die es erlaubt, Modelle verschiedener Sichten und Abstraktionsebenen zu synchronisieren. Dabei werden Metriken zur Erhöhung des Automatisierungsgrads eingesetzt, die Expertenwissen abbilden. Der Ansatz erlaubt unterschiedliche Grade an Benutzerinteraktion, von vollautomatischer Funktionsweise bis zu feingranularen manuellen Entscheidungen.

Acknowledgments

This work would not have been possible without the tremendous support from several people and the great working atmosphere at the software engineering research group at the University of Paderborn/Heinz Nixdorf Institute.

I especially thank Wilhelm Schäfer for the guidance throughout the years at his research group and the confidence he had in me. Further thanks go to Gregor Engels, Jürgen Gausemeier, and Steffen Becker, who significantly helped me with their support and their views on my research from different perspectives.

I also thank all of my former colleagues at the software engineering research group, especially Joel Greenyer, Oliver Sudmann, Claudia Priesterjahn, Markus von Detten, Dietrich Travkin, Jens Frieben, Matthias Meyer, Christopher Brink, Christian Heinzemann, Stefan Dziwok, Uwe Pohlmann, Anas Anis, Christian Stritzke, and Sebastian Lehrig. Besides the inspiring scientific discussions, I very much enjoyed the atmosphere and the humor at the research group. Very special thanks go to "administrative" section, Jutta Haupt, Jürgen "Sammy" Maniera, Astrid Canisius, and Eckhard Steffen. You always knew what to do when technical or administrative problems arose. All of you made the time at Paderborn a very special experience.

My work highly benefited from the interdisciplinary work with researchers from the SFB 614 and particularly those from the group of Prof. Gausemeier. I thank Sascha Kahl, Sebastian Pook, Mareen Vaßholz, Rafał Dorociak, Roman Dumitrescu, and Harald Anacker for many interesting discussions.

It was fun working together with Sebastian Goschin, Philipp Ackermann, Andrey Pines, and Thomas Zolynski during their bachelor and master thesis and in the Project Group SafeBots II.

Also, Florian Schoppmann helped a lot with proofreading.

Most importantly, I thank my family for their continuous support: My parents Klara and Heribert and my brother Matthes. But first and foremost, I thank my beloved wife Doreen (besides her helpful design advices) for her constant encouragement, her patience, and her extraordinary support despite all the things she had endure during the last years.

Contents

List of Publications

[ADF⁺14] H. Anacker, M. Dellnitz, K. Flaßkamp, S. Groesbrink, P. Hartmann, C. Heinzemann, C. Horenkamp, B. Kleinjohann, L. Kleinjohann, S. Korf, M. Krüger, W. Müller, S. Ober-Blöbaum, S. Oberthür, M. Porrmann, C. Priesterjahn, R. Radkowski, C. Rasche, J. Rieke, M. Ringkamp, K. Stahl, D. Steenken, J. Stöcklein, R. Timmermann, A. Trächtler, K. Witting, T. Xie, and S. Ziegert. Methods for the design and development. In J. Gausemeier, F. J. Rammig, and W. Schäfer (editors), *Design Methodology for Intelligent Technical Systems*, Lecture Notes in Mechanical Engineering, pp. 183–350. Springer Berlin Heidelberg, 2014.

[BvDHR11] S. Becker, M. von Detten, C. Heinzemann, and J. Rieke. Structuring complex story diagrams by polymorphic calls. Tech. Rep. tr-ri-11-323, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Mar. 2011.

[GKRT08] J. Greenyer, E. Kindler, J. Rieke, and O. Travkin. TGGs for transforming UML to CSP: Contribution to the AGTIVE 2007 graph transformation tools contest. Tech. Rep. tr-ri-08-287, Software Engineering Group, Department of Computer Science, University of Paderborn, 2008.

[GPR11] J. Greenyer, S. Pook, and J. Rieke. Preventing information loss in incremental model synchronization by reusing elements. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011)*. 2011.

[GR12] J. Greenyer and J. Rieke. Applying advanced TGG concepts for a complex transformation of sequence diagram specifications to timed game automata. In A. Schürr, D. Varró, and G. Varró (editors), *Applications of Graph Transformations with Industrial Relevance*, vol. 7233 of *Lecture Notes in Computer Science*, pp. 222–237. Springer Berlin Heidelberg, 2012.

[GRSS11] J. Greenyer, J. Rieke, W. Schäfer, and O. Sudmann. The Mechatronic UML development process. In P. L. Tarr and A. L. Wolf (editors), *Engineering of Software*, pp. 311–322. Springer Berlin Heidelberg, 2011.

[GSG+09] J. Gausemeier, W. Schäfer, J. Greenyer, S. Kahl, S. Pook, and J. Rieke. Management of cross-domain model consistency during the development of advanced mechatronic systems. In M. N. Bergendahl, M. Grimheden, and L. Leifer (editors), *Proceedings of the 17th International Conference on Engineering Design (ICED'09)*, vol. 6. 2009.

[GST14] J. Gausemeier, W. Schäfer, and A. Trächtler (editors). *Semantische Technologien im Entwurf mechatronischer Systeme – Effektiver Austausch von Lösungswissen in Branchenwertschöpfungsketten.* Carl Hanser Verlag, München, 2014.

[HLG+13] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin, and A. Schürr. A survey of triple graph grammar tools. *EC-EASST*, Post-Proceedings of the Second International Workshop on Bidirectional Transformations (BX 2013), 2013.

[HPR+12] C. Heinzemann, U. Pohlmann, J. Rieke, W. Schäfer, O. Sudmann, and M. Tichy. Generating Simulink and Stateflow models from software specifications. In *Proceedings of the 12h International Design Conference DESIGN 2012*. 2012.

[HRB+14] C. Heinzemann, J. Rieke, J. Bröggelwirth, A. Pines, and A. Volk. Translating MechatronicUML models to MATLAB/Simulink and Stateflow. Tech. Rep. tr-ri-14-338, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, May 2014. Version 0.4.

[HRS13] C. Heinzemann, J. Rieke, and W. Schäfer. Simulating self-adaptive component-based systems using MATLAB/Simulink. In *Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO '13)*, pp. 71–80. IEEE Computer Society Press, Sep. 2013.

[LAS+14] E. Leblebici, A. Anjorin, A. Schürr, S. Hildebrandt, J. Rieke, and J. Greenyer. A comparison of incremental triple graph grammar tools. In *13th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014)*. 2014.

[RDS+12] J. Rieke, R. Dorociak, O. Sudmann, J. Gausemeier, and W. Schäfer. Management of cross-domain model consistency for behavior models of mechatronic systems. In *Proceedings of the International Design Conference – DESIGN 2012*. 2012.

[Rie11] J. Rieke. Model synchronization for mechatronic systems. In *Software Engineering (Workshops)*, pp. 309–314. 2011.

[RS12] J. Rieke and O. Sudmann. Specifying refinement relations in vertical model transformations. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*. Springer Berlin/Heidelberg, 2012.

[SEH+10] W. Schäfer, T. Eckardt, C. Henke, L. Kaiser, T. Kerstan, J. Rieke, and M. Tichy. Der Softwareentwurf im Entwicklungsprozess mechatronischer Systeme. In *7. Paderborner Workshop Entwurf mechatronischer Systeme*. 2010.

[vDHP+12] M. von Detten, C. Heinzemann, M. C. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. Story diagrams – syntax and semantics. Tech. Rep. tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2012. Version 0.2.

[vDRH+11] M. von Detten, J. Rieke, C. Heinzemann, D. Travkin, and M. Lauder. A new meta-model for story diagrams. In *Proceedings of the 8th International Fujaba Days*. University of Tartu, Estonia, May 2011.

CHAPTER 1

Introduction

From home appliances to transportation means, modern technical systems are becoming more and more complex. In addition, they incorporate an increasing amount of software. Software plays a key role especially with large networks of interconnected systems (so-called “systems of systems”). Such advanced technical systems are often aptly described by the term *mechatronics*. Mechatronic system development is characterized by the collaboration of several disciplines, e.g., mechanical engineering, electrical engineering, and control and software engineering.

The increasing complexity also poses challenges to the engineering process itself. Up to now, the development of the discipline-specific aspects of the system is often not thoroughly integrated into the overall system development processes, or no interdisciplinary developement processes exist at all. For instance, the primary focus in the early phases of the development often is on the mechanical engineering, and software and control engineering only play a minor role. Typically, mechanical engineers design the conceptual and mechanical aspects of a system first. After finalizing this system design, they pass it over to the control and software engineers[1], who are consequently unable to change the conceptual design of the system. For instance, the mechanical engineers may have expected that a certain system function is implemented by software; however, to do so, the software requires additional sensors that have not been integrated into the mechanical system design. This easily leads to increased production time and costs, as problems related to the conceptual design of the system are identified late, and additional iterations in the development process are necessary. Therefore, mechatronic system design requires a more integrated development process.

[1]Note that in this thesis we distinguish between control engineering and software engineering, although both disciplines mainly develop software in a broad sense. Control engineering is responsible for the continuous behavior of the system, i.e., system dynamics and controller design. Software engineering deals with the discrete behavior, i.e., state-based behavior and message-based communication between different system elements.

In particular, the effort of building complex technical systems requires a view on the *system as a whole*, where every aspect of the system under development is considered, and all these aspects are integrated. This is called *systems engineering*. Systems engineering focuses both on the system under development and the corresponding development project: It “integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation.” [INCO04]

Several design guidelines exist that target different types of systems. For mechatronic systems, the VDI 2206 [VDI2206] or the V-Model [Ben05, BRD06]) propose that the development of such systems should take place in three phases. Figure 1.1 shows the *V-Model*, a typical process model for the development of technical systems [Ben05, BRD06].

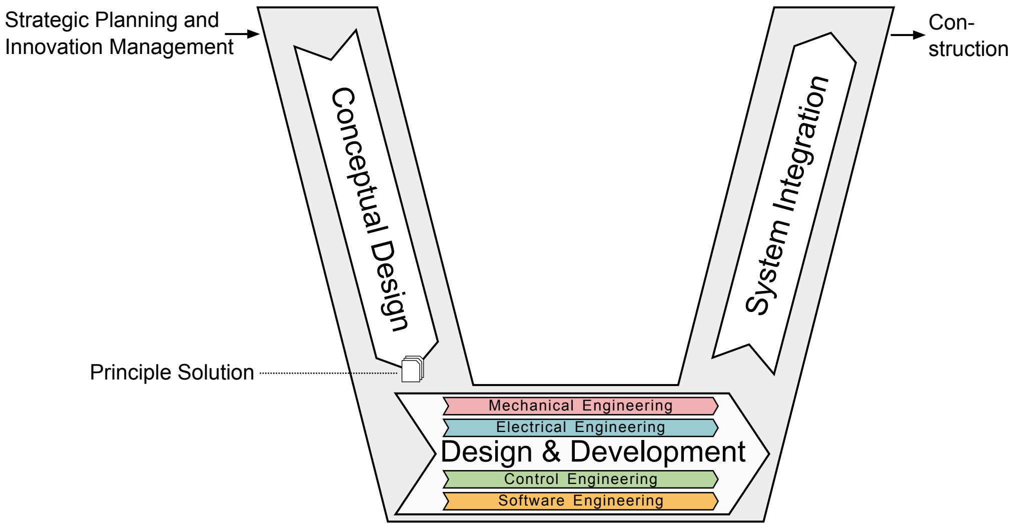


Figure 1.1: V-Model as a process for mechatronic system development (adapted from [VDI2206, BRD06])

After performing the initial strategic product planning, experts from all disciplines collaborate in a first development phase, called the *conceptual design*. In this phase, they work out the *principle solution*, a system model that captures all interdisciplinary concerns. Discipline-specific details, however, are usually not contained in that system model.

Next, the principle solution serves as a basis for the second phase, the discipline-specific *design and development* phase. Here, the engineers from the different disciplines use their own models, artifacts, and tools to design and implement all discipline-specific details of the system. The general idea is that tasks previously performed consecutively are now executed simultaneously by collaborating engineers of the different disciplines; this idea is known as “concurrent engineering” [FGYO95, MCT08].

Finally, in the *system integration* phase, all discipline-specific artifacts are combined to form a holistic model of the system, which, for instance, is used for model-based testing and simulation. Later, this model is also the basis for the actual construction of the system.

The V-Model itself is just the organizational framework for the development. It therefore has to be individually detailed for the actual system under development. In particular, it only addresses the development process, but not which types of development documents ("artifacts") are used, how they are used, and how they influence each other. Thus, the development methodology elaborated in the Collaborative Research Center (CRC) 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering" expands and details existing methodologies with focus on self-optimizing mechatronic systems. This methodology supports engineers and developers by providing specific methods and tools (cf. Gausemeier et al. [GKP+14]). One particular part of this methodology is the *collaboration and coordination of the different disciplines.* On a more technical level, we have to ensure the *consistency of the different development artifacts* used throughout the development process.

Ideally, the principle solution, which is the result of the conceptual design phase, covers all discipline-spanning concerns. This means that all interfaces between the disciplines have been finally defined. Thus, there should be no need for further discipline-spanning coordination during the discipline-specific refinement phase. However, in practice, interdisciplinary concerns that the engineers have not foreseen frequently arise during the conceptual design. For instance, engineers may identify new interrelations between the disciplines during the development. Additionally, changes to the system design may become necessary, e.g., due to changing requirements.

For example, it can turn out that a designated sensor has a failure rate that is too high to allow a safe operation of the system. This requires either selecting another sensor type, adding a second redundant sensor, or implementing new safety procedures – each of which has an impact on several disciplines. Even relatively small modifications may have a severe impact on other disciplines. For instance, only slightly increasing the total weight of a vehicle in mechanical engineering may require new braking strategies in control engineering, together with increased power consumption of the braking system, which affects electrical engineering. These problems have long been recognized (cf., Fohn et al. [FGYO95]). Ma et al. state that, in concurrent engineering, it is "not easy to maintain the consistency among related product models because information associations are not established."[MCT08]

To sum up, changes that affect several disciplines are likely to arise during the discipline-specific design and development phase. This requires communicating these changes between the disciplines, to assure that all discipline's engineers still develop *the same system.* Otherwise, severe issues will arise during the final system integration phase, as there will be contradictions when bringing together the development artifacts of the different disciplines. Not surprisingly, a recent study with 32 engineering companies of the German-speaking region reveals that the orchestration of the discipline-spanning development and the tools that enable this orchestration are seen as key enablers for future products [GCW+13].

In order to tackle the complexity of modern systems, most engineering disciplines use *model-based development* approaches. According to Stachowiak [Sta73], a model is a restricted representation (*abstraction*) of entities of the real world and connections between them used for well-defined purposes (*pragmatism*). In engineering, models are typically used to ease the development the system. For instance, by abstracting from unimportant details, models improve the understanding of a certain aspect of a system. Models can also be used to simulate the behavior of a system without using a real-world prototype (*model-based testing*). This reduces the necessity of costly prototype construction, and allows performing analyses during all phases of the development. In computer-aided engineering (CAE), we primarily find virtual models that only exist as a representation within computers.

During the development of a system, several development models reflect different aspects of a system under development. Models are used in all stages of the development process, and consequently, these models differ in purpose and viewpoint. For instance, there are models that exclusively cover mechanical engineering or software engineering aspects, and models that represent discipline-spanning information. In other words, the way a model abstracts from details and represents information depends on the purpose of this model (pragmatism).

As described before, it is necessary to keep all models consistent. Due to the difference in the models' abstraction and representation, a change constitutes differently in each model. In addition, when changing a model, an engineer may not even be aware of the consequences of this change to other, "foreign" models. Automatic *model transformation and synchronization techniques* are a promising approach to tackle such scenarios. However, as we describe in the following, existing approaches often focus on simple unidirectional transformation scenarios and lack features important for complex bidirectional synchronization scenarios. This restricts their applicability to relatively simple scenarios and mappings. Thus, existing model transformation techniques face new challenges in mechatronic system design: Models of different disciplines are manifold, and mapping between them is a complex task. Moreover, even if these transformation techniques would provide the necessary language features, defining, maintaining, and executing such complex transformation specifications is not adequately supported by the tools in use. As a result, a substantial amount of fine-grained user intervention is necessary, frequently offsetting the gains achieved through the increased automation due to the model synchronization.

Before summarizing the inapplicability of existing approaches in Sect. 1.2, we first describe the running example that is used in this thesis.

1.1 Running Example

As a running example throughout this thesis, we consider the *RailCab* research project[2], which also serves as an extensive case study for the CRC 614. The vision of the RailCab project is that, in the future, the schedule-based railway traffic will be complemented or replaced by small, autonomous RailCabs that transport passengers and goods on demand, being more energy-efficient by dynamically forming convoys. Figure 1.2 illustrates this vision, showing some aspects of this system and the test track built at the University of Paderborn.

[2]Neue Bahntechnik Paderborn/RailCab: `http://www-nbp.uni-paderborn.de/`

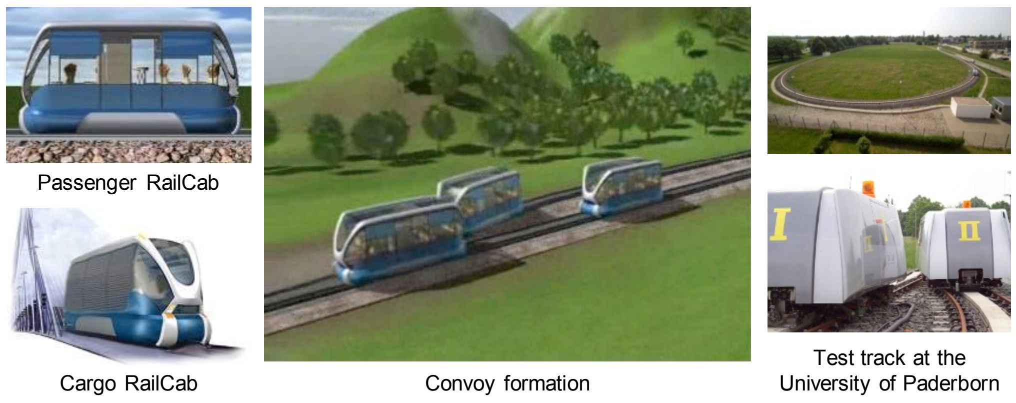


Figure 1.2: Illustration of the RailCab research project

In the following, we consider how the model-based development process could be applied to the RailCab project. In the first phase, an interdisciplinary team of engineers develops the principle solution of this RailCab system. The active structure is part of the principle solution and models the basic structure of the system under development. More specifically, it describes the *system elements* which the system comprises. Figure 1.3 shows an excerpt of the active structure of the RailCab.

A system element is a building block that fulfills one or more functions. For instance, a Distance Sensor[3] that measures the distance to other objects is such a system element. These system elements have *ports*, which form the interface to other system elements. Ports may transfer *material*, *energy*, or *information*. Consequently, there are *material flows*, *energy flows*, and *information flows* that connect ports of different system elements. In the case of a distance sensor, the sensor transfers the measured distance to a RailCab driving in front via an information flow to the system element Hazard Detection. System elements can also be hierarchically structured, i.e., they consist of further sub-elements in order to fulfill a more complex function.

Let us have a look into the active structure shown in Fig. 1.3. To move the RailCab, this Traction Unit generates a force F_{traction}, which is transferred to the railway tracks outside of the RailCab system (not contained in the figure). The Velocity Control is responsible to provide the Traction Unit with a designated force value F^* that will accelerate or decelerate the RailCab to the speed it should drive at. The different controllers inside the Velocity Control calculate this force, which will be described in the following.

RailCabs are able to form convoys dynamically when traveling. In a convoy, the RailCabs drive closely together to optimally make use of the slipstream, but there is no mechanical connection between the RailCabs. The participating RailCabs therefore have to drive at exactly the same speed. When driving alone or as a leader in a convoy, the Velocity Control compares the current speed with a given reference speed to calculate the acceleration or deceleration force. When following in a convoy, it uses the difference between a given reference distance to

[3]When referring to specific model elements, we use this sans-serif type face in this thesis.

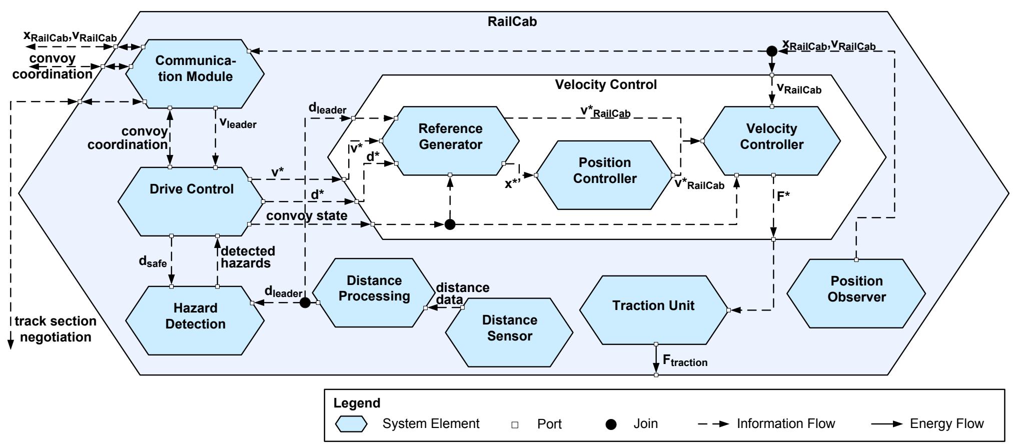


Figure 1.3: Active structure of the RailCab

the preceding RailCab and the actual distance measured by a Distance Sensor. In this way, RailCabs can ensure keeping a safe distance in a convoy. Also the Hazard Detection uses the distance measured by the distance sensor to initiate emergency responses or failsafe behavior.

The general behavior of the RailCab is also specified within the principle solution. For instance, the communication protocol to negotiate convoys is defined, as it is interdisciplinary relevant: Software engineers later have to prove the safety of the protocol, and control engineers have to allow for the time constraints when implementing the reconfiguration of Longitudinal Dynamics Controller.

As described, changes that affect several disciplines are likely to occur during the design and development phase. In the running example, the distance sensor is a highly safety-critical system element, as faulty data or a malfunction of the sensor when the RailCab is in a convoy may lead to a crash, putting human lives at risk. Therefore, after selecting the actual sensor component, the engineers have to perform a hazard analysis. It may turn out that the risk of the sensor causing such hazards is, in fact, too high. Thus, in such a case, the system engineers may decide to add a second distance sensor. In this way, the RailCab can detect bad data by comparing the values from both sensors. If they differ significantly, at least one of the sensors is not working correctly, and the system can initiate a fail-safe behavior, e.g., immediately leaving the convoy.

The necessary additions to the system model are relevant to several disciplines: The software and control engineers have to integrate the new sensor into their models so that the RailCab compares both values to detect possible sensor failures. Furthermore, the fail-safe behavior must be developed and added to the behavioral models, and the software engineers must prove whether this behavior is actually safe. In electrical engineering, engineers have to wire the sensor. Finally yet importantly, mechanical engineers have to mount the sensor into the chassis in their models.

Manually altering all discipline-spanning and discipline-specific models with the required changes is a highly time-consuming and error-prone task. Thus, we aim at a model consistency management support that helps the engineers in performing the necessary changes to all affected models (mostly) automatically. Figure 1.4 shows how the different models evolve and how information must be propagated between them during the development scenario described above. After generating initial discipline-specific models in step 1, engineers start refining their models (step 2), e.g., implementing control strategies in control engineering. In step 3, the second distance sensor is added to the system model. This change has to be propagated to all affected discipline-specific models in step 4.

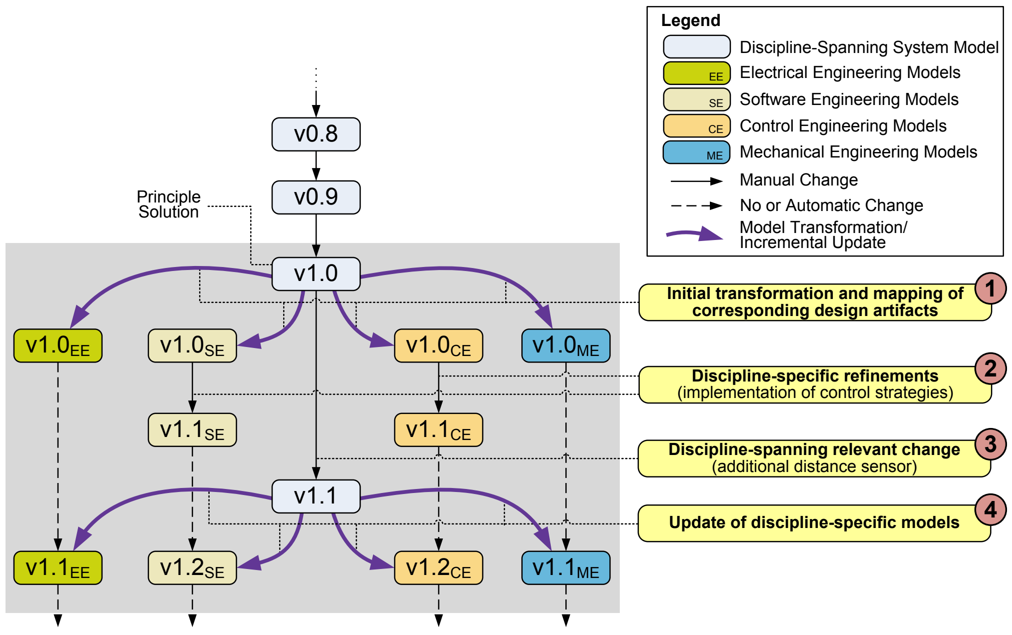

Legend
Discipline-Spanning System Model
Electrical Engineering Models
Software Engineering Models
Control Engineering Models
Mechanical Engineering Models
Manual Change
No or Automatic Change
Model Transformation/ Incremental Update
v0.8
v0.9
Principle Solution
v1.0
v1.0EE
v1.0SE
v1.0CE
v1.0ME
v1.1SE
v1.1CE
v1.1
v1.1EE
v1.2SE
v1.2CE
v1.1ME
1 Initial transformation and mapping of corresponding design artifacts
2 Discipline-specific refinements (implementation of control strategies)
3 Discipline-spanning relevant change (additional distance sensor)
4 Update of discipline-specific models

Figure 1.4: Evolution of the different models in the example scenario

1.2 Problem

Model transformation and synchronization techniques are a promising approach to tackle such model consistency issues. Languages and algorithms for bidirectional model synchronization are an intensively researched topic today (cf. [HLR06, GH09, GW09, XSHT09, LAVS12a, HLG$^+$12, Lau13]). However, existing model synchronization techniques mainly focus on application scenarios where models of the same or similar expressiveness and/or structurally similar models have to be kept consistent [RS12]. If, like in our case, models of different abstraction levels, different scopes, or of different disciplines have to be synchronized (*vertical transformations*[4]), these techniques are often insufficient. The graph structures, on which the model transformations work, differ significantly between, for instance, software engineering models like architecture descriptions, mechanical engineering models like CAD, and electrical engineering

[4]Horizontal transformations map between models of the same abstraction level, vertical transformations map between models of different abstraction levels [MG06].

models like block diagrams. In particular, existing techniques have three major problems in our scenario.

First, they may cause unnecessary *loss of information.* When the additional distance sensor is introduced, the discipline-specific implementation has already started: The different models now contain discipline-specific information that has been added by the discipline engineers. In the example scenario of Fig. 1.4, this is the case for the software engineering models ($v1.1_{SE}$) and the control engineering models ($v1.1_{CE}$). For instance, the software engineer may have already elaborated on the reconfiguration behavior, and the control engineers may have already implemented some of the controllers. This additional, discipline-specific information is not contained in the discipline-spanning system model, and therefore not subject to the model transformation and synchronization. When updating such modified and enriched models, existing model synchronization techniques cause unnecessary information loss to the discipline-specific models: Because they are not aware of such additional discipline-specific information, they may damage or even delete it.[5]

Second, existing model transformation techniques do not provide sufficient support for transformations between models of *different abstraction levels.* In our scenario, the system model has a higher level of abstraction than the discipline-specific models. This means that the discipline-specific models contain more information. Therefore, there often exist several ways to translate a concept from the system model to the discipline-specific models. For instance, controllers may be realized for different kinds of electronic control units (ECUs), communicating via different kinds of bus systems. Moreover, not all possible ways of concretizing an abstract concept in an discipline-specific model may be known in advance; i.e., the mapping between the models changes during the development. Existing model transformation techniques do not provide first-class support for such changing *1-to-*n *relations* between models.

Third, *editing conflicts*[6] may occur when engineers simultaneously modify related models. For instance, the time necessary to reconfigure the Longitudinal Dynamics Controller in control engineering may be in conflict with the timing constraints for the communication protocol that the software engineers selected. The model synchronization approach also has to support engineers in resolving these editing conflicts. Model differencing and merging tools can solve some of these conflicts automatically and provide dedicated means for manual conflict resolution. However, such tools are not integrated into existing model synchro-

[5]Especially model-to-text transformations (e.g., for code generation) address these issues by defining user-editable areas in the resulting target model/text. Changes to these areas are preserved in further transformation runs. However, this heavily restricts the kind of changes a user may apply to the target model/text. Furthermore, it only works if these areas are completely independent from the rest of the models, and do not influence the semantics of the other model parts – which typically does not hold for highly interlinked graph structures (i.e., models) like we find in our application scenario.

[6]In the context of model transformations, the term *conflict* is also used to describe a situation where more than one model transformation rule is applicable during a transformation run. When speaking about a "conflict" in this thesis, we mean *editing conflicts* (a conflict caused by two users concurrently editing models). Otherwise, we explicitly use the term "rule conflict".

nization approaches.

These three problems restrict the applicability of model transformations to simple scenarios. Some model transformation approaches partially tackle the described issues by introducing a large amount of *user interaction* or include an increased number of manual steps during the transformation. This can be a solution for certain use cases and scenarios, especially when there are many engineers available that have profound knowledge about more than one discipline and the models and tools in use – those engineers can then perform the necessary transformations and the respective user decisions. However, such engineers are typically difficult to find [GCW+13]. If they are available, they often have to focus their time effort on higher-level task like managing general interdisciplinary concerns.

Finally yet importantly, existing model transformation approaches suffer from bad understandability and maintainability of the transformation specifications, e.g., due to verbosity of their rules, or a lack of analysis and debugging facilities. On the other hand, we require a model transformation approach that has precisely defined semantics, because it is applied in a safety-critical context where uncertainties or impreciseness can lead to dangerous situations.

1.3 Objective

The objective of this dissertation is to improve existing model synchronization techniques in order to support the advanced requirements of a development process of mechatronic systems. Especially, we aim at resolving the problems described in the previous section. As these issues are partially interwoven and interdependent, we aim at integrating the improvements into a single model synchronization technique. To allow evaluating the approach, a prototypical tool suite shall implement this technique.

As mentioned, the problems described in the previous section can (at least partially) be solved with an increased manual effort when synchronizing models. In fact, in some situations, expert decisions are indispensable. On the other hand, manual solutions always increase the risk of errors and flaws and enlarge the workload for the engineers involved. Thus, we aim for a solution that provides an *adjustable level of automation*, i.e., its users can customize it from highly automatic to fully manual. Depending on the transformation scenario, engineers can fine-tune the solution to their specific needs and requirements.

Our hypothesis is that applying such an improved model synchronization technique within the development of advanced mechatronic system reduces the time spent on manual consistency assurance tasks. Consequently, engineers could spend more time on developing and analyzing the different models itself, which will allow that more consistency modeling errors and flaws are detected, and that they are detected earlier. This will accelerate the development process and reduce flaws in the developed system. Thus, it will eventually lead to better and less expensive products. We also expect other model-based development approaches, like Model-Driven Architecture (MDA) [OMG01], to benefit from this technique, as similar issues occur there, too.

1.4 Approach and Contribution

As there are three main problems, we propose a three-fold, but combined approach.

To approach the first problem, we have to *incrementally update without changing unaffected target model parts*, as models may contain discipline-specific information which otherwise might get lost. This is important when updating discipline-specific models that have been enriched with discipline-specific refinements in the meantime. We propose a novel model synchronization algorithm that avoids unnecessary deletion by a more intelligent way of rule revocation and application. The underlying idea is, when propagating changes, not to delete elements right away, but to *mark for deletion*. In this way, the algorithm preserves information and can reuse it during the model synchronization process if necessary.

Second, the improved technique should allow *relating models of different abstraction levels*. In this way, transformation engineers can more easily specify the relation between an abstract system model and concrete discipline-specific models. In addition, as new interrelations between models may still be identified during the development process, we need to support changing this abstraction-refinement relation later on. In general, defining a complete 1-to-n relation is often time-consuming and complex, even if a transformation technique would provide support for such 1-to-n relations. To simplify the definition of such abstraction and concretization functions, we propose that the transformation engineer first defines an initial transformation function I. In order to make it more comprehensible and maintainable, I only contains a default case and no discipline-specific refinements. The transformation engineer then defines so-called *refinement rules* that describe what kind of changes to a default discipline-specific model are refinements. These rules are described directly in terms of the discipline-specific model, so that even discipline engineers without further knowledge of the abstract system model and the transformation can define them. We combine the refinement rules with the initial transformation function I to form an overall consistency relation R that also covers the refinements.

To address the third problem, the technique should allow *concurrent modifications, conflict detection and resolution* in cases where simultaneous changes to models occur. Furthermore, it should support the engineers in restoring the consistency in cases where editing conflicts cannot be automatically resolved. We propose adopting existing model comparison and merging techniques for model transformations. Thereby, we can automatically solve many editing conflicts automatically. The user can then inspect remaining editing conflicts. To facilitate this manual task, we propose to use model synchronization previews to estimate the impact of possible conflict resolutions to all affected models.

The proposed approach also allows involving the user in ambiguous situations. We use a set of heuristics and metrics that represent and encode knowledge of transformation experts to allow reducing the amount of user interactions. This helps non-expert users in applying our technique. The amount of desired interaction can be customized according to the level of knowledge of the users and/or the availability of work force.

We use Triple Graph Grammars (TGGs) as the common model transformation formalism. TGGs are a rule-based, declarative model transformation language invented by Schürr [Sch95]. Its semantics are precisely defined (based on the well-known formalism of graph grammars), which allows, for instance, sophisticated formal analyses. This is particularly important when developing safety-critical systems, where we want to guarantee the correctness of the result of a transformation. However, there are drawbacks in terms of expressiveness, understandability, and maintainability of the transformation specification language. In order to implement the required model transformations of our application scenario (i.e., to the different disciplines' models), we develop some extensions to the TGGs formalism that increase its expressiveness. Further TGG extensions aim at an improved user experience when developing a model transformation, for instance, debugging facilities and an alternative concrete syntax for TGG rules.

1.5 Structure of this Thesis

This thesis is structured as follows. In Chap. 2, we lay the foundations necessary to understand the contents of this thesis. We suggest to read this section selectively or to use it as a reference. Chapter 3 provides an overview about the scenario. Here we describe the concepts of the different model transformations and explain the running exemplary process, which is used throughout the thesis to motivate and explain the different contributions. The core contributions of this thesis are presented in Chap. 4 and 5. We describe our new model synchronization algorithm in Sect. 4.1, the technique to define abstraction/refinement model transformation in Sect. 4.2, and the support for concurrent changes in Sect. 4.3. Chapter 5 presents the extensions to the TGG formalism that have been developed in this thesis. In Chap. 6, we present how we implemented the different developed techniques in our prototype tool, the TGG Interpreter tool suite. Furthermore, we evaluate the developed approaches. We conclude this thesis in Chap. 7 and give an outlook on future research directions. As a reference, Appendixes A and B contain the specifications of transformations that we developed in the course of this thesis.

CHAPTER 2

Foundations

Contents

This chapter introduces the foundations for this thesis. The principles, processes, and languages used in a model-based, multi-disciplinary development of mechatronic systems are explained in Sect. 2.1. Section 2.2 introduces the basic notions, terminology, and features of model transformations. Before introducing Triple Graph Grammars (TGGs), the model transformation technique used in this thesis, in Sect. 2.4, we explain graph grammars, the foundation of TGGs, in Sect. 2.3.

2.1 Model-based Development of Mechatronic Systems

The complexity of technical systems has grown tremendously over the last decades. For instance, transportation systems like cars were mainly mechanical systems with no or little electronics before 1980. In the 1980s, the use of anti-lock braking systems (ABS) in serial-production cars marked a first significant change. Since then, software running on electronic control units (ECUs) increasingly influences the behavior of cars. Nowadays, even middle-class cars are software-intensive systems that include a high degree of automation, e.g., parking assistants or adaptive headlights. Today, we are on the verge of a second significant change – a change towards advanced mechatronic systems that form so-called *systems of systems*, i.e., autonomous systems that communicate with each other to perform tasks that would not be achievable individually.

Developing such advanced mechatronic systems is a complex task and poses several challenges to the development process and the engineers involved. It requires the expertise of several disciplines, namely mechanical engineering, electrical engineering, control engineering, and software engineering. In particular, the interdisciplinary dependencies require a strong collaboration between the different disciplines involved. Moreover, the development is often also partitioned into separate modules that are developed by different teams or subcontractors. This further increases the need for an effective collaboration.

To tackle the complexity of such a development, most engineering disciplines use *model-based development* approaches. Thus, we first describe the notion of a model and its use in development in Sect. 2.1.1. In Sect. 2.1.2, we describe how the development of such systems proceeds. This process is divided into two main phases. We describe these process phases in detail in Sect. 2.1.3 and 2.1.4.

2.1.1 Models and Model-Based Development

According to Stachowiak [Sta73], a model is a representation of an original that is restricted in a particular way. It can be characterized by three main properties:

- *Representation*: A model represents an original. The original can be real-world entities and/or notions, but also other models.
- *Abstraction*: A model represents an original such that it abstracts from some of its properties, i.e., partially or fully disregards them in the model.
- *Pragmatism*: The way in which a model abstracts depends on the purpose of the model. The model disregards properties that are irrelevant for the model's purpose.

A model is valid for a given purpose if operations on the model yield results that are equivalent to performing these operations on the original (or are approximating the results good enough for the given purpose).

In system development, models are used to ease the development in several ways. For instance, by abstracting from unimportant details, models improve the understanding of a certain aspect of a system. Models can also be used to simulate the behavior of a system without using a real-world prototype (*model-based testing*). This reduces the necessity of costly prototype construction, and allows performing analyses during all phases of the development.

In computer-aided engineering (CAE), we find virtual models that exist as a representation within a computer. Using this representation, we can, for instance, perform analyses like a formal verification of the behavior of a system to ensure that no deadlocks or unsafe states can ever be reached.

The most important types of models for the development of mechatronic systems are [GST14]:

- *Process models* describe sequences of process steps, e.g., models for development processes.
- *Requirements models* define which (functional and non-functional) requirements a system has to fulfill.
- *Structural models* describe the hierarchical and modular structure of the components of a system. They can be physical (e.g., representing constructional elements) or logical (e.g., representing a software architecture).
- *Behavioral models* describe the dynamic, temporal, or static behavior of a system. Examples are statecharts, which encode the behavior as states and transitions, or signal-flow-graphs (block diagrams), which encode it using mathematical equations.
- *Geometric models* model the (two- or three-dimensional) shape of a component. *CAD models* often further contain information on materials, dimensions and its tolerances.

Typically, models are created for a specific purpose, and they cover a specific field (*domain*). To allow a meaningful use of models, they have to be formalized such that their syntax and semantics are clear and unambiguous. A *domain-specific language* (DSL) precisely defines how its instances (models) are built and which semantics they have [SV06]. In particular, a DSL consists of a *metamodel* to define

- its *abstract syntax*, i.e., the building blocks (elements) of a model, their properties, and how they relate to each other, and
- its *static semantics*, i.e., additional criteria on the well-formedness of models.

Additionally, a DSL defines[1]

- at least one *concrete syntax*, i.e., how elements of the DSL are represented visually, e.g., using a graphical or text-based representation, and
- its *dynamic semantics*, i.e., the meaning of the different modeling constructs in particular situations.

2.1.2 Development Process

As explained above, software plays an increasing role in modern engineering products. For instance, experts estimate that software will induce over 70 to 80 percent of future innovations in the automotive sector [Gri03, HKK04, BKPS07]. However, the development processes used in practice are often not suited for

[1]Some authors regard the concrete syntax and dynamic semantics definitions as part of the metamodel [RH09, pp. 94f]; in practice, however, most DSL tools separate these.

such advanced mechatronic systems engineering. In fact, until a few years ago, the development often followed a so-called "throw it over the wall" principle [AP96, SW07], as depicted in Fig. 2.1. First, the mechanical engineers started developing the system according to their needs. Once finished, the results were handed over to the electronics development, then further making its way through control engineering to software engineering. The underlying problem is that the traditional development process was not adapted thoroughly, but simply extended by adding the supervening disciplines' processes at the end.

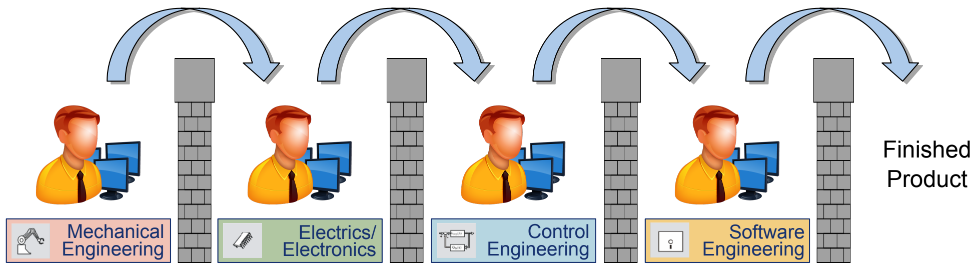


Figure 2.1: Traditional sequential development process ("throw it over the wall")

It is obvious that such a development process does not only take more time (as the disciplines cannot develop simultaneously), it also drastically impedes the fixing of design flaws that are detected during the later phases. In order to overcome these process-related issues, several interdisciplinary design methods and processes have been proposed. Most of them are derived from the V-Model [FM92], which was developed in the late 1980s and early 1990s. A general design guideline for mechatronic systems is described by the VDI standard 2206 [VDI2206], published by the Verein Deutscher Ingenieure (VDI). The concept of developing different aspects of a system (and also the associated production system) simultaneously is known as *concurrent engineering* [FGYO95, MCT08].

The V-Model, as depicted in Fig. 2.2, is a general development process that has to be adapted for the specific needs and requirements of the type of system under development. For instance, development methods elaborated in the Collaborative Research Center (CRC) 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering" in Paderborn further improve, extend, and detail this guideline, with respect to intelligent, self-optimizing mechatronic systems.

In this thesis, we assume that this adapted development process of the CRC 614 is used. One important difference to the V-Model is that the distinction between the phases "Design and Development" and "System Integration" has been removed. Instead, the system integration is seen as a continuous part of the second phase. A continuous integration of the results of the disciplines during the second phase allows a shorter response time to problems and failures and therefore reduces development time and costs [GRS14a]. This especially holds for highly structured systems, i.e., systems that contain several subsystems/modules that are developed independently.

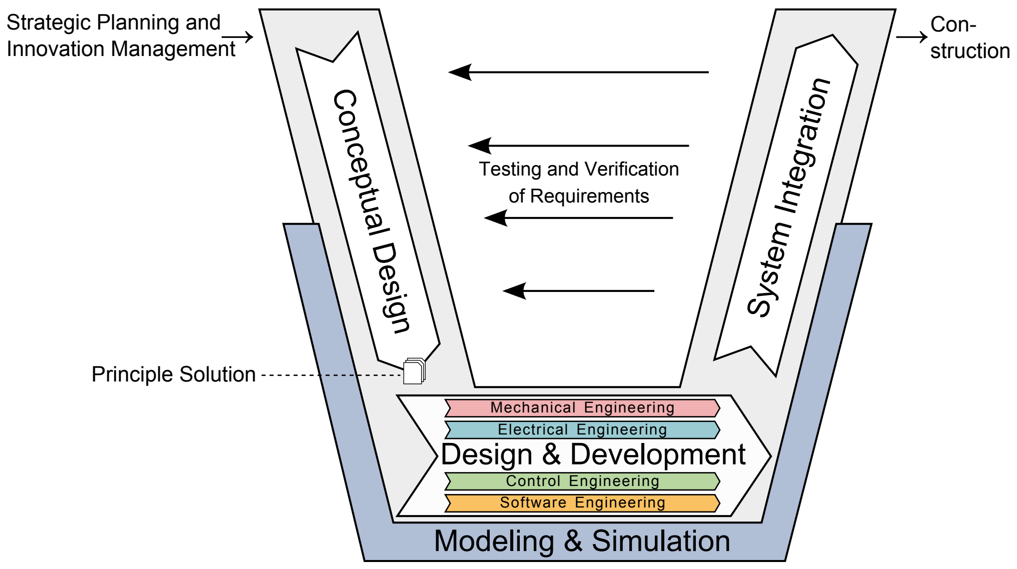


Figure 2.2: V-Model as a process for mechatronic system development (adapted from [VDI2206, BRD06])

In the following sections, we describe these two phases with a focus on the development of self-optimizing mechatronic systems. In the first, *conceptual design* phase, experts from all disciplines collaborate to develop the conceptual design, as explained in Sect. 2.1.3. Second, the engineers design all discipline-specific details of the system under development during the *design and development* phase, described in Sect. 2.1.4. For details of the development process and the development phases, we refer to Gausemeier et al. [GFDK09, GRS14b].

2.1.3 Interdisciplinary Conceptual Design

The goal of the first, interdisciplinary *conceptual design* phase is to define the basic principles and concepts of the system under development. All disciplines that are involved in the development collaborate in developing a system model that defines everything necessary to start the discipline-specific design and development. The result of the phase, therefore, is the so-called *principle solution*, which covers at least all discipline-spanning information.

There are different modeling techniques to specify such a principle solution. One well-known language is the *Systems Modeling Language (SysML)* [OMG10a], which has been developed by the Object Management Group (OMG) together with the International Council on Systems Engineering (INCOSE). SysML is a very generic language, and there is no strong connection to a reference process [ABD$^+$14]. Because it is usable for a wide range of systems, users have to tailor it to the type of system under development (e.g., using profile mechanisms).

In this thesis, we use a language called CONSENS (**Con**ceptual Design **S**pecification Technique for **En**gineering of Complex **S**ystems) [GFDK09], which was developed in the CRC 614. It is tailored for developing complex, self-optimizing mechatronic systems. It consists of a reference process and several distinct, yet coherent *partial models*, each describing a different aspect of the system. Figure 2.3 shows these partial models. Here, we focus on the partial models that are most important for this thesis, namely *Active Structure* and *Behavior*. For further reading, we refer to Frank [Fra06] and Gausemeier et al. [GFDK09], who describe this specification language in detail.

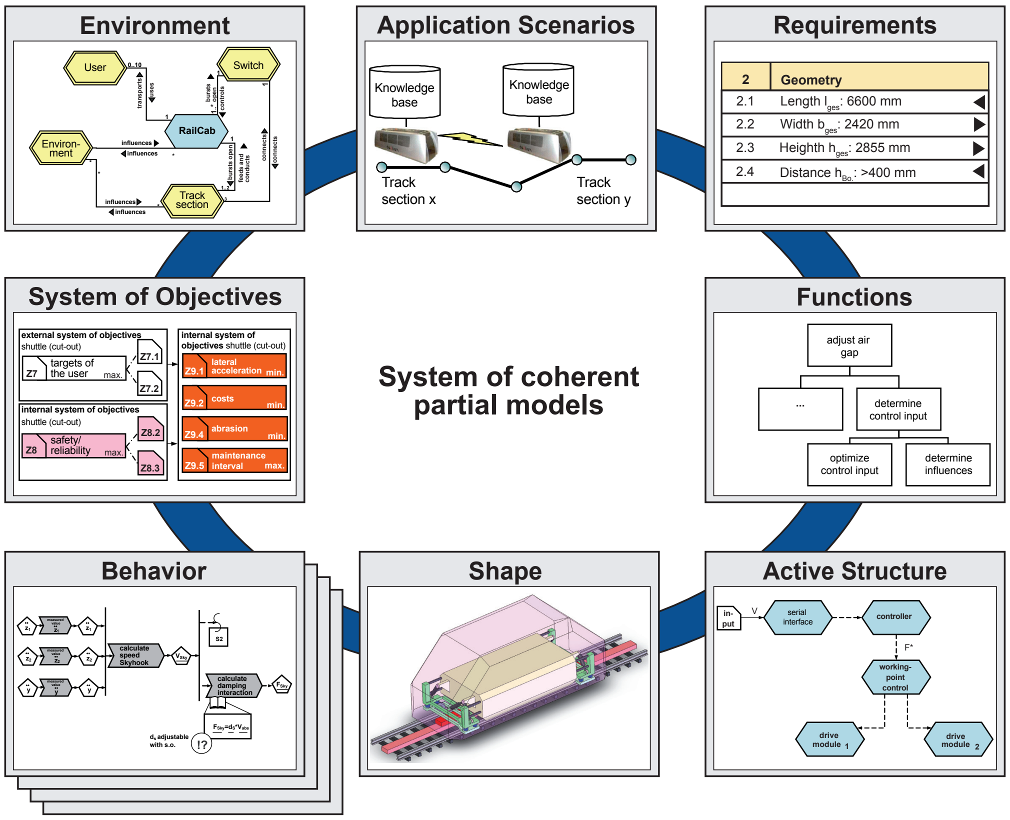


Figure 2.3: Aspects of the principle solution (adapted from [Fra06, GFDK09])

The conceptual design phase consists of four main activities, which are depicted in Fig. 2.4. In the following, we briefly describe these activities of the first phase briefly. For a more detailed introduction, we refer to, e.g., Gausemeier et al. [GFDK09], Adelt et al. [ADG+09], and Gausemeier et al. [GRS14b].

2.1.3.1 Planning and Clarifying the Task

The conceptual design phase starts with *planning and clarifying the task*. This is where the classical *requirements engineering* takes place: We analyze the environment and the system's interfaces to it and identify application scenarios and core requirements of the system under development. The results are specified using the partial models Environment, Application Scenarios, Requirements, and System of Objectives.

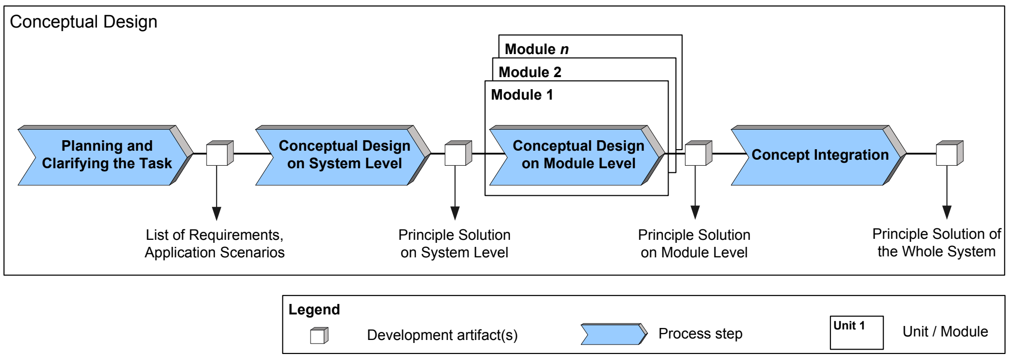

Conceptual Design
Module n
Module 2
Module 1
Planning and Clarifying the Task
Conceptual Design on System Level
Conceptual Design on Module Level
Concept Integration
List of Requirements, Application Scenarios
Principle Solution on System Level
Principle Solution on Module Level
Principle Solution of the Whole System
Legend
Development artifact(s)
Process step
Unit 1
Unit / Module

Figure 2.4: Conceptual design phase (adapted from [GFDK09])

Next, we continue with the conceptual design on system level, using these determined requirements.

2.1.3.2 Conceptual Design on System and Module Level

In this phase, different solution variants for the design of the complete system are explored and evaluated. After engineers choose the best design for the system, they continue with the same process for all designated modules of the system.

In detail, they first define the *function hierarchy* such that the functions of the system fulfill the defined requirements. They may reuse existing solution patterns that have proven useful in previous systems. Information about such solutions patterns can be found using online catalogs. However, the search in such catalogs is based on keywords rather than specific solution pattern information like functions. Thus, semantic technologies (like the “Semantic Web”) are a promising approach to improve identifying reasonable solution patterns [OJT+12, GST14].

Once the engineers identified promising solution variants, these solution patterns form the basis for defining the Active Structure.

Active Structure Figure 2.5 shows a simplified active structure of the RailCab system. Such an active structure describes which elements the systems comprises and how these system elements are cross-linked, i.e., how they influence each other. More specifically, there are unidirectional or bidirectional flows that connect system elements. There are three types of flows: energy, information, and material (the latter not contained in the figure).

In Fig. 2.5, there are several information flows between system elements, and an energy flow from the Traction Unit to the Track Section. The latter is the actual driving force that accelerates or decelerates the RailCab. The traction unit receives a target driving force F^* from the Velocity Control. This system element is responsible for calculating the target speed in different driving modes. For a more detailed discussion of the active structure of the RailCab see Chap. 3.

Behavioral Modeling Engineers also define the basic principles of the behavior of the system during the conceptual design.

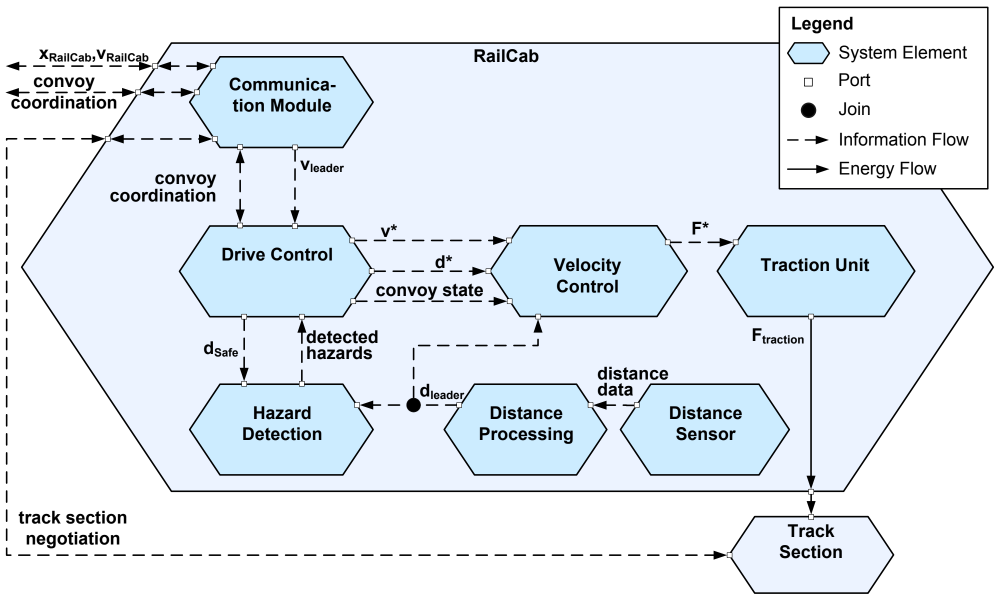


Figure 2.5: Simplified active structure of the RailCab

In general, the behavior of a mechatronic system can be separated into two parts: the *discrete behavior* (i.e., event-based and communication behavior, usually defined using statecharts) and the *continuous behavior* (i.e., controller and physical/dynamic behavior usually defined by differential equations and/or block diagrams). The principle solution primarily considers discrete behavior.

As an example, consider the Behavior–States model in Fig. 2.6. This model defines the discrete behavior in terms of states and the necessary communication for forming and breaking convoys. It consists of three states: In the noConvoy state, the RailCab is in single-driving mode; this is the default state In states convoyLeader and convoyFollower, the RailCab is in a convoy, either leading it or following another RailCab.

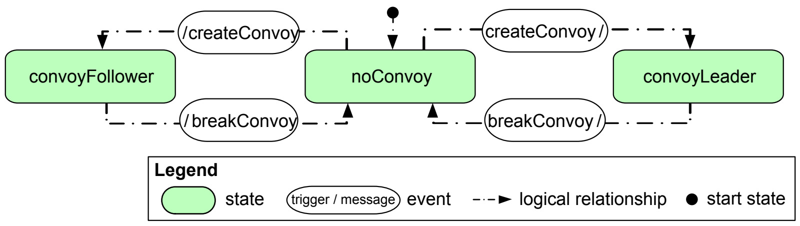


Figure 2.6: Behavior–States model defining the discrete and communication behavior for convoys

According to the statechart of Fig. 2.6, switching between the different states happens when certain message are sent or received. For instance, when a Rail-Cab approaching a preceeding RailCab, it sends a message createConvoy to initiate a convoy with the preceding RailCab. (Sending a message is denoted as the message's name after the slash "/", reception, or *trigger*, as the message's name after the slash. Both the trigger or the message can be empty.) The rear RailCab switches from the noConvoy state to the convoyLeader state. However, such a switch also involves the continuous, dynamic behavior of the system, as the Velocity Control system element must be reconfigured to now use the distance to the leading RailCab to control its speed. So the discrete behavior and the continuous behavior are highly interlinked.

The behavioral models that are developed during the conceptual design can only serve as a first, rough sketch, and must be refined during the design and implementations phase. Therefore, the engineers of these two disciplines have to closely collaborate during the second phase.

2.1.3.3 Concept Integration

In the final step of the conceptual design, engineers combine the selected concepts to form the detailed principle solution for the whole system. We check whether there are conflicts between the chosen concepts. Furthermore, we analyze and evaluate the complete solution (in contrast to the evaluation on the sub-system level performed before).

If any issues or deficiencies are identified, we go back to the respective development phase. For instance, we may have to select a different solution variant for a sub-system and, thus, perform the conceptual design on subsystem level again. Only if the complete principle solution has no deficiencies, we move on to the next phase, the discipline-specific design and development, where the disciplines start their discipline-specific development tasks.

2.1.4 Discipline-Specific Design and Development

The principle solution that has been developed in the conceptual design phase is used as a starting point for the discipline-specific design and development. In this phase, the disciplines work mainly on their own and use their own discipline-specific development methods, models, and tools. As depicted in Fig. 2.7, a separate development process is started for each system module. In parallel to the disciplines' development processes, the system engineers have a close look on the discipline-spanning interplay. In particular, they perform the module integration, where they keep track of the development artifacts of the different disciplines and synchronize changes between the different disciplines' models. The same holds for the whole system: changes to a module may also affect other modules. Therefore, we need to propagate such relevant changes to other models of other modules.

Modern technical systems increasingly rely on software to fulfill their functions, and software is the key innovative factor for many of these systems [HPR$^+$12]. For instance, in a car, software influences acceleration (traction control systems), braking (anti-lock braking system (ABS)), and even steering (automatic parking). Already in 2003, DaimlerChrysler experts estimated "that 80 percent of all future automotive innovations will be driven by electronics, 90 percent thereof by software." [Gri03]. The amount of software in cars grows exponentially, and a modern car nowadays has tens of millions of

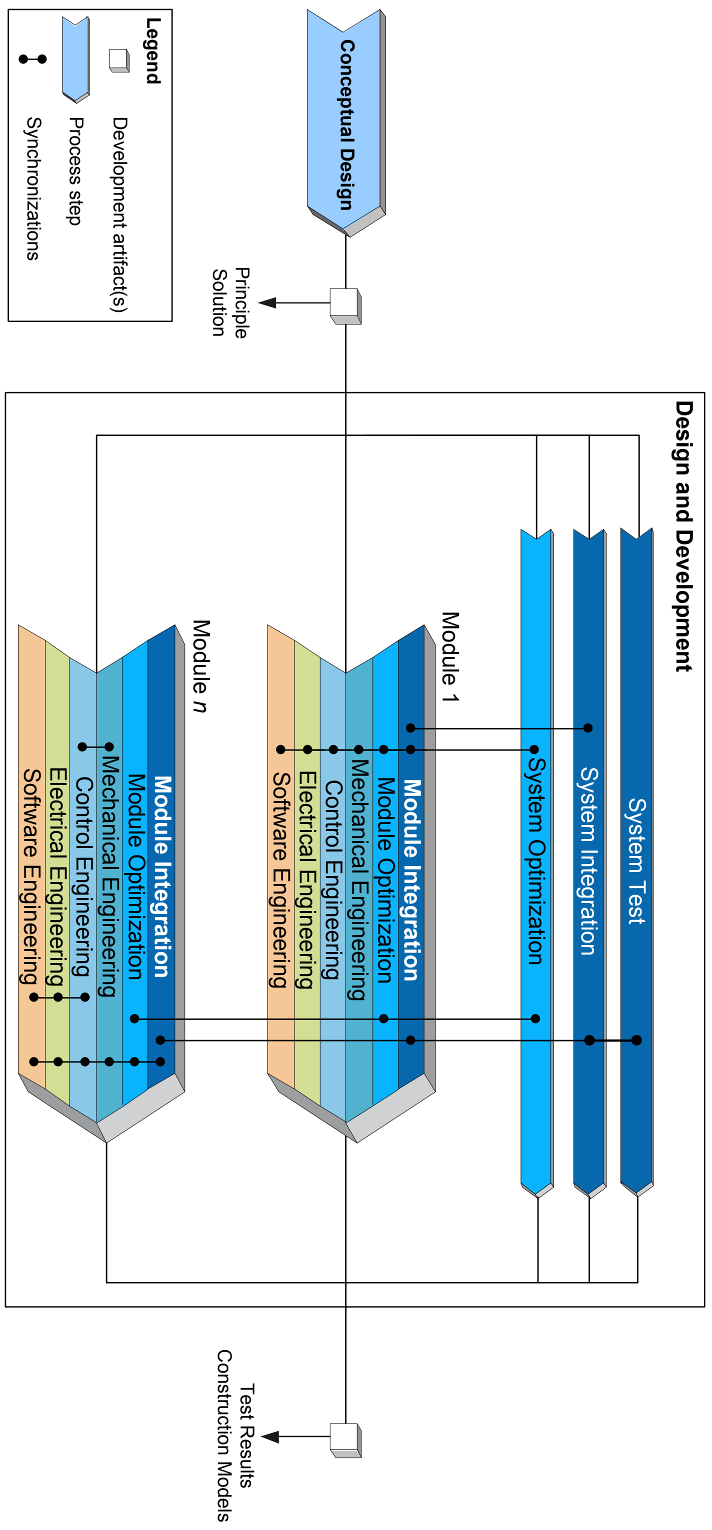


Figure 2.7: Design and development phase (from [GRS14a])

lines of code [BKPS07]. In the near future, we will have true autonomous driving available for standard passenger cars. This again will dramatically increase the amount of software in a car, but also the necessity for car-to-car communication. For instance, vehicles will exchange information on planned routes to reduce congestions or notify each other about oncoming hazards like rough road conditions and accidents. The exchange of information influences the behavior of the vehicles. This is typically called "system of systems", i.e., different systems interact and cooperate using message-based communication.

Regarding the software within such technical systems, we distinguish between *continuous* and *discrete* parts of software. The continuous parts, which are designed by control engineers, contain the controllers that continuously process input data from sensors to compute outputs for actuators. Their behavior is usually defined by differential equations and/or block diagrams.[2] For instance, the ABS uses continuous controllers to actuate the brakes of each wheel individually. The discrete software defines, for instance, the message-based communication or event-driven behavior like system states and transitions. For instance, cars communicate with each other using discrete software. It is typically implemented by software engineers.

However, there is a strong relation between continuous and discrete software. The car-to-car communication influences the behavior of the continuous controllers, e.g., when another car warns about a hazard and the car decides to decelerate. In a convoy of RailCabs, the convoy is negotiated by message-based communication between the RailCabs, and that determines the mode of operation of the velocity controller. Therefore, the interplay and consistency between software and control engineering models is of great importance when developing modern technical systems. For more details on interplay between the continuous behavior and the discrete software, we refer to Burmester et al. [BGO06] and Heinzemann et al. [HSST13].

Due to the growing importance of software-related aspects, we focus mainly on the disciplines of software engineering and control engineering in this thesis. However, the methods and approaches presented in this thesis can be applied for all disciplines that are involved in developing complex technical systems.

Next, we describe the development process in software engineering and control engineering and introduce the discipline-specific models used in this thesis. For further details on the specific developments tasks and processes that are performed in the different disciplines, we refer to Gausemeier et al. [GRS14a].

2.1.4.1 Software Engineering

In general, software engineers implement software that is responsible for the discrete behavior of the system. For instance, this could be the message-based

[2]Note that in most modern mechatronic systems the continuous behavior is implemented as controller software that runs on embedded devices (ECUs). This controller code is executed periodically, e.g., every 0.5 ms. Also the analog/digital converter for the sensor data runs periodically (*signal quantization* with a given *sampling rate*). As a result, this controller code is also not truly continuous; sometimes, it is also referred to as "quasi-continuous".

communication between two system elements or between several RailCabs, or event-driven behavior like system states and transitions between states.

First, software engineers refine the software architecture that was defined by the “software” system elements in the active structure. Every software system element is mapped to a software component. The software engineers may now introduce sub-components or add new ports and connectors when additional communication is necessary.

Most importantly, they implement the real-time behavior of the components by means of statecharts. In this thesis, we use MechatronicUML [BBB+12] as a specification technique for the component real-time behavior. The idea is that only components on the “lowest level”, so-called *atomic components*, have a behavior assigned. The behavior of *structured components* (which consist of other structured and atomic component) is only defined by the behavior of its sub-components. This is also called *compositional behavior*.

With atomic components, we distinguish between *continuous* and *discrete components*. Continuous components are typically controllers that continuously process input data from sensors to compute outputs for actuators. These controllers are implemented in control engineering (cf. Sect. 2.1.4.2). However, MechatronicUML allows to integrate them as “black-box” components, i.e., no actual behavior is attached to continuous components in MechatronicUML. In this way, they can be used to define the interface to control engineering in a MechatronicUML software model.

In contrast, the behavior of discrete components is implemented using MechatronicUML. Discrete components communicate with each other via discrete ports using asynchronous, message-based communication. This communication is implemented using real-time statecharts. Discrete components can also send or receive signals to or from continuous components using hybrid ports.

Component Structure Figure 2.8 shows the software component structure of the RailCab system. It resembles the system element structure from the active structure (cf. Fig. 2.5), but contains only the parts that are relevant to software engineering. For instance, the Communication Module, which realizes the wireless connection between RailCabs on a hardware level, has no direct counterpart in the software model. Instead, the properties of the wireless connection, e.g., average round-trip time, packet loss probability etc., are mapped to properties of the corresponding connectors in the software model. In this way, we abstract from the technical details of the Communication Module to allow focusing on the software-specific information.

As explained, components may consist of sub-components. In the example, the Velocity Control component consists of up to three instances of atomic components, as Fig. 2.9 shows.

MechatronicUML allows specifying dynamically changing software component structures, so-called *reconfigurations* [HB13]. In Fig. 2.9, you see that the Reference Generator and the Velocity Controller have cardinality 1, where the Position Controller has cardinality 0..1. Thus, we can reconfigure the Velocity

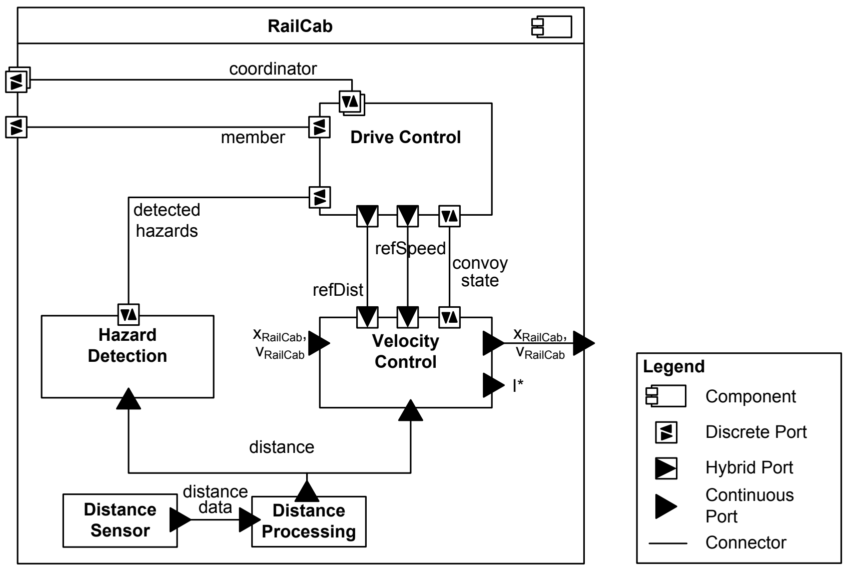


Figure 2.8: Software component structure of the RailCab (excerpt)

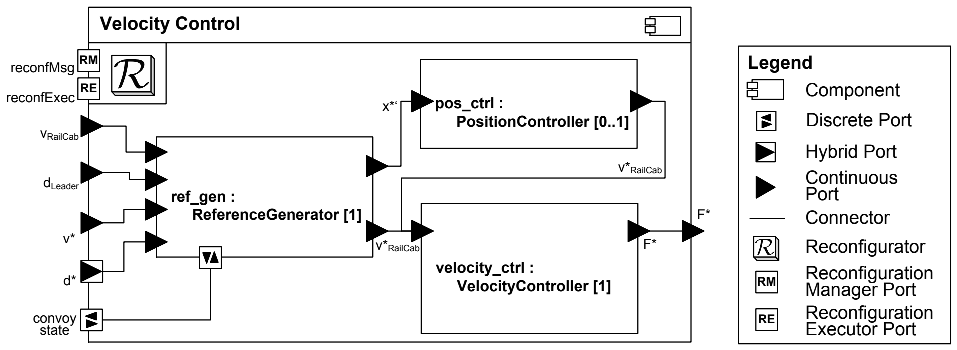


Figure 2.9: Software component structure of the Velocity Control component

Control component by instantiating or destroying the instance pos_ctrl : Position Controller [0..1] during runtime.

Component Behavior We define the state-based behavior of software components using *real-time statecharts*. Real-time statecharts combine features of UML state machines [OMG10b] and timed automata [AD94, BY03]. In the following, we only briefly introduce their main features. For a detailed description of the syntax and the semantics of real-time statecharts, we refer to *Becker et al.* [BBB+12].

Figure 2.10 shows such a statechart that implements the discrete behavior of the DriveControl component. This component is responsible for negotiating convoys with other RailCabs as well as triggering the necessary reconfiguration in the RailCab's controllers when entering or leaving a convoy.

Initially the component is in state noConvoy. In that state, it continuously evaluates whether it is reasonable and useful to build a convoy. It is useful to

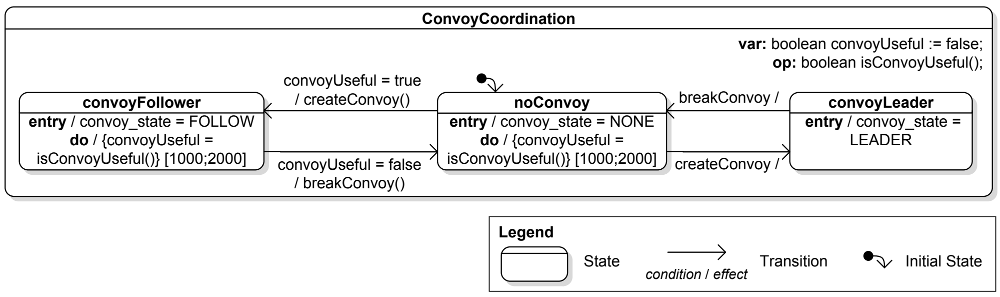

ConvoyCoordination
var: boolean convoyUseful := false;
op: boolean isConvoyUseful();
convoyFollower
entry / convoy_state = FOLLOW
do / {convoyUseful = isConvoyUseful()} [1000;2000]
convoyUseful = true / createConvoy()
convoyUseful = false / breakConvoy()
noConvoy
entry / convoy_state = NONE
do / {convoyUseful = isConvoyUseful()} [1000;2000]
breakConvoy /
createConvoy /
convoyLeader
entry / convoy_state = LEADER
Legend
State
Transition
condition / effect
Initial State

Figure 2.10: (Simplified) statechart that defines the behavior of the DriveControl component

build a convoy if we drive behind another RailCab and share a larger part of our route, such that exploiting the slipstream effect saves more energy than what is required to form the convoy (e.g., the necessary acceleration to approach the preceding RailCab). Typically, reasoning algorithms that are not specified in MechatronicUML perform such calculations, because these algorithms cannot be easily implemented in hard real-time using state-based behavior. Therefore, we implement such behavior in external operations and allow calling these operations from MechatronicUML. Here, we call the operation isConvoyUseful() every 1000 to 2000 ticks (usually milliseconds), assigning its return value to the local variable convoyUseful.

As the local variable convoyUseful becomes true, it enables the transition from the noConvoy state to convoyFollower. This transition has a *raise message* as effect, createConvoy(). This message is sent to the leading RailCab to inform it that we are now creating a convoy.

Similar to the RailCab that is approaching from the rear, this leading RailCab previously was in the noConvoy state when it receives the createConvoy() message. This enables the transition to the convoyLeader state, so the leading RailCab will switch to the convoyLeader state.

The follower RailCab continuously monitors the usefulness of the convoy in the convoyFollower state. When it is reasonable to leave the convoy, the RailCab takes the transition back to the noConvoy state, sending a breakConvoy() message to the leader RailCab. When the leader RailCab receives this message, it also switches back to noConvoy state, so that both RailCabs are now in single driving mode.

Reconfigurations When a RailCab enters a convoy as a follower, it has to change the way the RailCab's speed is controlled: The target velocity is now calculated based on the distance to the preceding RailCab (cf. Fig. 2.9). We also call this *reconfiguration* [HB13]. To perform such reconfigurations, we allow the transitions of a statechart to trigger so-called *reconfiguration rules*. These rules perform the actual instantiation or destruction of component instances and the rewiring of port connectors. In our example, there is a rule that instantiates the pos_ctrl (cf. Fig. 2.9) and its connectors whenever the RailCab enters the state convoyFollower. These reconfigurations are specified using durative graph transformation rules, i.e., the execution of the reconfiguration may take time. For brevity, we omit the details of reconfiguration here and refer to Heinzemann and Becker [HB13] instead.

Hazard and Risk Analysis As most mechatronic systems operate in safety-critical scenarios, ensuring their safe operation is crucial. One aspect of a system's safety is that the software running on it is free of bugs. MechatronicUML supports model checking to proof safety properties and therefore helps ensuring that the software contains no safety-critical faults.

As mechatronic systems also consist of hardware parts, this hardware may fail as well. For instance, the sensor that measures the distance to the leading RailCab may fail, which leads to a potentially hazardous situation when driving in a convoy. Hardware errors are often due to wear, are random and can be neither predicted nor avoided. It is important to ensure a certain level of dependability nevertheless. The developers must guarantee that hazards due to hardware failures only occur with a probability below a given threshold. As not every hardware failure immediately leads to a hazardous situation, we must analyze how this failure propagates through our system and in which situations it may lead to hazards. The software engineers perform a *hazard and risk analysis* to identify hazards that occur with a probability above the given threshold.

If such a hazard is identified, the software engineers have different options to solve the problem. They can apply *self-healing* methods, e.g., they can implement a reconfiguration of the software architecture to prevent the bad data from the sensor to propagate through the software of the system. If self-healing is not possible or does not reduce the hazard probability sufficiently, the engineers have to re-design parts of the system. For instance, adding redundancy, i.e. an additional distance sensor, helps reducing the hazard probability, as the system may switch to the second sensor in case of a failure.

For details of the hazard and risk analysis, we refer to Priesterjahn et al. [PST13, GRSS14]

2.1.4.2 Control Engineering

The control engineers deal with the continuous behavioral aspects of the system.[3] Mainly, these are:

- the physical behavior of the mechanical system parts, i.e., the dynamics of the multibody system; also called *system* or *plant*, and
- the information processing, i.e., the controllers that compute the outputs for the actuators from the sensor inputs.

Together, these aspects form the control loop, as depicted in Fig. 2.11.

Continuous, dynamic behavior models When the system engineers have created a first principle solution of the system, the control engineers start validating this principle solution. They create a simplified, idealistic version of the

[3]In the literature, these aspects often are further separated into several disciplines, like, for instance, mechanical engineering, mechatronics, systems theory, or control engineering. Here, we subsume all these aspects under the term "control engineering".

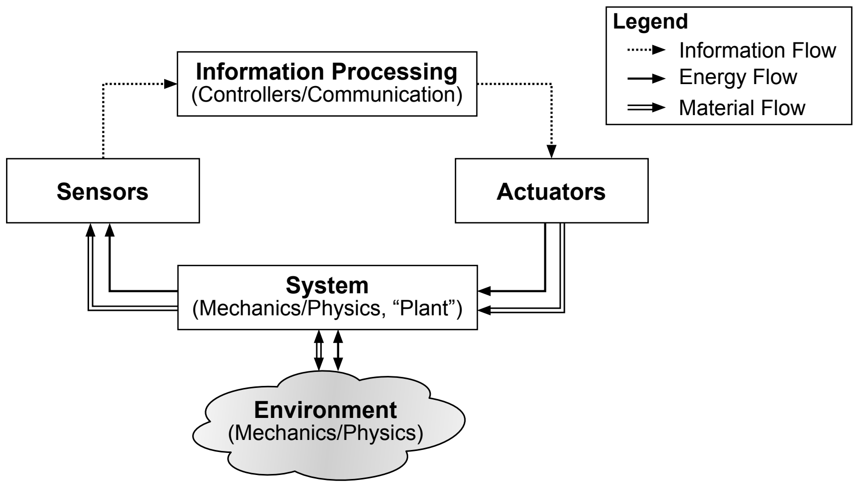


Figure 2.11: Basic structure of a mechatronic system ("control loop")

control loop: They identify the physical effects of the system and the environment, define the interfaces between the system and the environment, and create simplified controllers. With the help of this idealistic model of the system dynamics, the engineers are able to check whether control strategies are realizable for the planned system concept.

Once the engineers have proven the control concept of the principle solution, the actual implementation of the controllers starts. This requires a detailed physical system and environment model. This model is much more detailed than the early idealistic model and also contains things like environment disturbances, e.g., wind resistance. Based on this dynamics model, the control engineers design the control strategies. The controllers are typically implemented using tools like MATLAB/SIMULINK[4] or Dymola[5].

Here, we focus on MATLAB/SIMULINK, as it is the industrial de-facto standard. Figure 2.12 shows a cutout of the velocity control strategy implemented using a MATLAB/SIMULINK block diagram. It models how information flows and how it is processed within the different blocks. It resembles the system element structure from the principle solution (cf. Fig 2.5). You see that switching between different inputs for the velocity_ctrl is implemented using a Multiport Switch block. However, this can cause a sudden step in the value of the input signal for the velocity_ctrl whenever switching between convoy state. A step in the input of the block may, in turn, cause a step in the output actuating variable F^*. Such a sudden step is uncomfortable to passengers, can make the system instable or even damage the actuator. Therefore, this change must not happen immediately. Instead, the system must fade between the output values of two controllers in a given time period. Thus, an input switch is usually implemented using a *fading function*, as depicted in Fig. 2.13. A fading function could be a simple blending curve (e.g., a linear or sine functions) [BGO06] or flat switching functions [OMT+08].

[4]MATLAB/SIMULINK website: `http://www.mathworks.com/products/simulink/`

[5]Dymola website: `http://www.3ds.com/de/products/catia/portfolio/dymola`

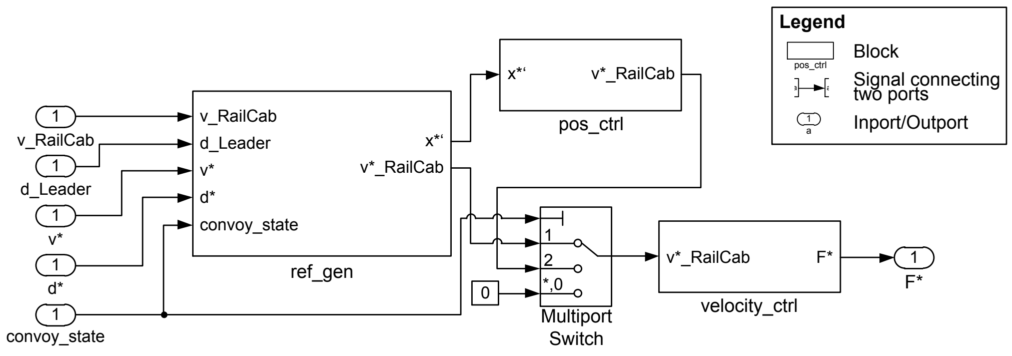


Figure 2.12: Velocity controllers of the RailCab

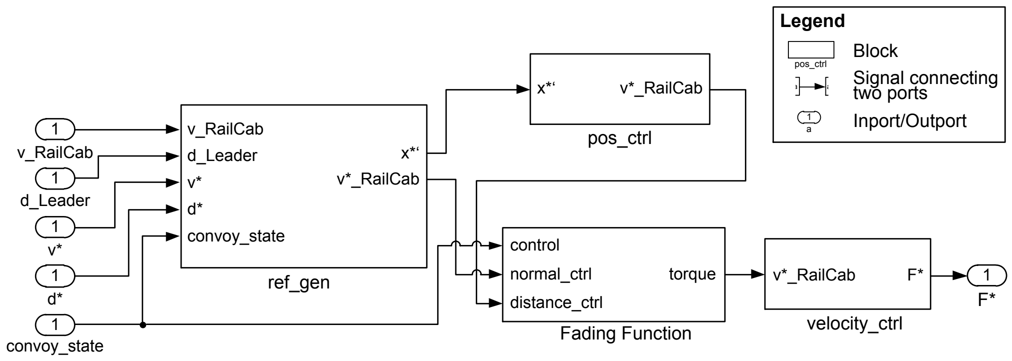


Figure 2.13: Fading between signals with a fading function in the velocity controllers of the RailCab

The control engineers implement these fading functions when designing the controllers. In particular, the duration of such a fading is to be determined. This fading duration also influences the discrete software models: When applying timed model checking methods on the discrete software models, the duration of a transition (which corresponds to such a fading function) is an important factor. Other factors that may influence timing aspects in the discrete software models are worst-case execution times (WCET) of calculations that the control engineers implement in MATLAB/Simulink.

State-based behavior MATLAB also allows implementing state-based behavior using Stateflow[6]. Figure 2.14 shows a Stateflow chart, which is similar to UML statecharts. Stateflow charts have a very limited expressiveness. In particular, it is difficult to specify complex timing constraints, as Stateflow has no clock concept. Furthermore, neither Stateflow nor Simulink have concepts for message-based communiction. For this reasons, these aspects of a system are typically implemented using special software engineering languages, like MechatronicUML, as explained in Sect. 2.1.4.1.

[6]MATLAB/Stateflow website: `http://www.mathworks.com/products/stateflow/`

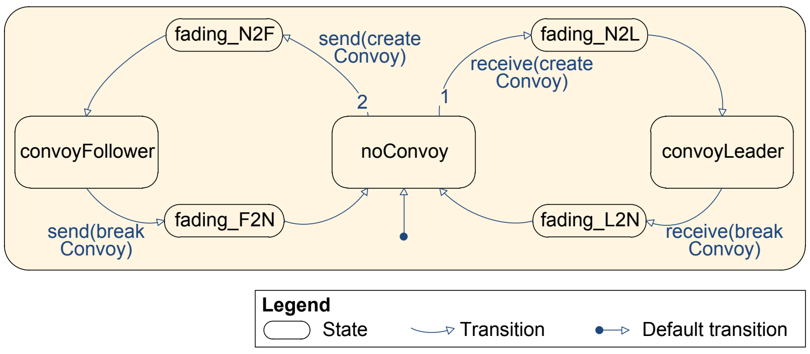


Figure 2.14: MATLAB/SIMULINK chart for controller reconfiguration

2.1.5 Further Disciplines

In this thesis, we focus on the discipline-spanning models of the system and the discipline-specific models of software engineering and control engineering. Other disciplines are typically also involved in the development of mechatronic systems. Most prominent, there are mechanical engineering and electrical/electronic engineering. Both disciplines use different types of modeling languages.

Techniques, tools, and models used in mechanical engineering include (besides others) computer aided design (CAD), finite element analysis (FEA), and computational fluid dynamics (CFD). According to ZORRIASSATINE ET AL. [ZWPG03], the main categories are:

- "mechanical design, e.g. two-/three-dimensional drafting, sketching and solid modelling;
- shape design and styling to address innovative forms and complex shapes such as freeform curves;
- analysis and simulation solutions including stress analysis, design optimization in terms of mass, displacement and principle stresses, and kinematic and dynamic simulation." [ZWPG03]

This constitutes a wide range of models with complex dependencies between them and to the system model. For instance, the mechanical design determines mass distribution and total weight of a system. This, in turn, has an influence on material/stress simulations. Also the system model and other discipline can be affected, e.g., when a control strategy depends on the weight.

In electrical engineering, a common modeling language is circuit diagrams. They are used to describe electrical or electronic circuits from a logical point of view. A circuit diagram abstracts from the real-world appearance of the circuit by using a) simplified standard symbols for the elements, and b) a layout that does not correspond to the physical locations of the elements. The goal is to make engineers recognize the function of the circuit more easily. There are several standards for the syntax of circuit diagrams. Figure 2.15 shows an example of a circuit diagram using the IEC 60617 standard [IEC96]. The underlying physical/electrical laws define the semantics of a circuit diagram. Because circuit diagrams only represent the logical function and abstract from the physical layout, we need further models that represent the physical design. These models also include information like the size and arrangement of wires.

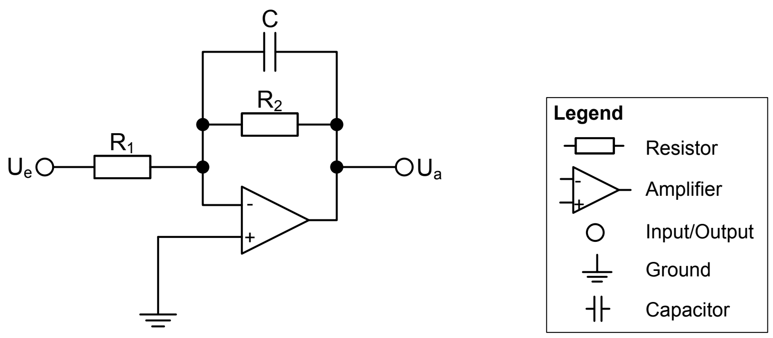


Figure 2.15: Circuit diagram of an active low-pass filter (according to IEC 60617)

2.2 Model Transformations

With the growing importance of model-driven software development approaches, the need for model transformation also increases, since different models have to be mapped onto each other. There exist many different approaches and formalisms for model transformation; moreover, the Object Management Group (OMG) has proposed a model transformation standard [OMG08]. In addition, several model transformation tools have emerged in the last years. Model transformation remains an area with intensive research activities (e.g., see [GRR13, bx12, bx13, EEKR12, DK13, MSG+13]). Next, we introduce the fundamental concepts of model transformations in Sect. 2.2.1 and present a classification of model transformation features in Sect. 2.2.2.

2.2.1 General Concepts and Terms

Figure 2.16 shows the fundamental concepts of model transformation. In general, a model transformation is an operation that takes a model as input and produces a (typically different) model as output. Multiple models are possible as input or output, too.

All the models conform to metamodels. It is possible that the source and target metamodels are the same, and also the source and target models can be identical. The metamodels of the source and target models also serve as a basis for the transformation definition. This means that the transformation definition does not argue on concrete instances (models), but on their modeling concepts (metamodels). We say that the transformation *refers to* or is *typed by* these metamodels. The transformation definition itself also conforms to a metamodel (or grammar).

This definition applies to many different formalisms, approaches, and tools, and they serve many different purposes and applications. Next, we present an

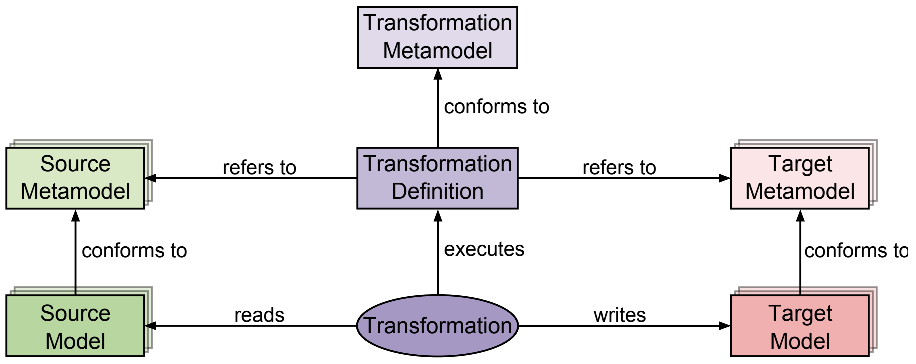


Figure 2.16: Fundamental concepts of model transformation (adapted from [CH06]

approach to classify the different approaches that exist in the field of model transformation.

2.2.2 Feature-Based Classification of Model Transformation Approaches

To structure the large area of model transformations, Czarnecki and Helsen [CH03, CH06] present a classification framework based on feature diagrams. They analyze 30 different model transformation approaches/languages/-tools, and derive a classification scheme from them.

We have adapted this classification in some aspects that we found imprecise or uncovered due to the results of our research. This adapted feature diagram is shown in Fig. 2.17. The features shaded in yellow have been modified or added; the rationale for the modification is explained in the footnotes.

On the top level, Czarnecki and Helsen propose eight features [CH06]. Here, we describe the features that are relevant in the scope of this thesis.

- *Specification:* The way the transformation is defined, e.g., using pre- and post-conditions that are not (immediately) executable or a directly executable specification.
- *Transformation Rules:* Transformation rules form the smallest units of a transformation. For instance, these could take the form of a function or graph rewrite rules.
  - *Domain:* The part of a transformation (rule) that refers to a model. Usually, there are a source and a target domain, which refer to the source and target (meta-)model, respectively. For in-place transformations, there is only a single domain.
    - ∗ *Body:* The body of a transformation rule can contain different constructs. *Variables* hold elements or values from the models. *Patterns* represent model fragments using variables. Depending on the type of model transformation, they can have the form of *string patterns*, *term patterns*, or *graph patterns*. Their representation can be based upon the abstract syntax of the domain's language (e.g., in the style of object diagrams or abstract syntax

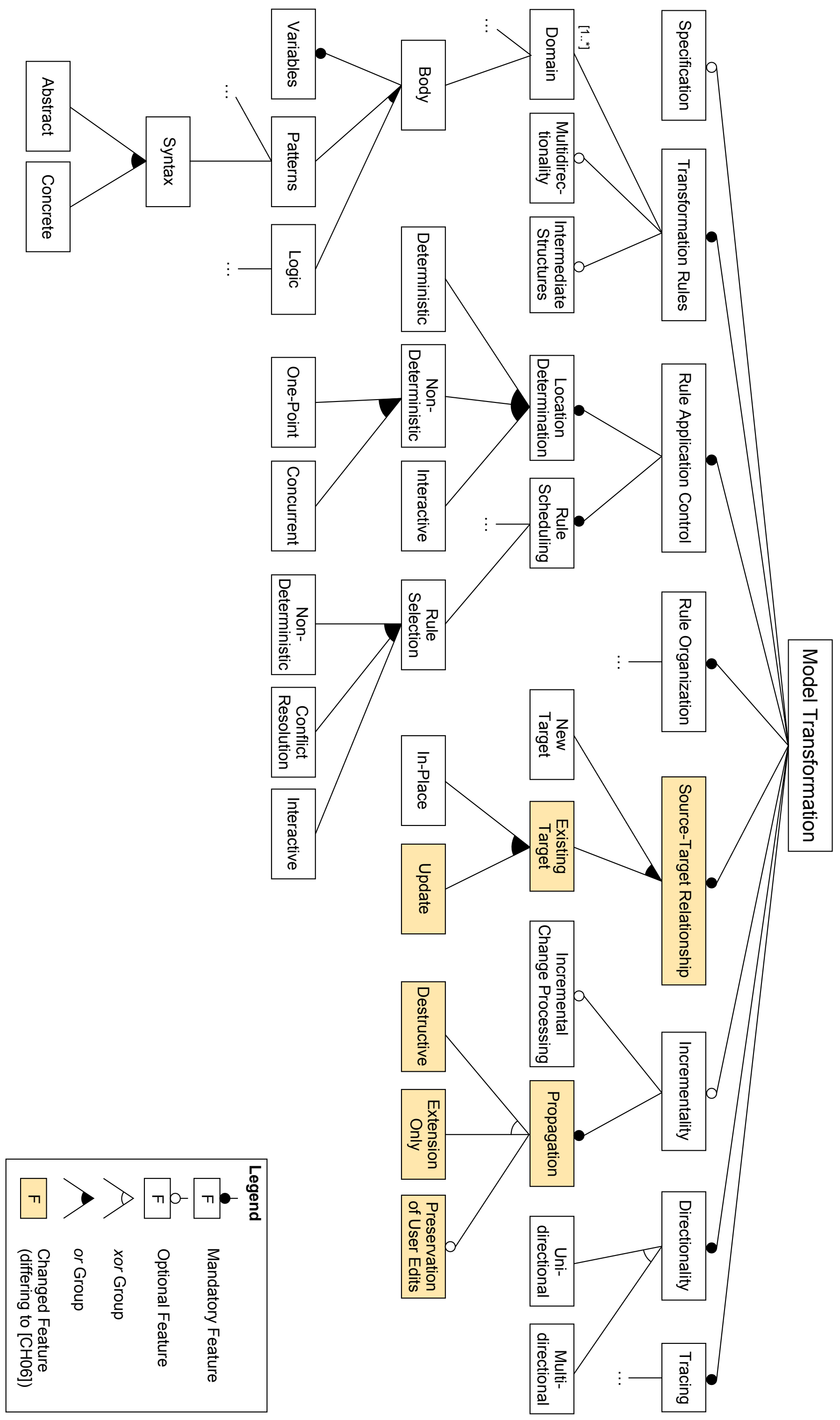


Figure 2.17: Features of Model Transformation (adapted from [CH06])

graphs) or re-use the concrete syntax of the domain's language (which can be graphical or textual). The *logic* feature describes the programming paradigm behind the transformation logic. For instance, a transformation definition can be imperative, functional, or declarative.

  - *Multidirectionality:* Multidirectional rules can be executed in more than one direction, i.e., the same rule can be used for transformations from source to target and vice versa. Typically, multidirectional rules accompany multidirectional transformations (cf. the top-level feature "Directionality" described later); however, a transformation can be multidirectional but have only unidirectional rules.
  - *Intermediate Structures:* A transformation can create intermediate data structures during the transformation run, e.g., to perform calculations that are only needed during the actual transformation.
- *Rule Application Control:* How the approach determines the locations in the model(s) to which transformation rules are applied, and how it schedules the order of the transformation rules.
  - *Location Determination:* In general, there is more than one place in the models where a rule is applicable. Therefore, the transformation algorithm has to decide where to apply a rule. A transformation definition can either deterministically declare where to apply rules or randomly. In addition, a transformation can involve the user in deciding where to apply a rule (interactive). However, unless different rules can be applied for the same model elements ("rule conflict", cf. "Rule Scheduling"), this decision does not influence the transformation result.
  - *Rule Scheduling:* "Scheduling mechanisms determine the order in which individual rules are applied" [CH06]. Most important aspect here is how to select the rule that is applied next (*Rule Selection*). As stated above, there can be more than one rule that is applicable for certain model elements. If applying a rule will render other rules inapplicable, we call this a "rule conflict". A transformation could use conflict resolution strategies like explicit priorities. It can also non-deterministically decide which rule to apply. This potentially lets the transformation become non-functional, as different transformation runs could lead to different results. Also the user could be involved in deciding which rule to apply.
- *Rule Organization:* How rules could be structured or modularized.
- *Source-Target Relationship:* Whether the source and the target models are the same, and whether the transformation creates a new target model.
  - *New Target:* A transformation creates a new target model. This is typically called a *batch* or *initial transformation*.
  - *Existing Target:* An already existing target model may be processed by the transformation as follows.

    * *In-Place:* If source and target model are the same, this is called an in-place transformation.[7]
    * *Update:* A transformation changes the existing target model, typically in an incremental transformation when it propagates changes (cf. "Incremental Change Processing")[8].

- *Incrementality:* An incremental approach can update an existing model with changes that occurred in another model. Typically, the model that was changed is called the source model, and the updated model is the target model. However, changes may also happen to both models simultaneously. Not every model transformation approach/tool supports incremental updates – a transformation tool can also completely rerun the transformation and create new (target) models afresh.
  - *Incremental Change Processing*[9]*:* When a model that is involved in the transformation changes, these changes have to be processed by the transformation engine. Incremental change processing means that the engine only processes model parts that are affected by the changes, but not the whole model that has changed.
  - *Propagation*[10]*:* Changes have to be propagated to other models that are part of the transformation. A *destructive* propagation mechanism will also delete target model parts if they have no corresponding parts in the source model any more. In contrast, *extension only* will just propagate new elements. Transformation engines differ in their strategies to *preserve user edits*; for instance, target model elements that have been edited by the user can be ignored during propagation.
- *Directionality:* A *unidirectional* transformation can only be executed from source to target, whereas *bi- or multidirectional transformations* can exchange the notion of source and target.
- *Tracing:* Whether the mapping between the source and the target model elements (i.e., which source element(s) correspond to which target element(s)) is explicitly stored in a trace model.

We refer to Czarnecki and Helsen [CH06] for a detailed description of the remaining features. We use these features to classify our approach and compare it to related work (cf. Sects. 2.4.4, 4.1.1, 4.2.1, and 4.3.2).

2.3 Graph Grammars and Graph Transformations

Many model transformation techniques base on the concept of *graph transformations*. *Graphs* are the underlying data structure on which graph transformations

[7]Note that a transformation engine incapable of performing in-place transformations can emulate them by copying the source model to the target model and altering it. However, we consider such a transformation as a "New Target" transformation.

[8]The subfeatures "Destructive" and "Extension Only" have been moved to the "Incrementality—Propagation" feature, because they do not describe the relation between the source and the target model, but how propagation of changes works.

[9]Originally, this feature was called "Source Incrementality". To avoid confusion with the typical notion of incremental updates and to reflect that changes may happen both in source and target model simultaneously, we renamed it to "Incremental Change Processing".

[10]Equivalently, "Target Incrementality" was renamed to "Propagation".

work. Before discussing graph transformations in detail in Sect. 2.3.2, we first give a brief introduction to graph theory.

2.3.1 Graphs

A graph is a data structure that consists of *nodes* and *edges*, where each edge connects two nodes [ERD+97, EEPT06]. When working with models in terms of MOF [OMG06], graphs can be used as a representation for object structures. Then, graphs are typically *typed* and *attributed*. In typed graphs, nodes and edges have a type associated, which is defined in an additional *type graph*. I.e., this type graph contains different possible types for nodes and edges; additionally, it constrains how these types can be used in combination. In attributed graphs, nodes can also have attributes. In terms of MOF, a type graph together with the attributes is a *metamodel* that contains classes (i.e., types for nodes), which may have references (types for edges) and attributes (attributes for nodes).

Here, we follow the definitions of Ehrig et al. [EEPT06], modified and extended for our needs.

Definition 1 (graph)**.** A *graph* G is defined as $G = (V, E, s, t)$, where

- V is the set of vertices,
- E is the set of edges, and
- $s, t : E \to V$ are the source and target functions of edges, i.e., iff $e \in E, v \in V : (e, v) \in s$ then the edge e has the vertex v as a source; t respectively.

As noted above, in the context of MOF models we usually deal with attributed and typed graphs. A typed graph is typed over a *type graph*.

Definition 2 (type graph)**.** Let P be the set of primitive attribute types, and let $M = (M_V, M_E, M_A)$ be a mark alphabet over vertices, edges and attributes. M_V contains the class names and M_E the names of the references. M_A contains the attribute names. A *type graph* T is defined as $T = (V_T, E_T, s_T, t_T, c_T, r_T, a_T)$, where

- V_T is the set of vertices (i.e., “classes” in terms of MOF),
- E_T is the set of edges (i.e., “references”),
- $s_T, t_T : E \to V$ are the source and target functions of edges, i.e., $e \in E, v_1, v_2 \in V : (e, v_1) \in s_T \land (e, v_2) \in t_T$ iff the reference e has the class v_1 as a source and the class v_2 as target,
- $c_T : V_T \to M_V$ is the class naming function, i.e., $v \in V_T, m \in M_V : (v, m) \in c_T$ iff the class v has the name m,
- $r_T : E_T \to M_r$ is the reference naming function, i.e., $e \in E_T, m \in M_E : (e, m) \in r_T$ iff the reference e has the name m, and
- $a_T : V_T \to (M_A \times P)$ is the attribute function, i.e., $v \in V_T, m \in M_A, p \in P : (e, m, p) \in a_T$ iff the class v has an attribute with the name m and the primitive type p.

Usually, we also want to include inheritance relations to the type graph, such that an object is also an instance of the superclasses of its class, and that references and attributes of a superclass are also contained in the subclasses. As Ehrig et al. [EEPT06, pp. 259–281] show, each graph transformation and grammar that is based on a type graph with inheritance can be mapped to an equivalent transformation/grammar without inheritance. Thus, we use type graphs with inheritance for illustration and simplification, but formally argue only on type graphs without inheritance.

Definition 3 (typed and attributed graph). Let $T = (V_T, E_T, s_T, t_T, c_T, r_T, a_T)$ be a type graph. A *typed and attributed graph* G_T is defined as $G_T = (V, E, s, t, c, r)$, where

- V is the set of vertices (i.e., "objects" in terms of MOF),
- E is the set of edges (i.e., "links"),
- $s, t : E \to V$ are the source and target functions of edges, i.e., $e \in E, v_1, v_2 \in V : (e, v_1) \in s \wedge (e, v_2) \in t$ iff the edge e has the node v_1 as a source and the node v_2 as target,
- $c : V \to V_T$ is the vertex typing function, i.e., $v \in V, v_T \in V_T : (v, v_T) \in c$ iff the vertex v is of type v_T,
- $r : E \to M_r$ is the edge typing function, i.e., $e \in E, e_T \in E_T : (e, e_T) \in r$ iff the edge e is of type e_T.

Figure 2.18 shows a type graph with inheritance. It consists of three classes A, B, C with references (x and y, respectively). Class C has an attribute id of the simple data type int, and there is an inheritance relation between classes B and A.

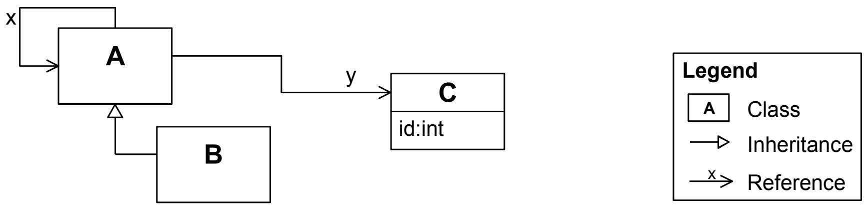


Figure 2.18: A simple type graph in class diagram syntax

The graph shown in Fig. 2.19 is typed over this type graph. You see that the nodes and edges are labeled with the classes and references of the type graph, respectively. Additionally, node :C has a attribute id with a value of 42.

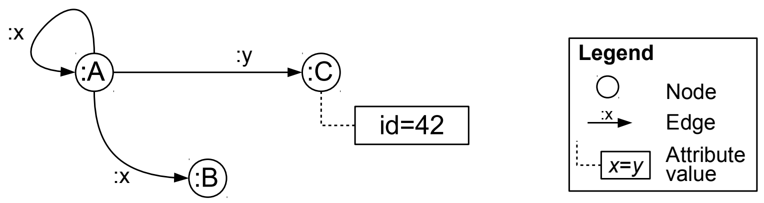


Figure 2.19: A simple graph typed over the type graph of Fig. 2.18

In general, a MOF object structure can be represented by a graph. For instance, the graph in Fig. 2.19 is an equivalent representation for the object diagram shown in Fig. 2.20.

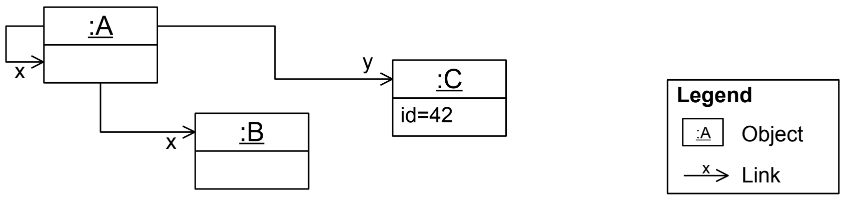


Figure 2.20: Object diagram of Fig. 2.19

In this way, graph transformation techniques for typed and attributed graph grammars can be applied to object structure and models. We therefore do not distinguish between graph transformation and model transformation in this thesis.

Next, we describe how graph transformations can be used to modify graphs.

2.3.2 Graph Transformations

Modifications to graphs can be specified using *graph transformation rules* (also called *graph rewrite rules* or *production rules*). When the host graph is typed and attributed, the graph transformation rule can also be typed and attributed.

Figure 2.21 shows a typed graph transformation rule. Similar to string rewrite rules, such graph transformation rules consist of a left-hand side (LHS) and a right-hand side (RHS). The LHS defines the precondition that must be fulfilled before applying the rule. The RHS describes the post-condition, i.e., how the host graph must look like after applying the rule. Furthermore, there is a morphism from the LHS to the RHS that defines which nodes represent the same object. This morphism is also called *glue graph*, as it glues together LHS and RHS.

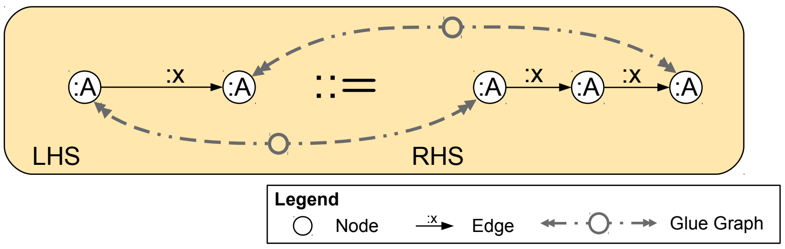


Figure 2.21: A simple typed graph transformation rule

Definition 4 (morphism)**.** A *morphism* is a structure-preserving mapping from one structure to another. The source structure is called *domain*, the target structure *codomain*. A morphism f with domain X and codomain Y is written as $f : X \rightarrow Y$.

When working with graphs, a morphism maps a graph onto another graph. That means we map all vertices and edges of the domain to vertices/edges in the codomain:

Definition 5 (graph morphism)**.** Let G_1, G_2 be two graphs. A function $f := (f_v, f_e)$, $f_v : V_1 \to V_2, f_e : E_1 \to E_2$ that maps all vertices and edges of G_1 onto G_2 is a graph morphism iff $\forall e \in E_1 : f_v(s(e)) = s(f_e(e)) \wedge f_v(t(e)) = t(f_e(e))$.

There are several special types of morphisms. A *monomorphism* is injective, i.e., it always maps distinct elements of the domain to distinct elements of the codomain. An *isomorphism* is bijective, i.e., it builds a complete set of unique pairs between the domain and the codomain. In contrast to a monomorphism, there must not be edges between vertices of the codomain that have no corresponding edge in the domain.

Definition 6 (graph transformation rule)**.** A *graph transformation rule p* is a pair of graph morphisms $(L \xleftarrow{l} K \xrightarrow{r} R)$, where L, R are typed graphs as in Def. 3 (the LHS and RHS) and K is the glue graph.

Intuitively, you can think of the morphisms l and r as inclusions, i.e., $L \supseteq K \subseteq R$. With l and r, K glues the corresponding elements of L and R together.

A graph transformation rule is applied as follows. First, we try to find a matching of L in the host graph H, i.e., we match L with a subset of H (*morphism*). Second, we replace L by R in the host graph H. In the following, we formally define these two steps.

Definition 7 (matching)**.** A *matching f* is a morphism between the LHS graph G and the host graph H, i.e., there exists a function $f := (f_v, f_e)$, $f_v : V_G \to V_H, f_e : E_G \to E_H$ such that $\forall e \in E_G : f_v(s(e)) = s(f_e(e)) \wedge f_v(t(e)) = t(f_e(e))$. When matching typed graphs, we further require that $\forall v \in V_G : c(v) = c(f_v(v)) \wedge \forall e \in E_G : r(e) = r(f_e(e))$.

Intuitively, that means that a valid matching ensures that a) each node in the graph G must be mapped to a node in the host graph that has the same type, b) the edge structure between the nodes must be preserved, and c) the edges' types must also be the same.

Typically, a matching is a *monomorphism*, i.e., f is a injection. Such a matching is shown in Fig. 2.22. However, such a matching can also just be a morphism, i.e., f can be not injective. Such a matching is depicted in Fig. 2.23, where both nodes :A of the LHS are matched to the same node :A in the host graph.

As most graph transformation formalisms use an monomorphic matching, whenever we speak of "matching" in this thesis, we refer to an monomorphic matching unless stated otherwise. Monomorphism as well as subgraph isomorphism is an NP-complete problem [Coo71, GJ90] in general. However, "both the enumeration and decision problems can easily be solved in polynomial $O(n^l)$ time" for fixed patterns with l vertices (and n host graph vertices) [Epp95].

2.3.3 Graph Grammars

A graph transformation rule, as described in the previous subsection, defines how to modify a host graph that satisfies a certain (pre)condition. A set of such graph transformation rules together with a start graph form a *graph grammar*.

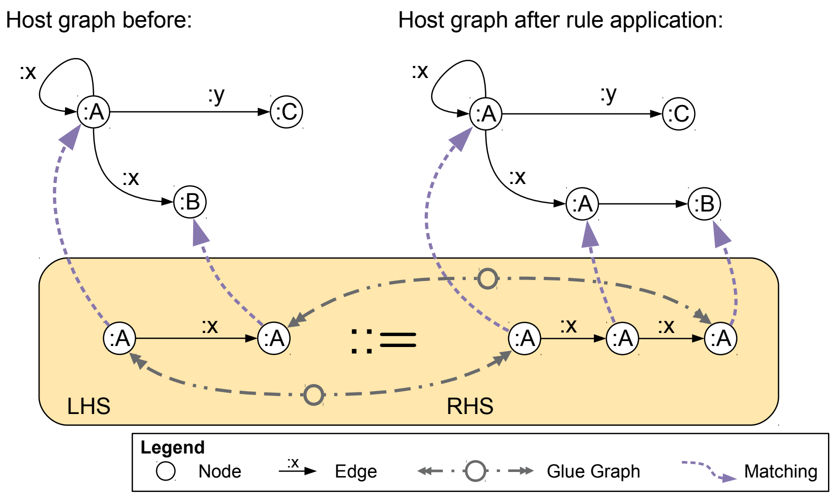


Figure 2.22: Finding an monomorphism ("matching") and applying a graph transformation rule

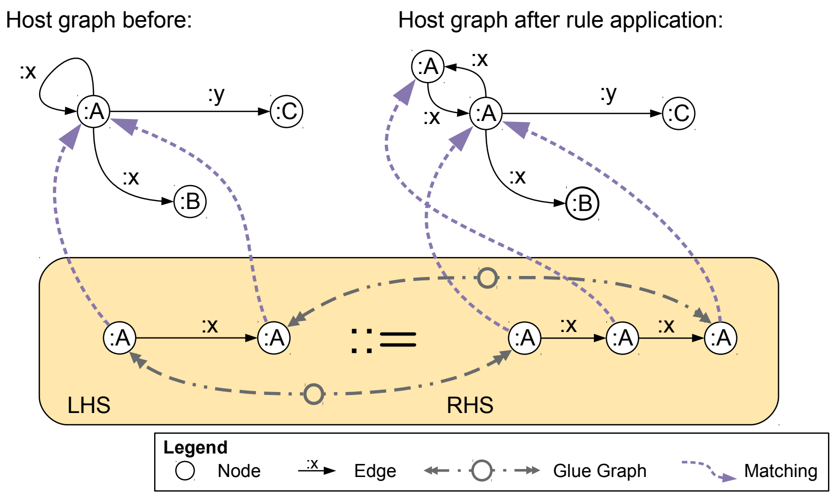


Figure 2.23: Matching with a non-injective morphism

Definition 8 (graph grammar)**.** A *graph grammar* is a pair $\mathcal{G} = (G_0, \mathbf{P})$ where $G_0 \in G$ is the start graph and $\mathbf{P} = \{p : (L \xleftarrow{l} K \xrightarrow{r} R)\}_{n\in\mathbb{N}}$ is a set of graph transformation rules.

Similar to a string grammar, a graph grammar $\mathcal{G}$ defines the language $L(\mathcal{G})$, which is the set of graphs that can be derived from G_0 using $\mathbf{P}$. Thus, a graph is a word of the graph language if and only if it can be derived from the start graph using the graph transformation rules of the graph grammar.

Definition 9 (word of a graph language)**.** A graph g is $\in L(\mathcal{G})$ iff

$$\exists(p_1, p_2, ..., p_n \in \mathbf{P}) : G_0 \xrightarrow{p_1} G_1 \xrightarrow{p_2} ... \xrightarrow{p_n} g$$

In other words, a graph grammar is a constructive way to describe a graph language: Consecutively applying its graph transformation rules, starting with the start graph, only words of the graph grammar are created.

Identifying graph derivation sequence To find out whether a given graph g is contained in the language induced by a graph grammar, we have to find out whether it if possible to derive g from the start graph. Similar to string grammars, this is a complex problem in general (depending on the class of the grammar). We have to explore all possible rule applications sequences that can possibly lead to g. Basically, this means that we have to compute a derivation tree: For every alternative rule that we can apply in a certain situation, we have to branch and – in the worst case – compute all respective subtrees. Typically, this is performed using a heuristics-guided depth-first search. If this depth-first search reaches a leaf in the tree, and this tree does *not* lead to the given graph g, we have to backtrack in the tree, computing other alternative sequences.

For string grammars, we know several restrictions (like LR(k) or LL(k) grammars) that make the problem practically solvable. Similarly, to allow finding a derivation sequence for large graphs in practice, a graph grammar can be restricted in different ways. On the other hand, only allowing a restricted class of graph grammars reduces expressiveness.

No conflicting rules: For every step of the derivation sequence, there is never more than one rule that can be applied to a certain element. In other words, there are no conflicting rules[11]. This means that an algorithm that tries to determine the graph derivation sequence that leads to g always knows which rule to apply next. As a result, the computation tree is a simple linear list; no backtracking is required.

Maximum backtracking depth: Restricting the maximum depth of necessary backtracking provides a compromise between expressiveness and computation complexity. Assume a maximum backtracking depth of bd. This means that after bd further rule applications after the application of rule R, it must be certain whether rule R will eventually lead to g. $bd = 0$ is the same as "no conflicting rules".

Monotonic productions: If all rules in a graph grammar only produce vertices and edges and do not delete them, all rules are *monotonic productions*; we call such a ruleset *monotonic*. Such a ruleset also reduces the computational complexity, because the derivation search algorithm knows when to stop: If the size of the (intermediate) derived graph exceeds the size of the graph g, applying further rules will never result in g.

[11]Note that most related work call these rule application conflicts simply *conflicts*. In this thesis, we are also dealing with *editing conflicts* that occur when different developers contradictorily change the same model element. Thus, we explicitly distinguish between *rule (application) conflicts* and *editing conflicts* where necessary for clarification.

In this case, "a given graph directly contains all necessary information about its derivation history, and graph parsing simply means covering a given graph with right-hand sides of productions" [Sch95]. However, it still may require to compute the whole derivation tree up to the size of the host graph (cf. Fig. 2.24). This has a worst-case runtime of $O(n^{k \cdot l})$, where n is the size of the host graph, k is the number of rules, and l is the maximum size of the rules. When there are no conflicting rules, this is reduced to $O(k \cdot n \cdot n^l)$, because we always know which rule to apply, and only the pattern matching has to be performed O($k \cdot n$) times. Assuming that both k (the number of rules) and l (the maximum size of the rules) are fixed, the time complexity is polynomial in the size of the input host graph n.

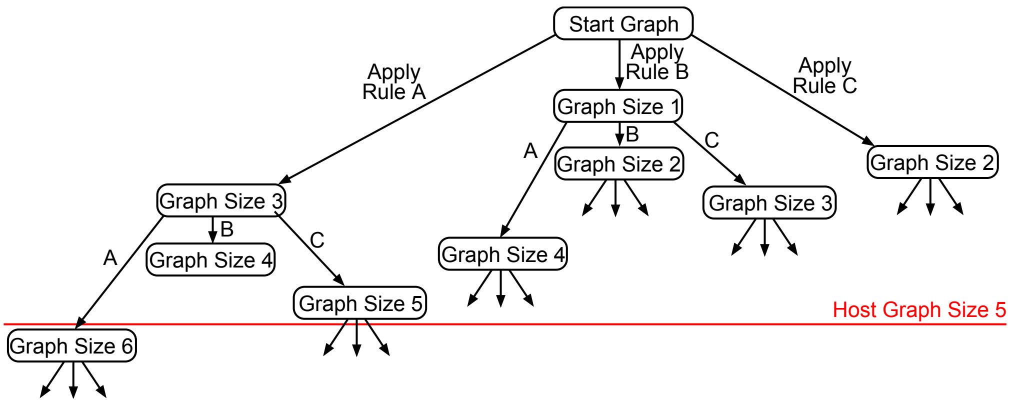


Figure 2.24: Derivation tree created by sequentially applying rules

Triple Graph Grammars, the model transformation approach used in this thesis, only uses monotonic productions to allow efficient graph transformation algorithms. Furthermore, most TGG implementation do not allow conflicting rules. We will give details on Triple Graph Grammars in the next section.

2.4 Triple Graph Grammars

The section describes the syntax and semantics of *Triple Graph Grammars* (TGGs), which are used as the model transformation technique throughout this thesis.

Triple Graph Grammars have been introduced by Schürr [Sch95] in 1994 as an extension to pair grammars. They are a rule-based, declarative formalism that allows specifying how two models evolve in parallel. Every TGG rule consists of two graph grammar rules which describe how each of the models can be modified. These two rules are linked by third, a so-called *correspondence graph* grammar rule, which allows to explicitly store the links between the two models.

TGGs can be used for different model transformation and synchronization scenarios. More specifically, TGGs can be used for bidirectional transformation, i.e., to produce a target model from a given source model and vice versa. Furthermore, they can be applied to *synchronize* the models in situations where a pair of corresponding models exists and changes to one of the models occur.

To execute model transformations specified by TGGs, we developed the TGG Interpreter tool suite. It is a set of Eclipse plug-ins based on the Eclipse Modeling Framework (EMF). It consists of a graphical specification tool for TGGs and a TGG execution engine. Details of the implementation can be found in Sect. 6.

Before we describe how TGGs can be applied to transform and synchronize models in Sect. 2.4.2 and 2.4.3, we explain the basic syntax and semantics of TGGs in the next section.

2.4.1 Basic TGG Syntax and Semantics

Figure 2.25 shows a TGG rule using the traditional graph production syntax. This rule is part of the TGG ruleset that defines the mapping between a CONSENS system specification and a MechatronicUML software model.[12] TGG rules are non-deleting graph grammar rules that consist of a left-hand side (lhs) and a right-hand side (rhs). TGG rules can also be represented using a compact notation, where left-hand side and right-hand side are combined into a single pattern. Fig. 2.26 show this notation; for brevity, this notation is also used in the further course of this thesis.

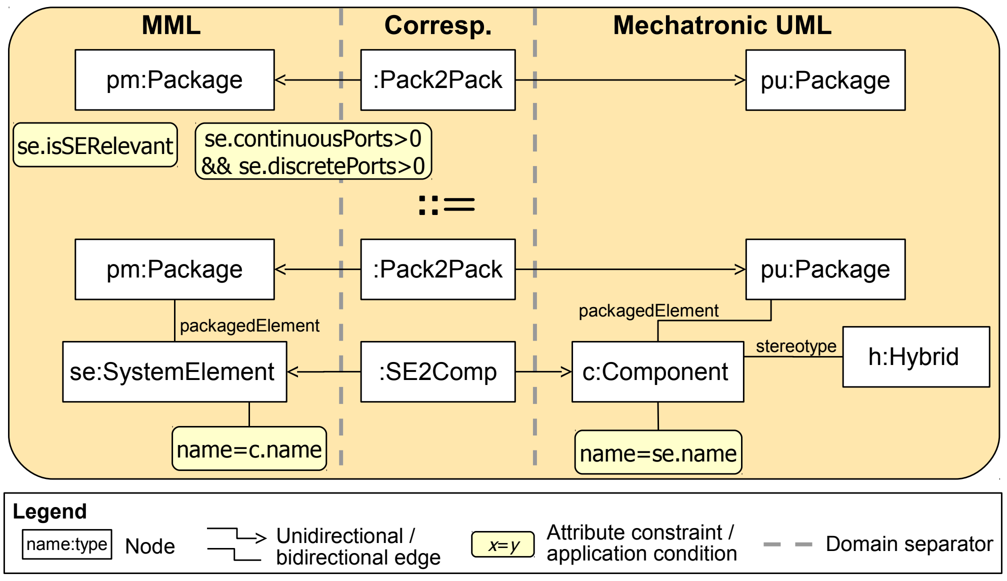


Figure 2.25: TGG Rule SystemElement2HybridComponent: relating system elements (CONSENS) to hybrid components (MechatronicUML)

Basically, a TGG rule describes how two models satisfying a certain precondition can evolve simultaneously. In terms of graph grammars, this precondition is called *context graph* (or lhs) of the rule. Accordingly, the nodes in

[12]This rule is slightly simplified for illustration purposes. The complete TGG ruleset for the transformation, including the full version of this rule, can be found in Appendix A.

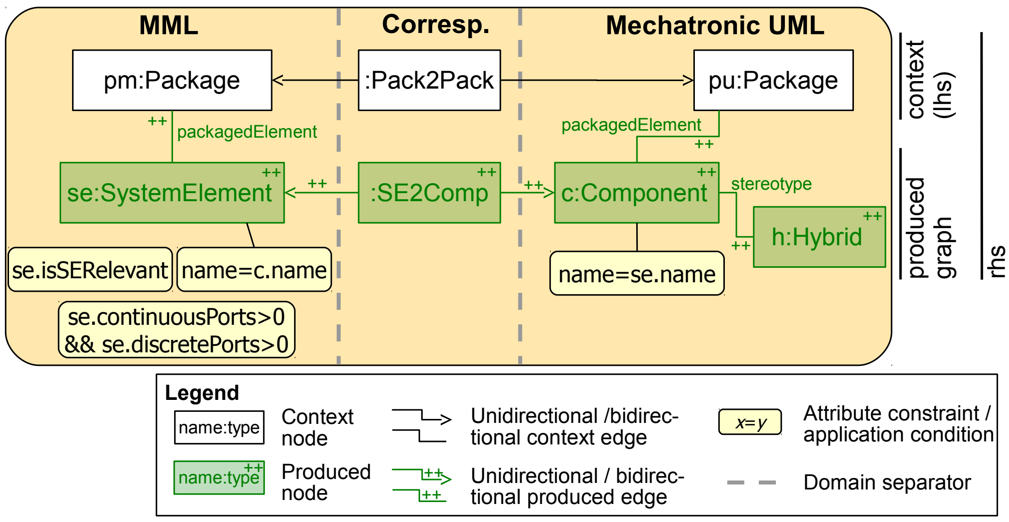


Figure 2.26: Compact notation of the TGG Rule SystemElement2HybridComponent (cf. Fig. 2.25)

the context graph are called *context nodes* (displayed as white boxes with black borders), the edges are called *context edges* (displayed as black arrows or connections). In Fig. 2.26, there are three context nodes (pm:Package, :Pack2Pack, and pu:Package) and two context edges (the edges originating in the context correspondence node :Pack2Pack).

Whenever there is a valid *matching* for the context graph, i.e., it can be found in the models, the rule can be applied. This is done by creating the so-called *produced graph* of the rule. The produced part contains everything from the rhs that is not element of the lhs. Elements of the produced graph are called *produced nodes* and *produced edges*, respectively, and are displayed in green color and marked with a "++".

The columns in a TGG rule are called *domain*. Each domain contains one of the three graph grammar rules for the three models. Usually, the domain on the left is called *source* and the domain on the right *target*. The *correspondence* domain links the source and target domain.

Each domain is assigned a metamodel. The nodes of this domain are typed over the classes in this metamodel, the edges over the respective references of the classes.

In Fig. 2.26, you can further see *attribute constraints* and *application conditions*, both depicted as yellow rounded rectangles.

Attribute constraints constrain the attribute value of objects that are matched by the corresponding node (denoted by an arrow). They are given in the form ⟨*prop*⟩ = ⟨*expr*⟩, where ⟨*prop*⟩ is a property of the node's class. We use OCL [OMG12] as an expression language in our TGG INTERPRETER. Thus, ⟨*expr*⟩ is an OCL expression whose result must conform to the type of ⟨*prop*⟩. When a TGG rule is applied, it must be ensured that the expression's result is equal to the value of the attribute. If this is not the case, the rule must not be applied. For instance, in Fig. 2.26, the constraint `name=c.name` ensures that the name of the object that is matched by node se:SystemElement in CONSENS must be equal to the name of the object of node c:Component in MechatronicUML.

Application conditions allow restricting the application of a rule to certain conditions. They are also specified as OCL expression that must evaluate to a boolean value. Before applying a TGG rule, it the application condition is evaluated. The rule may only be applied if the application condition holds. For instance, in Fig. 2.26, the application condition `se.continuousPorts>0 && se.discretePorts>0` ensures that the rule is only applied in situations where at least one continuous port and one discrete port exists in the matched system element. Note that application conditions are not invariants. Thus, they must only hold during rule application, and not during later transformation stages. See Sect. 5.1 for further details.

Like every graph grammar, a TGG also contains a start graph. This so-called *axiom* is the starting point for the derivation of all consistent model pairs. That means that a pair of two models is consistent in terms of the TGG if and only if there is a chain of productions applied on the start graph that produces this models pair. More formally, given a set of TGG rules M and an axiom A, a pair of two models $H = H_1 \times H_2$ is consistent iff

$$\exists p_1, p_2, \ldots \in M : A \xrightarrow{p_1} G_1 \xrightarrow{p_2} G_2 \xrightarrow{\ldots} H$$

As described in Sect. 2.1.4.1, MechatronicUML distinguishes between software components, hybrid components, and controllers. The underlying idea of the transformation is that every system element that has only continuous incoming or outgoing information flows becomes a controller component, which is implemented in control engineering. System elements with discrete information flows become software components, and system elements with both continuous and discrete flows become hybrid components. The rule shown in Fig. 2.26 is for the third case, i.e., mapping system elements with both continuous and discrete flows to hybrid components. Figure 2.27 shows another TGG rule from the CONSENS-MechatronicUML TGG ruleset. It deals with the first case, i.e., mapping system elements with only continuous flows to controller components.

2.4.2 Model Transformation with TGGs

In its default graph-transformation semantics described in the previous section, TGGs produce two consistent graphs in parallel. However, TGGs are mostly used in model-to-model transformations scenarios where a target model is created afresh from an existing source model. The general idea to apply TGGs in such a scenario is to find a chain of TGG rules whose source production rules would produce the source model. If such a chain of rule applications exists, you can create the corresponding correspondence and target models by also applying the correspondence and target production rules. By following this approach, only valid consistent model pairs are created.

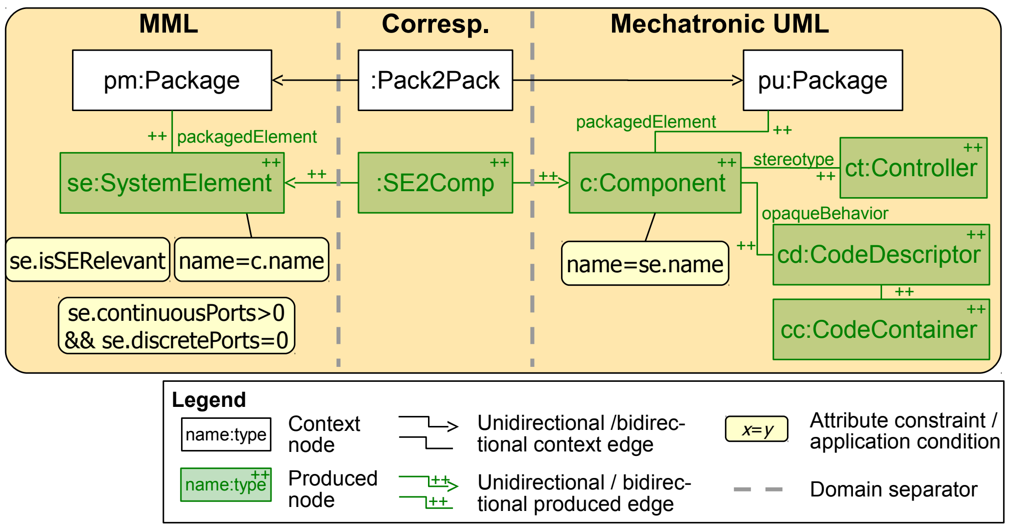


Figure 2.27: TGG Rule SystemElement2Controller: relating system elements (CONSENS) to controller components (MechatronicUML)

2.4.2.1 Application Scenarios

We can interpret TGGs for different *application scenarios*.

The application scenario most frequently used is called *forward transformation*, in which case we create a "target" model corresponding to a given "source" model. In this case, TGG rules are interpreted as follows: First, the context pattern and the produced source domain pattern of the rule are matched in the (source, target and correspondence) models, adhering to the following conditions. The context pattern of the rule must only be matched to *bound* model elements, which are objects and links that were previously matched by a produced node of another rule application. The source produced pattern is matched to yet *unbound* parts in the source model, i.e., elements that have not been covered by previous rule applications. If a matching respecting these conditions is found, we create the produced target and correspondence patterns, and we create bindings for all newly matched and created elements.[13]

The *backward* direction works accordingly, reversing the notion of source and target.

If both the target and the source model are given, we can identify the *correspondence* between them. In this application scenario, the source *and* target produced pattern is matched to yet *unbound* parts in the source or target models. Only elements for the correspondence graph are created. As a result, given two corresponding models, we have created the missing correspondence graph. If the source and the target models are not corresponding according to the TGG, some corresponding model parts are linked with the correspondence graph.

[13]In the following, we use the term *binding* when referring to a single node-to-object or edge-to-link match, and *matching* for a set of those bindings (i.e., when a whole pattern is matched to several elements).

Fig. 2.28 shows the binding semantics of TGG nodes in the different application scenarios. We refer to Greenyer and Kindler [GK10] for further details on TGGs and the binding semantics that we use in this thesis.

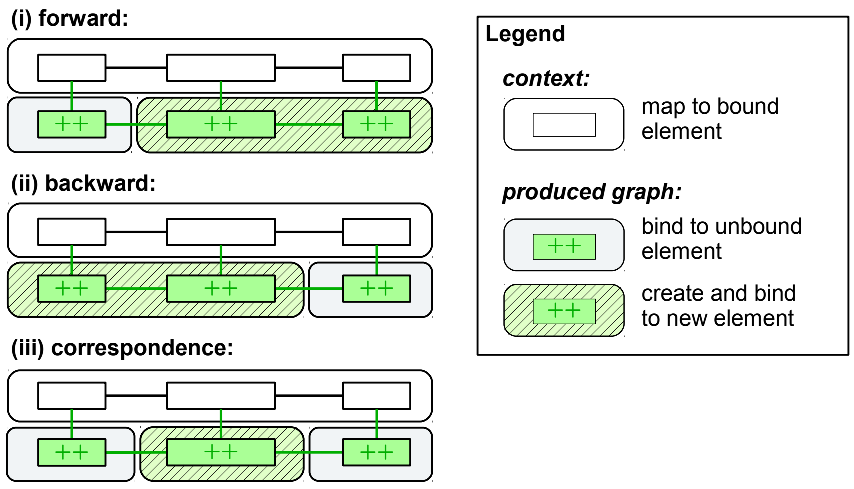


Figure 2.28: The binding semantics of the TGG nodes in the application scenarios

2.4.2.2 Formal Properties

There are several formal properties for transformation algorithms and TGGs. Here we focus on the properties that are relevant for this thesis.

Properties of a Transformation Algorithm A transformation algorithm should have certain properties that ensure that it works as expected. The definitions we use in this thesis are adapted from Hildebrandt et al. [HLG⁺13].

The most important property is *correctness*:

Definition 10 (correctness of a transformation result)**.** A result of a TGG transformation is *correct* iff the resulting model triple (consisting of the already existing source model, the newly generated correspondence model, and the newly generated target model) can also be produced by a sequence of TGG rule applications starting from the axiom.

In other words, this means that a transformation result is correct if it is an element of the language that is defined by the TGG. This definition can be extended for all transformation results that a transformation algorithm produces:

Definition 11 (correctness of a transformation algorithm)**.** A TGG transformation algorithm is *correct* iff all model triples that it will produce can also be produced by a sequence of TGG rule applications starting from the axiom.

Schürr [Sch95] shows that interpreting a TGG for model-to-model transformation in the way described above ensures correctness.

In several practical cases, requiring strict correctness can be too restricting: When transforming between two models, the aspects or viewpoints that these two models contain may differ. Therefore, only parts of the information that both models contain overlap. Consequently, the model transformation also only maps between these overlapping parts, but does not consider *model-specific information.* If, for instance, the source model contains information that is not subject to the transformation, the complete source model could never be produced by applying rules of the TGG.

Another relevant formal property is *completeness.*

Definition 12 (completeness)**.** A TGG transformation algorithm is *complete* iff it is able to derive the consistent correspondence and target models C, T for the given source model S for every model triple $A \in (S \times C \times T)$.

Basically, this property describes how powerful the transformation algorithm is: A complete algorithm is able to transform every valid source model correctly, regardless how difficult it is to find the correct TGG rule sequence to derive the source model.

As already described in Sect. 2.3.3, finding such a rule sequence is a complex problem. TGGs only produce vertices and edges and do not delete them; they are “production-only rulesets” by construction. Although this reduces the complexity with no severe impact on the expressiveness [Sch95], it still may require computing the whole derivation tree up to the size of the host graph (cf. Fig. 2.24). At worst-case, this requires $O(n^k)$ steps, where n is the size of the host graph and k is the number of rules. In each of the steps a monomorphism (pattern matching) has to be found ($\mathrm{O}(n^l)$, where l is the size of the rule). Thus, the overall worst-case runtime is $O(n^{k \cdot l})$.

Transformation approaches and tools therefore have to find a reasonable compromise between expressiveness on the one hand and efficiency on the other hand. Most transformation approaches therefore restrict the class of TGGs they support [HLG+12], i.e., they require a TGG to fulfill certain properties.

Properties of a TGG The TGG itself also has certain properties, which are independent from the transformation algorithm that is used.

A property that is often required by transformation algorithms is *functional behavior.*

Definition 13 (functional behavior of a TGG)**.** A TGG has functional behavior if for every source model S there exists at most one rule sequence (besides permutation) that produces this source model.

If there are different rule sequences leading to a single source model, these alternative rule sequences could possibly produce a differing result on the target side. This means that transformation would not be a function, as it has more than one result for a single input.

A TGG is likely to be non-functional if – in a forward transformation – there is more than one rule that is applicable for the same source model element. Such a condition is called *rule application conflict*. Rule application conflicts can be identified using *critical pair analysis* [HEGO10, HEOG10].

We must distinguish functional behavior of a TGG from functional behavior of a transformation algorithm. Most transformation algorithms will always produce the same result even if the TGG is non-functional. That means, even if more than one rule is applicable, the transformation algorithm will always select the same. This is due to the deterministic design of most algorithms. However, using the same TGG with another transformation algorithm is likely to produce a different target model.

Most transformation algorithms do not support backtracking over rules, i.e., they do not compute the complete rule derivation tree, but just run until they reach the first leaf. Especially, they do not search for another derivation path even if the computed leaf does not correctly produce the source model. Using such an algorithm requires the TGG to have no conflicting rules (cf. Sect. 2.3.3, "conflict-freeness" [GHL14]), which we call *local functional behavior*:

Definition 14 (local functional behavior of a TGG)**.** A TGG has local functional behavior if for every source model S, there exists at most one rule that translates a source model element in its respective context.

Given a TGG with local functional behavior, every rule derivation tree will be a linear list. I.e., a transformation algorithm cannot make wrong choices when computing the derivation sequence of the source model and, thus, will never run into a "dead end".

There are also relaxed forms of local functional behavior. For instance, eMoflon supports TGGs with a so-called look-ahead of 1 by checking for "edges that can no longer be translated if a wrong choice is made (Dangling Edge Check)" [HLG+13]. This is equivalent to allow backtracking one step ($bd = 1$, cf. Sect. 2.3.3).

2.4.3 Incremental Updates and Model Synchronization

After transforming a source model into a target model, changes to these models may occur. In cases of a strictly linear development process like a waterfall-like process model without any loops, this does not cause any problems. However, in practice, such linear processes are almost non-existing. Not synchronizing such changes between the different models may at best cause the documentation to be outdated (if models are used for documentation purposes only). Worse, it causes bugs and faults in the developed product that are detected not before integration tests, or at worst, when the system is already deployed. Thus, we want to propagate such changes to the other models so that the models are consistent to each other again.

Assume we have a source model and run a transformation to create the target model. Both models are consistent according to the transformation. Next, changes to one or both models occur. If only one model has changed, we can *incrementally update* the other model with these changes, which is described in Sect. 2.4.3.1. If both models have been changed, we speak of *simultaneous bidirectional synchronization*, as explained in Sect. 2.4.3.2

2.4.3.1 Incremental Updates

Assume that we have transformed a source model into a target model using a certain TGG. As the transformation is correct, the source and the target model are consistent to each other in terms of the TGG then. If changes occur to one of these models after the transformation, these changes may cause that the models are not consistent any more.

In such a situation, we often would like to restore the consistency between these models. A simple approach is to delete the unchanged model and re-create it using the changed model. If, for instance, the target model was changed, we delete the source model and run the transformation in backward direction with the changed target model as input. The resulting new source model will now contain all changes from the target model.

Such an approach has two main disadvantages. First, if only small parts of the target model were changed, we still run a complete transformation, although we could reuse the results of the first transformation run to a large extend. Second, there may be parts of the source model that are not subject to the transformation (cf. Sect. 2.4.2 and 4.2.2). These parts have no corresponding counterparts in the target model. Thus, if we deleted the source model, these parts of the source model are inevitably lost.

To address these issues, SCHÜRR suggested to use incrementally working translation processes in 1994 [Sch95]. Several algorithms for this *incremental update* problem have been described since (e.g., [GW09, GH09, HLR06]). In general, these approaches work in a similar way: First, we revoke rule applications that do not hold any more by deleting the produced part of the target models; second, we try applying new rules like in a batch transformation.

In the following, we describe the basic principles of the approach of WAGNER [Wag09, GW09, GW06].[14]

Fig. 2.29 shows the activities of the incremental update algorithm. The idea is that, before applying new rules, we first check all existing rule applications (created during the previous transformation run). First, we check whether the rule structure still holds: Do all nodes of the applied TGG rule still have correct object counterparts in the model, and do all edges have matching links? If not (i.e., the changes invalidated the rule application) we have to revoke the rule application. If the structure is still ok, we check whether all attribute conditions still hold. Changed attributes can be propagated without revoking the rule application by simply re-evaluating the attribute conditions. After this check, we apply new rules, as during a normal batch transformation. For rule applications that have been revoked we also try to find new rules that match.

WAGNER [GW09] argues that this incremental update algorithm produces the same results as the batch transformation described by SCHÜRR and, thus, is correct. HERMANN ET AL. [HEO$^+$13] provide a formal synchronization frame-

[14]Note that WAGNER and other authors also speak of "model synchronization" when only the changes of one model are propagated to another. In this thesis, we distinguish between "incremental updates" (when updating just one model) and "synchronization" (when simultaneously synchronizing changes in both models).

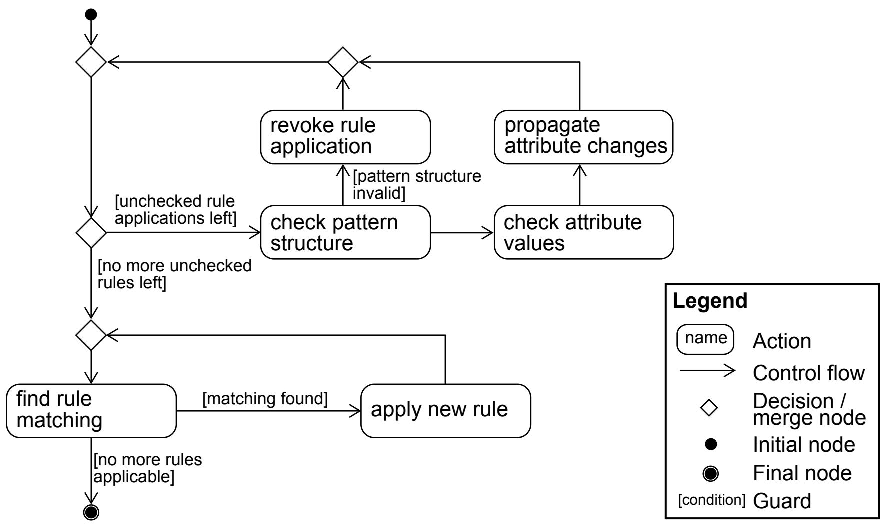


Figure 2.29: Incremental update algorithm (adapted from [Wag09])

work for this incremental update approach and present a proof for correctness, completeness, termination, and functional behavior.

We use Wagner's approach [GW09] as a basis in this thesis. We extend and improve the approach to fulfill the requirements for model transformation and synchronization in the development of mechatronic system.

2.4.3.2 Simultaneous Bidirectional Model Synchronization

If both models have changed since the last transformation or incremental update, we cannot simply propagate changes from the source model to the target model, as the changes in the target model may be overwritten then. In general, model synchronization is prone to conflicts. Two approaches have been suggested to address this issue.

Körtgen [Kör09] uses pre-generated repair actions that will put the models in a consistent state at that specific position. Several repair actions may be possible in a certain situation. Körtgen argues that the user should manually select one of these actions to propagate the change.

Xiong et al. [XLH+07, XSHT09] propose to propagate simultaneous changes by first incrementally updating the last synchronized state of the target model. Second, they perform a 3-way merge to combine the updated target model with the user-edited target mode. Finally, their algorithm incrementally updates the source model using this merged target model. Conflicts are processed by the model merger.

For a more in-depth discussion on these model synchronization approaches, we refer to Sect. 4.3.2.

2.4.4 Model Transformation Features

In the following, we classify TGGs according to the (adapted) classification by Czarnecki and Helsen [CH06] presented in Sect. 2.2.[15]

- *Specification:* A model-to-model transformation is specified using a formal graph grammar that defines a language of model triples. To compute the result of the transformation, the transformation engine has to derive such a valid model triple given only the source domain of that triple.
- *Transformation Rules:* Each rule defines how two models may evolve simultaneously. Thereby, a rule provides a declarative specification of a mapping between a source and a target modeling construct.
  - *Domain:* Typically, two domains are used as source and target. However, the number of domains is not limited ("multi-graph grammars" [KS06, KS05]), but there must be at least two (one source and one target domain).[16]
    - *Body:* The nodes and edges of the rule's pattern serve as *variables.* A rule consists of a *graph pattern* containing typed nodes and edges. Concerning the rules' *logic*, each rule specifies a mapping between a source and a target modeling construct.
  - *Multidirectionality:* In general, rules are multidirectional by construction[17].
  - *Intermediate Structures:* A correspondence model connect the domains. It can be used as a trace model and to store intermediate information.
- *Rule Application Control:* The rule application control highly depends on the strategy of the transformation algorithm that is used for the transformation. However, if the transformation does not contain rule conflicts, the only difference between different approaches is performance. If there are conflicts, different solution strategies are used. We refer to Hildebrandt et al. [HLG+13] for further details on the differences between existing TGG approaches. Here, we describe the algorithm of the TGG Interpreter, which is used in this thesis.
  - *Location Determination:* The Interpreter uses the concept of a *front* [KW07]. This front maintains a (first-in-first-out) list of source model objects that have unprocessed links left. New objects are added to the list whenever a rule is applied.
  - *Rule Scheduling:* Prior to the implementation of the additional concepts of this thesis, the TGG Interpreter selected the next rule to apply by the order in which they are contained in the ruleset. As optimization, the TGG Interpreter applies the same rule as long

[15]We only consider TGGs as a model-to-model transformation approach here, i.e., we do not regard the "simultaneous evolution" semantics of TGGs.

[16]Sometimes the correspondence links are also regarded as a domain, as they can be stored in the form of a model. In contrast to the source and target domains, the correspondence domain's (meta-)model is under control of the transformation developer.

[17]A rule has to be well-formed, such that it contains produced nodes for all domains and possible attribute constraints are also multidirectional.

as it is applicable, then switching to the next rule in the list. It continuously cycles through the list until no further rule can be applied. See Chap. 4 for details on rule selection within the novel approaches presented in this thesis.

- *Rule Organization:* Rules are contained in a ruleset, but do not have explicit relationships between them.
- *Source-Target Relationship:* TGGs can be used for batch and incremental model-to-model transformations.
  - *New Target:* Creating a new target model from a give source model is usually described with the terms "batch transformation" or "initial transformation" (the latter mainly to denote that later incremental updates will follow).
  - *Existing Target:* TGGs support incremental updates (see, e.g., GIESE AND WAGNER [GW09]), but no in-place model transformations.
- *Incrementality:* SCHÜRR already suggested incremental updates in his original publication [Sch95]. Since then, several approaches for incremental updates have been presented.
  - *Incremental Change Processing:* Given the change set that happened to a model since the last transformation run, the transformation engine can compute possibly affected rule application and, therefore, only has to check these rule applications. This reduces the runtime especially for large models [GH09].
  - *Propagation:* Different incremental update / change propagation approaches have been described in the literature, which differ in their amount of user-edit preservation (see Sect. 4.1.1 for a detailed discussion).
- *Directionality:* If all rules are multidirectional, a TGG ruleset is also multidirectional without further requirements.
- *Tracing:* The correspondence model that links the source and target pattern of a rule is typically persisted with the generated target model and can be used as trace information (and for further incremental updates).

CHAPTER 3

Synchronizing Mechatronic System Development Models

Contents

In this chapter, we describe the running example used in this thesis in detail. It contains exemplary, but characteristic situations and scenarios for model transformations and synchronizations. In Chap. 4, we will refer to these situations to illustrate the different model synchronization extensions developed in this thesis. Furthermore, we will explain the different transformations that are used to map between the system model and the discipline-specific models.

After giving an overview about the example scenario process in Sect. 3.1, we describe how to derive initial discipline-specific models in Sect. 3.2, and how to later synchronize models in Sect. 3.3.

3.1 Example Scenario Overview

As a running example, we use the RailCab system in this thesis. The RailCab – "Neue Bahntechnik Paderborn"[1] – is based on the vision that, in the future, the schedule-based railway traffic will be complemented or replaced by small, autonomous RailCabs that transport passengers and goods on demand, being more energy-efficient by dynamically forming convoys. A prototype has been developed at the University of Paderborn and served as an extensive case study for the Collaborative Research Center (CRC) 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering". A test track at the scale of 1:2.5 was built at the campus of the University of Paderborn.

When a RailCab drives separately, it calculates the desired speed based on a reference value. The reference value depends on several factors like the necessity for a quick arrival, desired user comfort, energy availability etc. It is calculated using self-optimization methods (see Gausemeier et al. [GRS14a] for details). The velocity controller compares this reference value to the current speed and calculates the operating point for the traction system.

In a convoy, the RailCabs drive closely together to optimally make use of the slipstream, but there is no mechanical connection between the RailCabs. The participating RailCabs therefore have to drive at exactly the same speed. To achieve this, RailCabs negotiate the convoy driving speed using WiFi communication before entering a convoy, as depicted in Fig. 3.1. Within a convoy, all RailCabs (except for the convoy leader) control their velocity based on the distance to the RailCab traveling in front of them. To measure the distance, there is a distance sensor mounted in the front of each RailCab. When entering a convoy, the RailCab *reconfigures* its velocity controller such that it now uses the data from the distance sensor. The controller tries to maintain a stable distance between the RailCabs by adjusting the acceleration/deceleration of the traction unit.

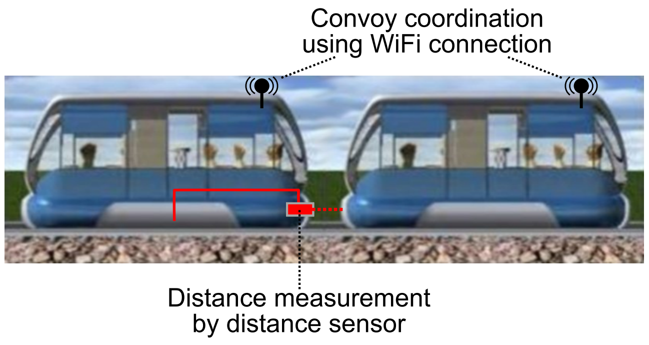


Figure 3.1: Convoy coordination concept of the RailCab system

Obviously, this distance sensor is a highly safety-critical element, as a failure (i.e., it outputs wrong distance data or no data at all) can result in two RailCabs colliding, putting the lives of humans at risk. Thus, a redesign of the distance

[1]Neue Bahntechnik Paderborn/RailCab: `http://www-nbp.uni-paderborn.de/`

measurement concept may be necessary even later in the development, as it may turn out that the original concept is not safe enough. Such a redesign potentially affects several disciplines. As the engineers of all these disciplines have already started the development, it is crucial to influence existing disciplines' development artifacts as less as possible.

In the conceptual design phase, experts from all disciplines design the principle solution. This principle solution covers all discipline-spanning relevant information, i.e., all interfaces and overlaps between different disciplines are described in this model. Thus, the principle solution can serve as a starting point for the discipline-specific design and development phase.

Model transformation techniques can be applied to *automatically derive initial discipline-specific models* that are consistent with the principle solution and all other discipline-specific models. Basically, these initial models contain skeletons that are filled by the discipline engineers in the following design and development phase. We explain how a model transformation generates software engineering models from a principle solution in Sect. 3.2.2. In Sect. 3.2.3, we show how initial control engineering models can be generated.

Ideally, the principle solution covers all discipline-spanning aspects. Thus, there should be no need for further discipline-spanning coordination. However, in practice, the principle solution rarely captures all discipline-spanning concerns. Additionally, changes to the overall system design may become necessary later, e.g., due to changing requirements, or, as explained before, to improve the safety of the system. Therefore, cross-discipline changes may become necessary during the discipline-specific design and development phase.

To illustrate transformations and synchronizations that may become necessary throughout the development, assume the following exemplary process. Fig. 3.2 shows how the different models evolve in this exemplary process.[2] The numbers in the red circles denote the different process steps. They are used in all following figures to refer to the respective process step.

After the system engineers designed the principle solution, model transformations can be applied to generate the different discipline-specific models (step 1). The engineers from the different disciplines start implementing the controllers (step 2). For instance, the control engineers elaborate on the velocity control strategies for driving in a convoy as a follower based upon the distance measured by the distance sensor. Software engineers implement the communication and reconfiguration behavior when a RailCab enters or leaves a convoy. These are discipline-specific refinements that have no direct influence on the models of other disciplines.

In the meantime, the system engineers notice that the current structure of system elements does not resemble the functionality of the system well. Thus, they restructure the system model such that the Distance Sensor and the Distance Calculation elements are now sub-system elements of a new Distance Measurement system element.

[2]This example is a modified and extended version of an exemplary process published in GAUSEMEIER ET AL. [ADF+14].

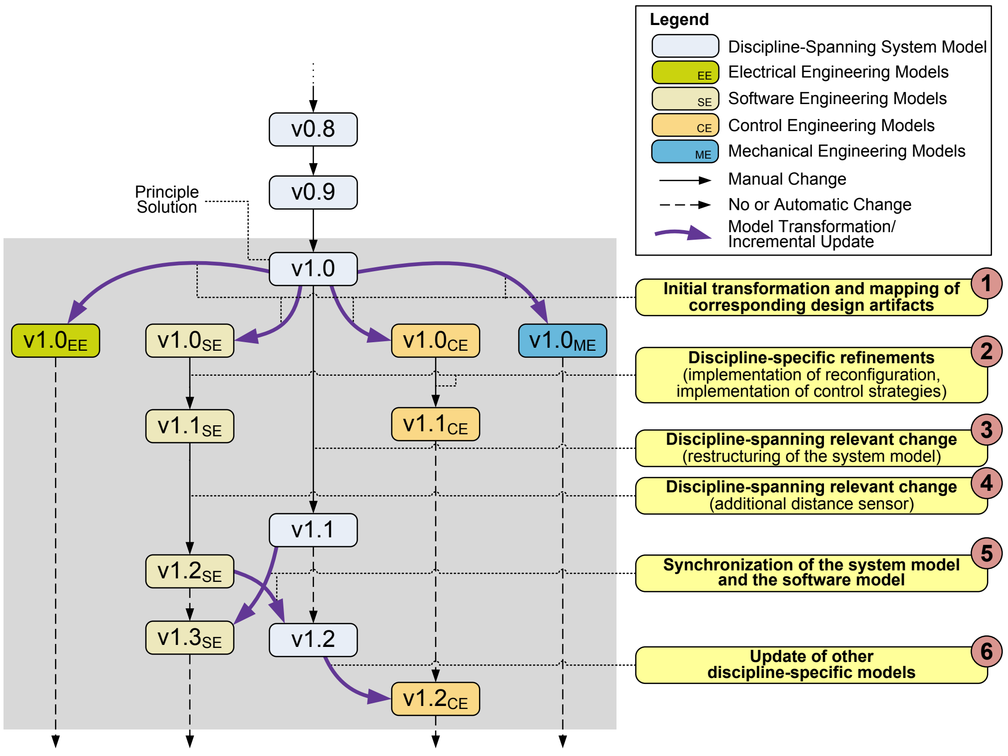

Legend
Discipline-Spanning System Model
EE Electrical Engineering Models
SE Software Engineering Models
CE Control Engineering Models
ME Mechanical Engineering Models
Manual Change
No or Automatic Change
Model Transformation/ Incremental Update
v0.8
v0.9
Principle Solution
v1.0
v1.0EE
v1.0SE
v1.0CE
v1.0ME
v1.1SE
v1.1CE
v1.1
v1.2SE
v1.3SE
v1.2
v1.2CE
1 Initial transformation and mapping of corresponding design artifacts
2 Discipline-specific refinements (implementation of reconfiguration, implementation of control strategies)
3 Discipline-spanning relevant change (restructuring of the system model)
4 Discipline-spanning relevant change (additional distance sensor)
5 Synchronization of the system model and the software model
6 Update of other discipline-specific models

Figure 3.2: Evolution of different models during the development process

Modern mechatronic systems incorporate self-healing to repair the system in case of failure. It is the duty of the software engineers to define and analyze this self-healing. Amongst others, the software engineers perform an analysis of the self-healing operations in order to judge whether they reduce the probability of hazards successfully. In our example, the distance sensor may fail or send bad data. It may turn out that even with self-healing the hazard probability cannot be reduced to an acceptable level: The hazard of two RailCabs colliding during convoy mode due to a failing distance sensor exceeds the acceptable hazard probability of the system.

Thus, software engineers propose to add redundancy by adding a second distance sensor. In this way, the system can detect when a sensor sends bad distance data, as this data then differs with the data of the other sensor. If this happens, the system can initiate fail-safe behavior like emergency convoy break-up.[3] They add a new sensor measurement component to their software model (step 4) to implement this. This is a discipline-spanning relevant change, i.e., it affects the discipline-spanning system model as well as several discipline-specific models. In particular, the velocity control strategy must be modified.

Thus, the change is propagated to the system model, and the system model changes are incorporated into the software model (step 5) using model syn-

[3]Note that with two sensors one can just detect a sensor failure, but not resolve it. To eliminate the bad data (i.e., still know which data is correct), three sensors are required – also called “triple modular redundancy” pattern.

chronization techniques. In contrast to model transformation that translates complete models, the idea is to modify only the model elements that have been changed after the initial model transformation. Thus, this is also called *incremental update*. Version 1.2 of the system model now contains a second distance sensor, and version 1.2 of the software model was restructured according to the new system model structure.

To allow all engineers reacting to the discipline-spanning relevant changes from the software model, they are propagated further from the system model to all affected discipline-specific models. For instance, the control engineering model is incrementally updated (step 6). The control engineers can now modify their control strategy to use both sensor data as input.

In steps 5 and 6, it is crucial that the discipline-specific models are updated in a way so that all refinements and implementations that have been added to it in the meantime are retained. In our case, the changes to the control-engineering model (see step 2) and to the software-engineering model (see step 4) must not be removed, overridden or otherwise affected.

Next, we describe in detail a) how to derive initial discipline-specific models, and b) how to propagate changes to the system model and further on to discipline-specific models.

3.2 Deriving Initial Discipline-Specific Models from the System Model

In this section, we present two example transformations from the system model to discipline-specific models. Here, the principle solution serves as the input to create models for the discipline-specific design and development phase. However, these transformations can also be used during the conceptual design to generate discipline-specific models, for instance, for early simulation and verification.

Before transforming to the discipline-specific models, engineers define which parts of the system model are relevant to which discipline; we describe this in Sect. 3.2.1. Next, we show how the active structure can be used to derive initial software component models in Sect. 3.2.2, and how the behavior–states model is transformed to an initial software statechart. Last, we describe how control engineering models can be derived from the system model in Sect. 3.2.3.

3.2.1 Defining Discipline Relevance

Not all models and aspects of the system model are relevant to each discipline. Even only parts of a single partial model may be relevant to a certain discipline. For instance, system elements in the active structure solely representing mechanical parts are irrelevant to software engineering, but sensors or actuators are. To define this *relevance*, we use the concept of *relevance annotations*, which we introduced in previous work (cf. RIEKE [Rie08]).

You can regard these annotations as stereotypes (as used in UML [OMG10b]) that are assigned to elements of the system model. Each discipline has its own kind of relevance annotations. Whenever there is a relevance annotation attached to an element of the system model, this annotation marks this element as relevant to the respective discipline. Mechatronic Systems usually consist of mechanic parts, electrical parts, controllers, and software. Therefore, we distinguish between *Mechanical Engineering* (denoted by "ME" in the relevance annotations), *Electrical Engineering* ("EE"), *Control Engineering* ("CE"), and *Software Engineering* ("SE").

For instance, consider the active structure of the RailCab (as shown in Fig. 2.5). Figure 3.3 shows this active structure (modeled in the CONSENS language) with these relevance annotations. The Drive Control is only relevant to software engineering, as it contains only state-based behavior and discrete communication. Although it communicates with other RailCabs and the Track Section Control using the Communication Module, this module is transparent to software engineering, as it just implements the hardware levels of the communication. Thus, it is only relevant to mechanical engineering (as it must be physically integrated into the chassis) and electrical engineering (as it must be powered).

Although we describe a method to automatically derive such relevances in previous work [Rie08], practice shows that this is still a largely manual process that has to be performed by systems engineers and discipline experts [GSG⁺09].

Next, we describe how to transform the relevant system model parts to the different discipline-specific models.

3.2.2 Transformation from CONSENS to Software Engineering Models

Here we describe our concept to map between the system model and software models defined in MechatronicUML (cf. Sect. 2.1.4.1). Figure 3.4 shows the basic principles of the transformation from CONSENS to MechatronicUML software models. In the active structure, you can see small colored annotations above the system elements. These so-called *relevance annotations* define which element is relevant to which discipline-specific model. For instance, "SE" and "CE" denotes software engineering and control engineering, respectively.

The central idea of the mapping is that every system element that has a software engineering relevance annotation (i.e., it fulfills software functions) should be represented by a software component in the MechatronicUML model. The information flows between system elements are mapped to ports and connectors in MechatronicUML.

In general, MechatronicUML only defines the discrete software. However, MechatronicUML allows to integrate continuous components, e.g., controllers from control engineering. Continuous components are black-box components in MechatronicUML, i.e., no actual behavior is attached to continuous components in MechatronicUML. In this way, they can be used to define the interface to control engineering in a MechatronicUML software model.

In contrast, the behavior of *discrete components* is implemented using MechatronicUML. Discrete components communicate with each other via discrete ports using asynchronous, message-based communication. This communication is implemented in using real-time statecharts (see Sect. 2.1.4.1).

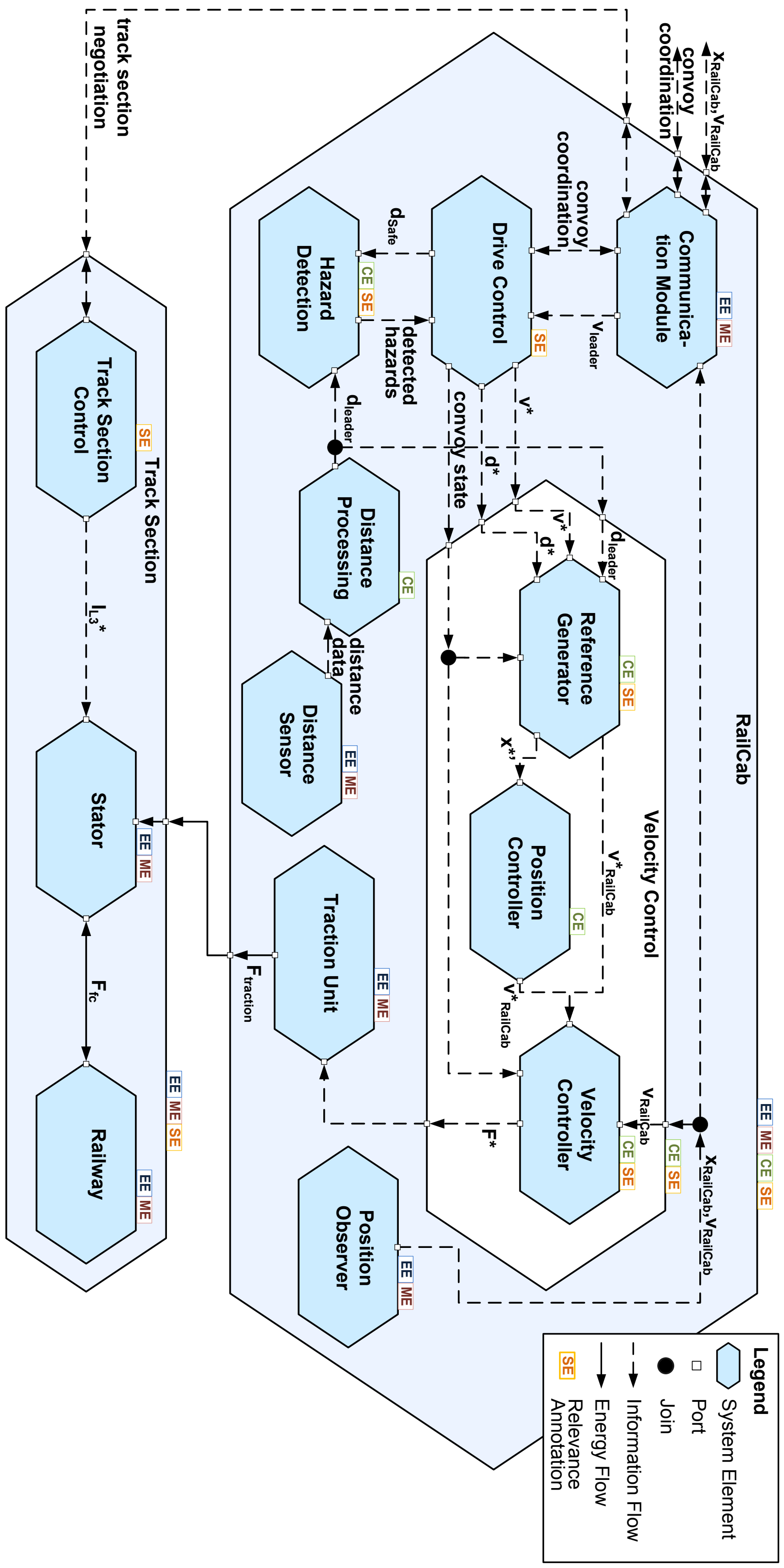


Figure 3.3: Excerpt of the active structure of the RailCab (from Fig. 2.5) with relevance annotations

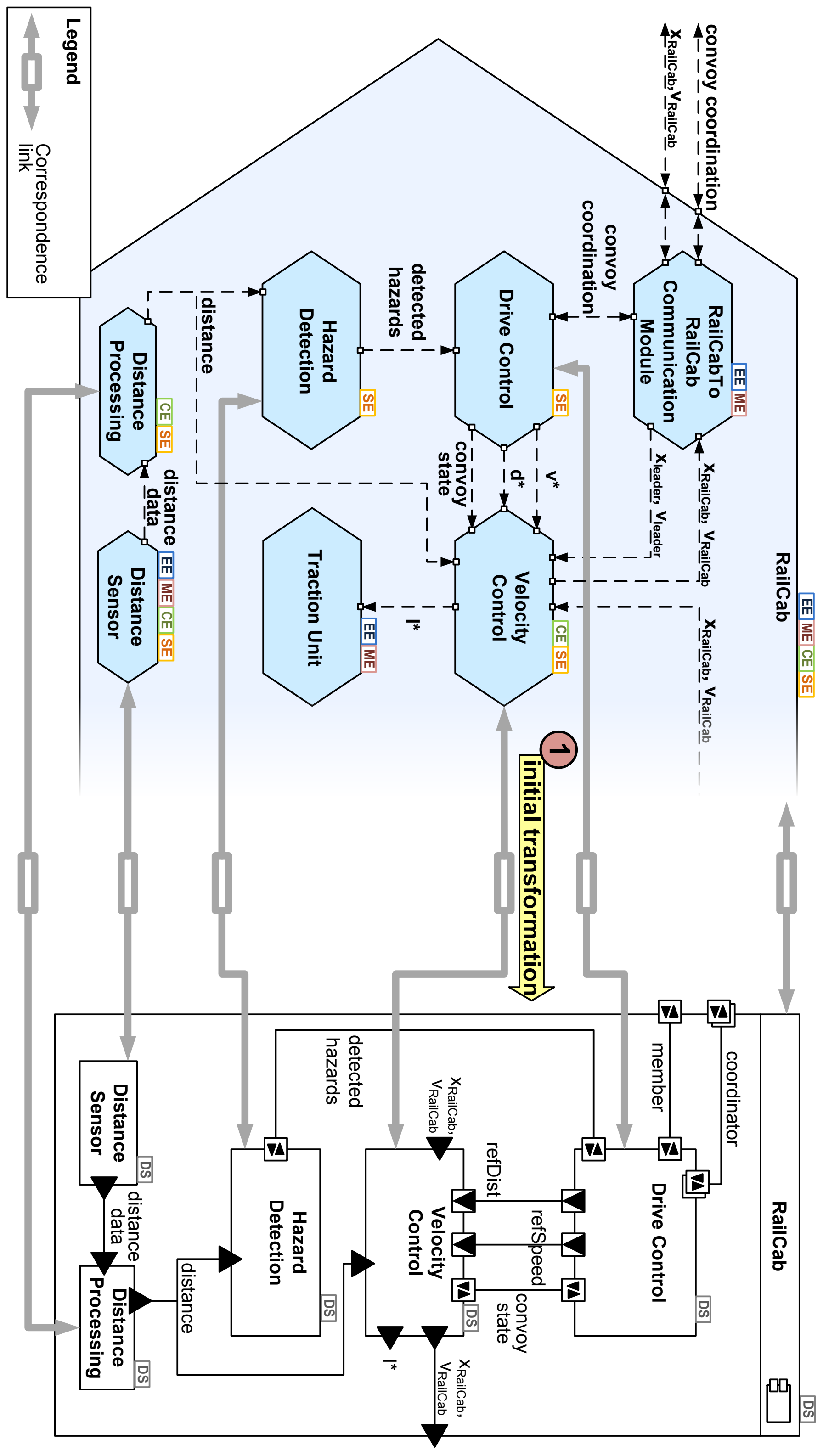


Figure 3.4: Initial transformation from the active structure to a software component diagram

Discrete components can also send or receive signals to or from continuous components using hybrid ports. In Fig. 3.4, the components Drive Control and Velocity Control have both discrete and continuous ports.

Figure 3.5 shows the main concepts of the mapping between the system model (modeled in the CONSENS language) and the software model (modeled in the MechatronicUML language). For instance, the transformation creates components for every system element that should be implemented in MechatronicUML. Most mappings are rather straightforward. As described in Sect. 2.1.4.1, we distinguish between discrete, state-based behavior implemented in MechatronicUML and continuous behavior implemented using MATLAB/Simulink. The mapping also resembles this distinction: System elements that are relevant to software engineering become discrete components, and elements relevant to control engineering become continuous components. System elements relevant to both disciplines become hybrid components, which integrate both continuous and discrete behavior. Note that the concrete syntax of MechatronicUML does not distinguish between component types; therefore, we added three notes to Fig. 3.5 to show the differences.

We use *Triple Graph Grammars* (TGGs, cf. Sect. 2.4) for precisely defining the model transformations to the different discipline-specific models. The complete TGG ruleset for the transformation can be found in Appendix A.

3.2.3 Transformation from CONSENS to Control Engineering Models

Next, we present the transformation from the system model to control engineering models in MATLAB/Simulink and Stateflow (cf. Sect. 2.1.4.2). Figure 3.6 shows the basic principles of the transformation from CONSENS to MATLAB/Simulink control engineering models. Generally, every system element that is relevant to software engineering or control engineering is mapped to a Simulink block. System elements relevant only to software engineering, however, are just placeholders. They are later filled with the implementation from software engineering. This is because a MATLAB/Simulink model is used at the end of the development process as a combined software/control engineering model from which code is generated.

MATLAB/Simulink neither has built-in means to define asynchronous, message-based communication, nor does it allow dynamically instantiating/destroying blocks or rewiring connectors. Nevertheless, such functions are necessary for the simulation of reconfigurable systems like self-optimizing systems. In such systems we find changing communication structures, rerouting of information flows (signals), and dynamic instantiation/destruction of software components/controllers.

MATLAB supports defining state-based behavior using its toolbox Stateflow. Stateflow only provides basic means for defining timed behavior.[4] For

[4]There are only simple temporal logic operators like `after(`*amount*`,` *unit*`)`, `before()`, and `at()`. More importantly, these operators are state-local, i.e., they can only measure time relative to the activation of the current state. Furthermore, when a subsystem is disabled, time does not progress for these operators. [TMW14]

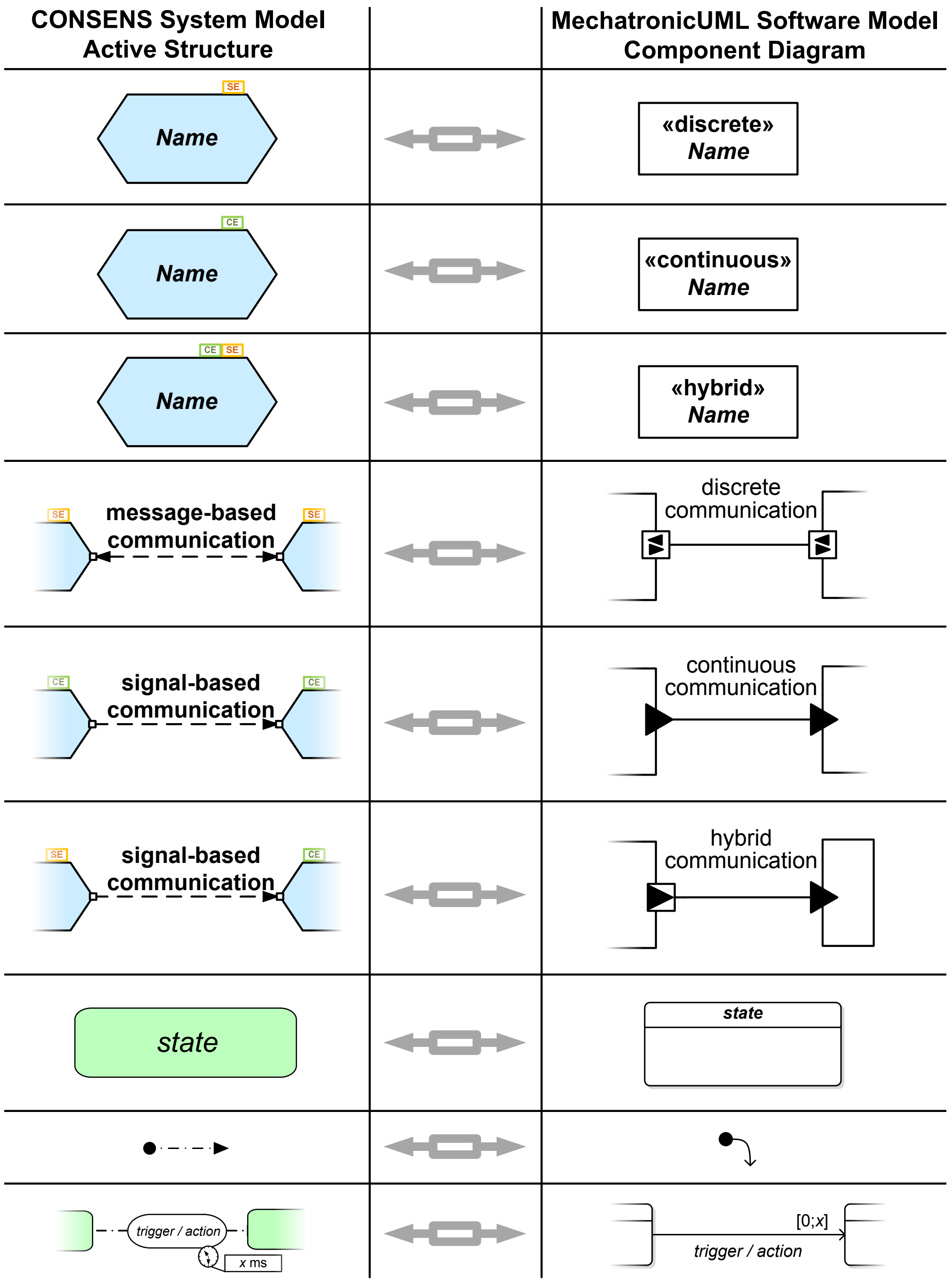


Figure 3.5: Mapping from CONSENS to MechatronicUML

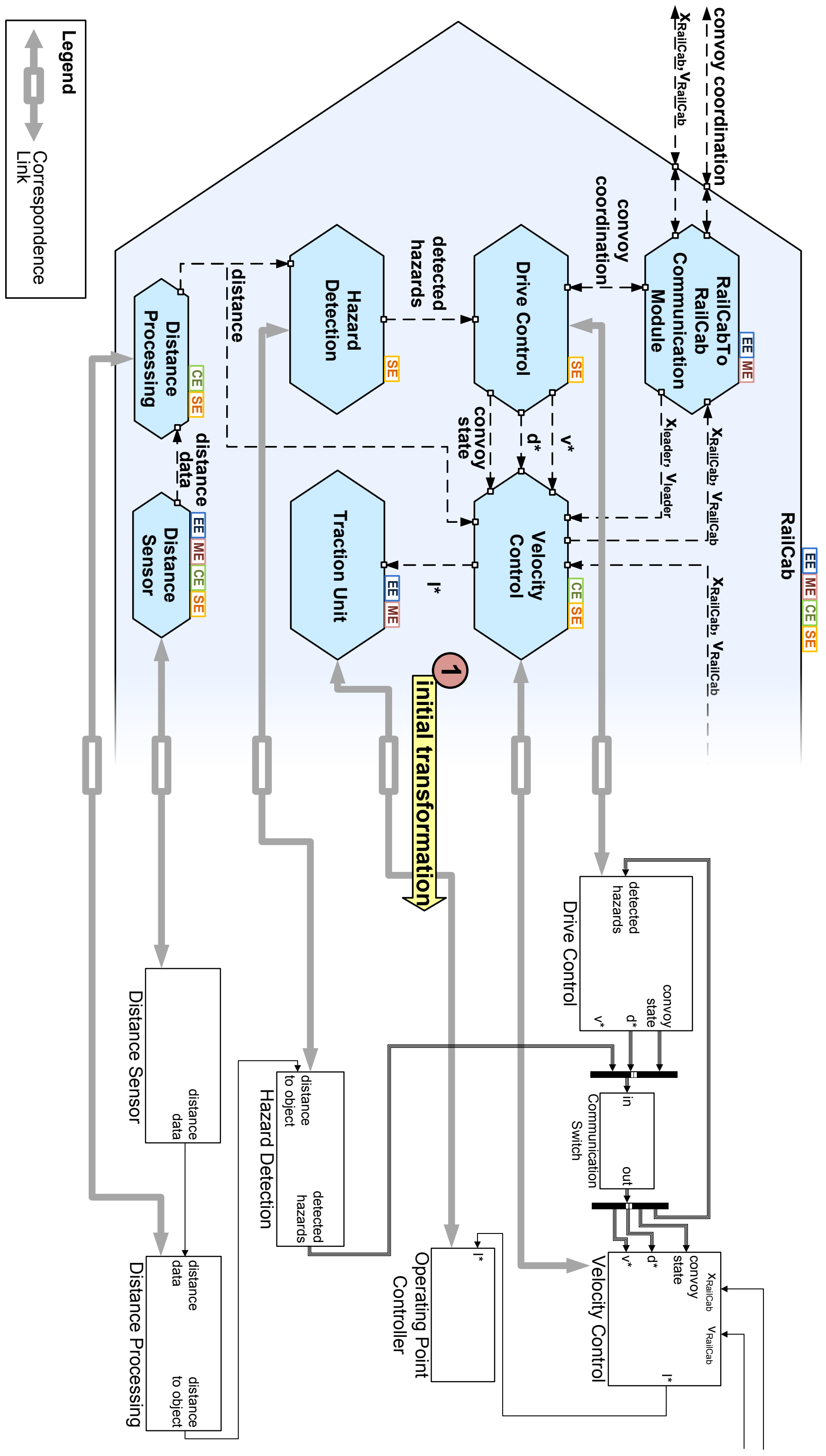


Figure 3.6: Initial transformation from the active structure to a MATLAB/Simulink control engineering model

safety-critical mechatronic systems, time constraints often require more sophisticated expressions that are also able to deal with absolute time[5].

It is, however, possible to implement these concepts using helper structures and custom blocks. This is a time-consuming and error-prone task if performed manually. Thus, we developed a concept for automatically generating these helper structures. This is most important when integrating software and control engineering models by generating this final combined software/control engineering model at the end of the of the system development. However, to allow early integration and testing, it is important that the control engineering model resembles this structures from the beginning of the design and implementation phase.

To allow the integration of discrete software components that use asynchronous, message-based communication and reconfiguration, we use a message bus. The communication between two discrete components is implemented using a Communication Switch. This switch connects to every component and is responsible for forwarding sent messages to the correct recipient.

Similar to most low-level communication protocols like Ethernet, a message consists of a header and a body, which contains the actual data. The header is a five-tuple `(message_id, sender_id, receiver_id, message_type, timestamp)`. Messages have sequential integer numbers (`package_id`) that can be used to track lost messages. The `sender_id` is the network address of the sender; the `receiver_id` is the address of the intended receiver. The type of message is encoded in the `message_type` field. `timestamp` contains the time at which the message was sent by the sender. Simulink does not support variable-sized data buses. Thus, messages with more than one parameter (or parameters of variable length like strings) need to be split into several messages. A message is sent via the Communication Switch, which forwards the message to their intended receiver [HPR+12, HRB+14].

Signal-based information flow, like the I^* value signal from Velocity Control to Operating Point Controller, is mapped to connected outputs and inputs of the respective blocks.

As MATLAB/Simulink does not support instantiating/destroying blocks, we have to emulate this behavior when mapping *reconfiguration* to MATLAB/Simulink. First, we compute the maximum number of possible instances of a component type using the reconfiguration rules.[6] We generate this number of *enabled subsystems* in Simulink. An enabled subsystem is only active if its control input is greater than 1. By setting this control input to 0, we can switch off the computation for this block. In contrast to a true destruction of a component, this does not free the memory required by this block; however, it does not consume any computation time for disabled blocks, because their transfer functions are not evaluated.

When reconfiguring communications, we simply change the target net address in sent messages. The communication bus then automatically routes the

[5]The Stateflow documentation [TMW14] mentions "absolute-time temporal logic", but this denotes that the time measured since entering the current state is *not influenced by the sample time* of the model.

[6]We set an upper instance limit in case the maximum number of instances is infinite.

messages to the correct component.

Furthermore, behavioral models of the principle solution can be used to generate STATEFLOW models. As described above, STATEFLOW does not provide the necessary temporal logic operators. Therefore, we also generate helper functions that can be used in STATEFLOW to, for example, reason about absolute time. Furthermore, we include direct access to the message-based communication described above by generating helper functions for sending and receiving messages. We describe the principles of this transformation of state-based models in [RDS+12].

Figure 3.7 shows such a transformation. CONSENS states are mapped to Simulink states, and transitions to transitions. Events, actions and time constraints (clocks etc.), however, cannot be simply translated, as MECHATRONICUML is far more expressive than MATLAB/STATEFLOW. Again, we solve this issue by automatically generating helper constructs that deal with events, actions and time constraints.

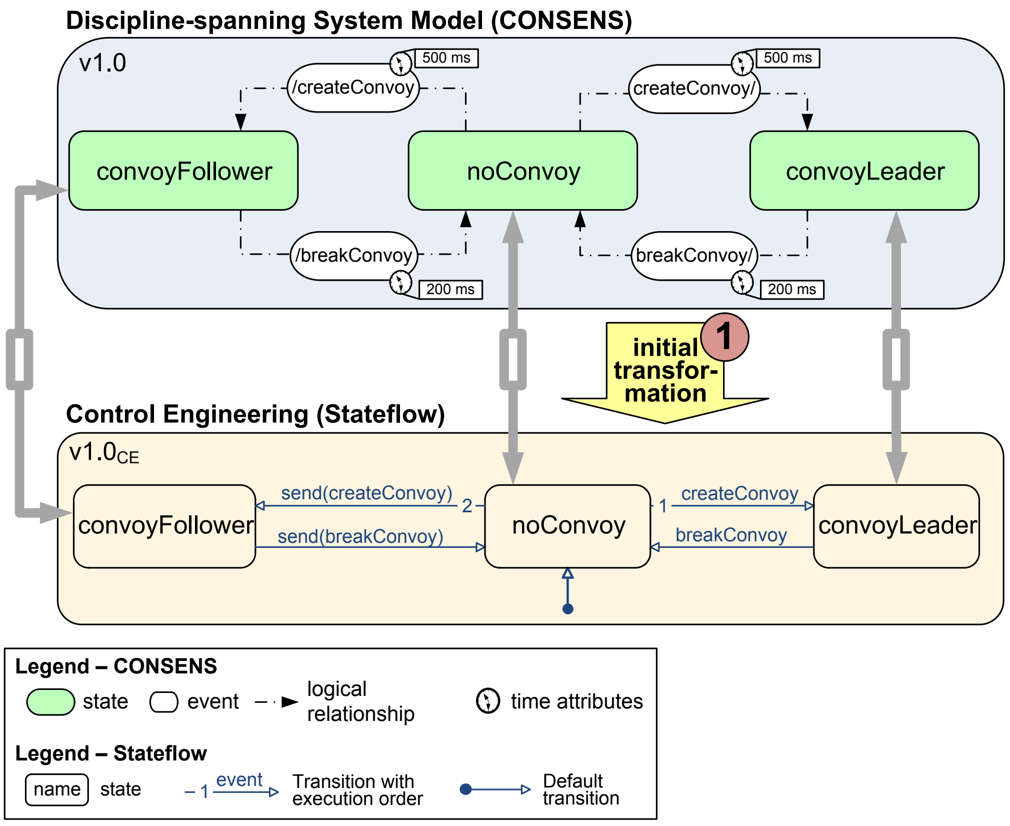
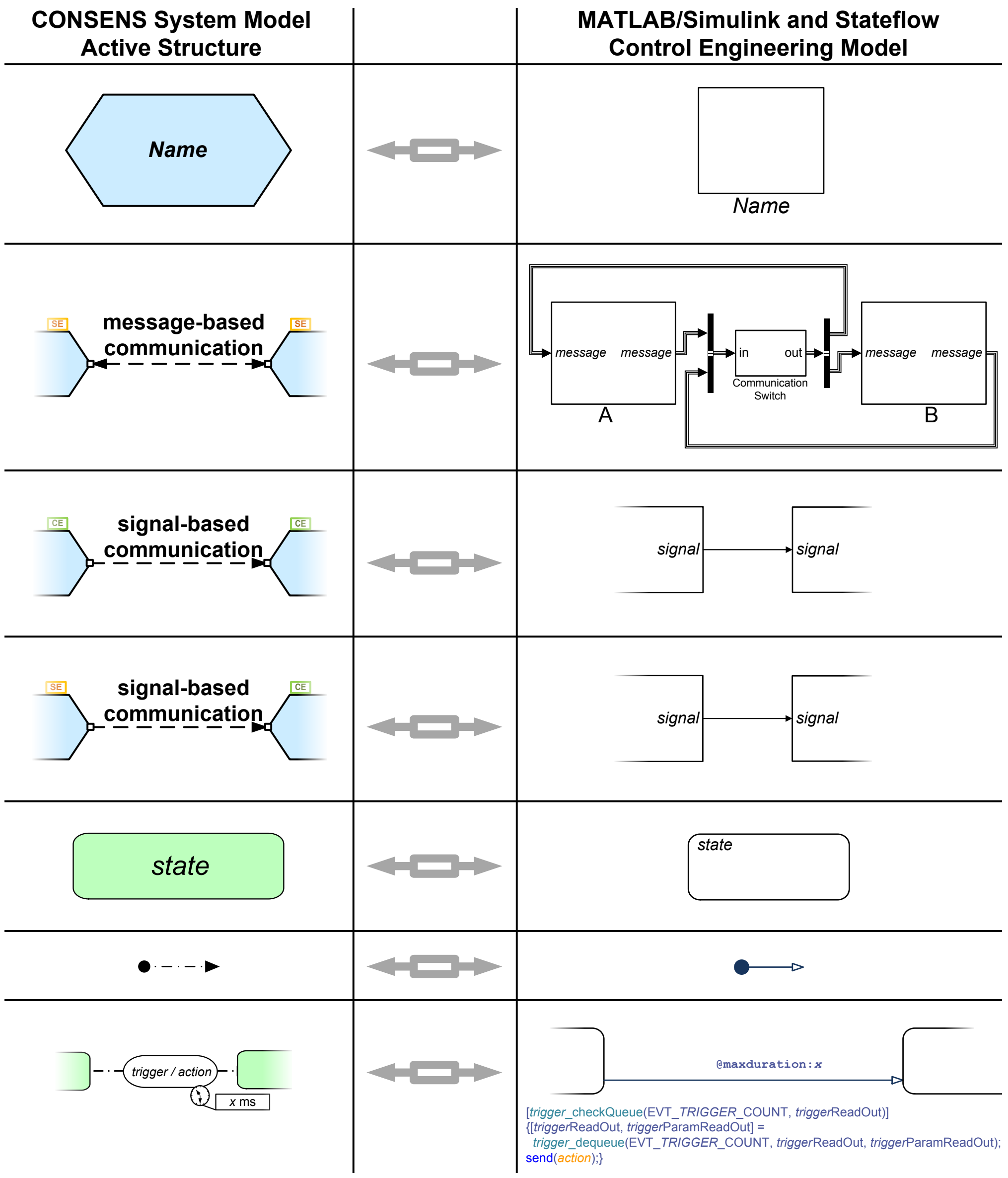


Figure 3.7: Initial transformation from Behavior–States to a MATLAB/STATEFLOW model

Figure 3.8 shows the main concepts of the mapping between the system model (modeled in the CONSENS language) and the control engineering model (modeled in MATLAB/SIMULINK and STATEFLOW). It also shows how triggers and actions are represented in MATLAB/STATEFLOW (the bottom-most map-



```
[trigger_checkQueue(EVT_TRIGGER_COUNT, triggerReadOut)]
{[triggerReadOut, triggerParamReadOut] =
  trigger_dequeue(EVT_TRIGGER_COUNT, triggerReadOut, triggerParamReadOut);
send(action);}
```

Figure 3.8: Mapping from CONSENS to MATLAB/Simulink and Stateflow

ping). Even for this simple trigger/action combination, the generated MATLAB code for the transition is barely human-comprehensible. Thus, we use an abbreviated form for events and messages in the STATEFLOW model in Fig. 3.7 and throughout this thesis.

Furthermore, STATEFLOW does not include concepts for specifying the maximum duration of a transition, as MECHATRONICUML does. However, this information is still important in later phases, so we add it to the STATEFLOW model as an annotation.

For further details on the generation of MATLAB/SIMULINK and STATEFLOW models we refer to HEINZEMANN ET AL. [HPR+12].

3.2.4 Transformation to Other Disciplines

Concerning electrical engineering, initial circuit diagrams that model the power distribution in a system can be generated from the system specification, especially from the active structure and its energy flows and energy-related system elements. Here we also find interconnections with models from other disciplines. For instance, the size of a wire or a cable conduit and its arrangement influences the mechanical design. Mass distribution and weight of a system, as defined by mechanical engineering, influence other disciplines in turn. However, it is difficult to automatically generate initial model for mechanical engineering, as the design of a product is a highly creative process.

Consequently, ensuring the consistency also for models of mechanical engineering and electrical engineering is crucial. Thus, we also need mappings between these types of models – and as these models are heterogeneous, such a mapping is complex. Applying sophisticated model-transformation techniques is therefore reasonable, too.

3.3 Synchronizing Models During the Discipline-Specific Refinement Phase

Although most discipline-spanning relevant information should already be present at the end of the conceptual design phase, changes to the system under development may become necessary during the discipline-specific design and development phase. For instance, requirements may still change during later phases, or it may turn out that some aspect of the system must be implemented in another way. This easily leads to changes that affect both the discipline-spanning system model and several discipline-specific models. Furthermore, early discipline-specific models may have already been generated during conceptual design to allow early checks and simulations of different concepts and ideas. It is reasonable to keep these early models, so that engineers can reuse and refine them during the design and development phase.

This requires keeping the development models consistent during all phases of the development. Manually checking and restoring the consistency of all models is a time-consuming and error-prone task. Therefore, we apply similar methods as with the derivation of initial models (described in the previous section) to synchronize models during the development.

First, we describe how the system model can be updated when changes in a discipline-specific model occur. Next, we show how discipline-specific models can be update with respect to these system model changes.

3.3.1 Updating the System Model

As described in Sect. 3.1, an extra distance sensor measurement component is added to the software model (step 3 in Fig. 3.2). This change must be propagated to the system model (step 4 in Fig. 3.2). Figure 3.9 shows how adding the extra sensor of the software model (marked with 2) to the system model (3) achieves this.

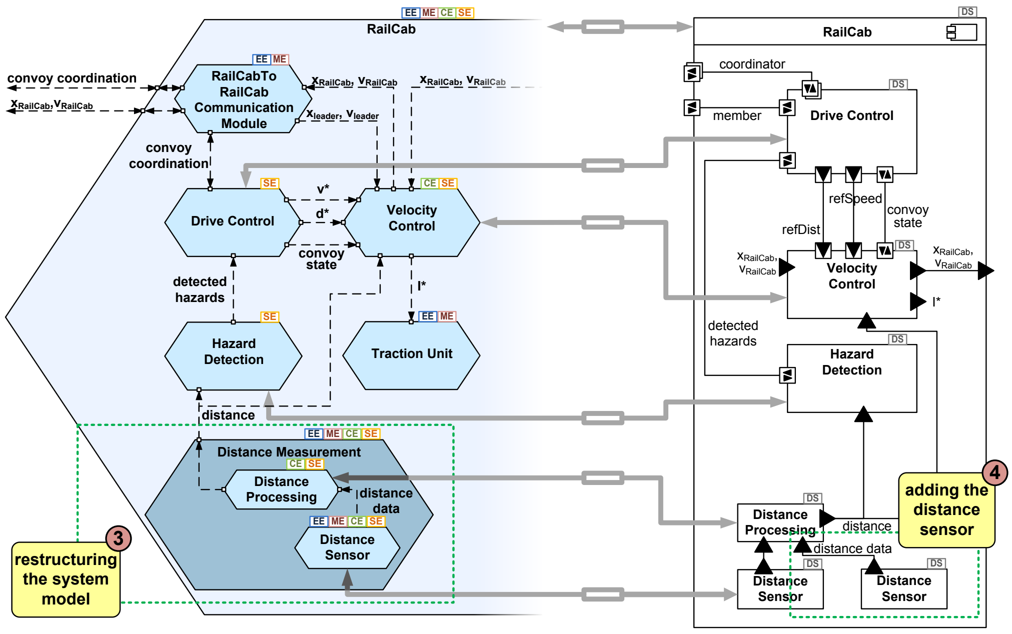


Figure 3.9: Simultaneous changes to both the system model and the software engineering model

We again use TGGs (cf. Sect. 2.4) to perform such model synchronization operations. TGG rules can be applied bidirectionally, i.e., transformation and synchronization operations can be performed both from the system model to the software model and vice versa. Here, we apply the TGG rules backwards, allowing the change to be propagated from the software model to the system model. Furthermore, we perform an *incremental model transformation* to update only the affected parts of the system model. The added Distance Measurement system element is shown on the left side of Fig. 3.10.

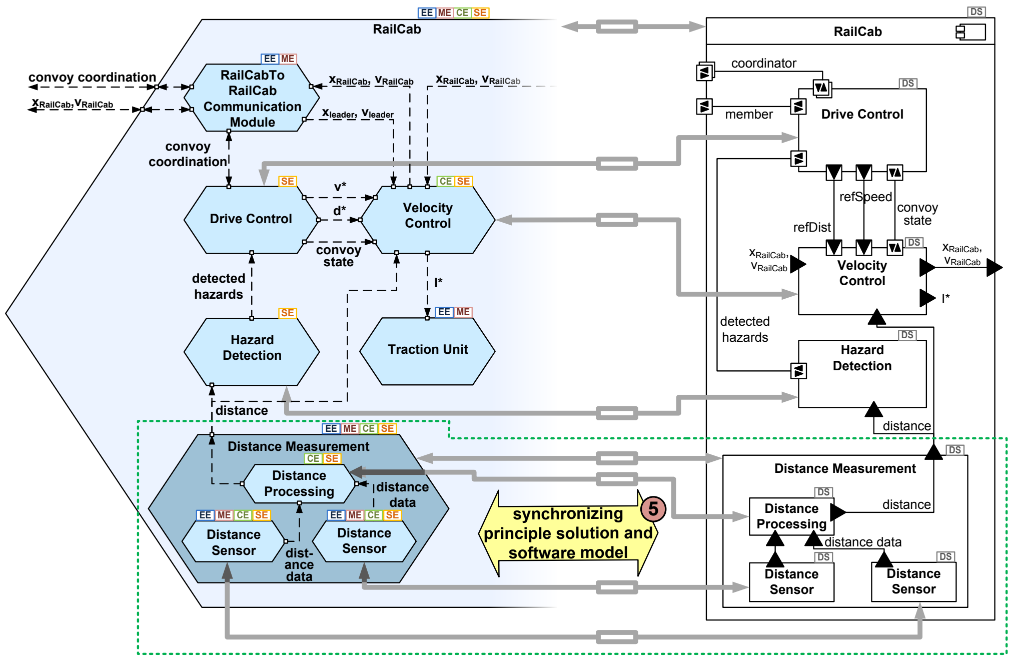


Figure 3.10: Updating the active structure using the altered software component diagram

3.3.2 Updating Control Engineering Models

After updating the system model, the changes must be propagated to other affected discipline-specific models (step 5 in Fig. 3.2). Figure 3.11 shows how the added Distance Measurement system element can also be added to the control engineering model. This is again achieved by rerunning the transformation incrementally, leaving the unaffected parts untouched, and only adding a new block with its respective inputs, outputs and lines.

3.3.3 Tackling the Challenges of Synchronizing Models for Mechatronic System Development

We have developed an improved model synchronization approach, which is one of the core contributions of this thesis. It prevents the loss of information in models during the synchronization process. For instance, the STATEFLOW model shown in the lower part of Fig. 3.7 is later refined such that it contains details of controller reconfigurations that happen when switching convoy states. Thus, the STATEFLOW model now contains information that is not relevant to (and, thus, not present in) the abstract system model. When the system model is changed later on, this change may affect parts of the STATEFLOW model that has been refined. This refinement must not be affected or lost when updating this discipline-specific model.

The approach allows to minimize the necessary user interaction by using expert knowledge encoded by metrics. When changes to a model occur, we are able to update other affected models automatically in most cases, using this

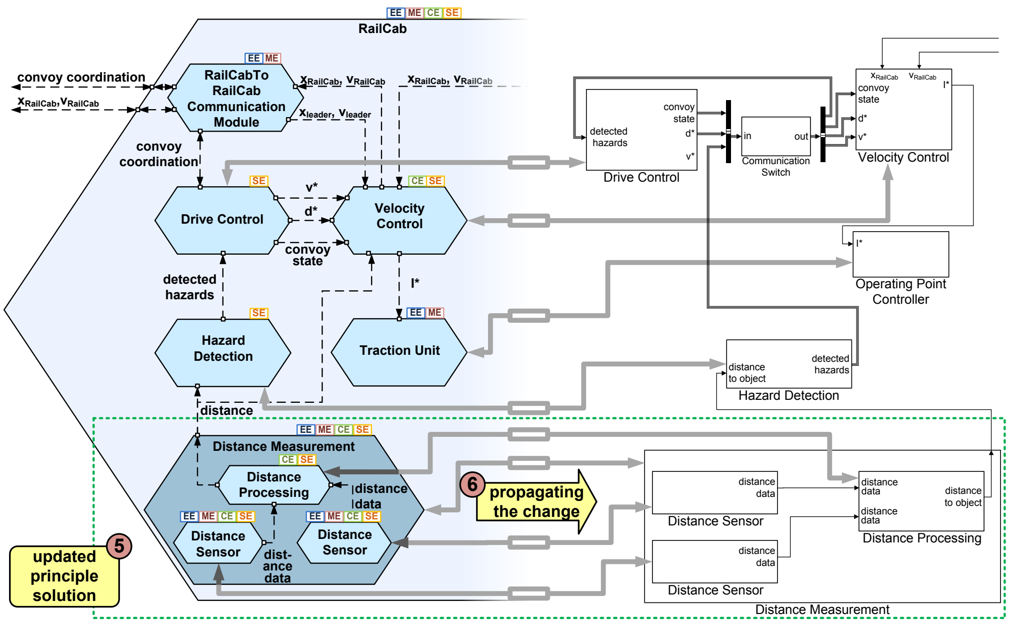


Figure 3.11: Updating the MATLAB/Simulink control engineering model using the updated active structure diagram

improved model transformation and synchronization technique. However, there might be cases where user decisions are indispensable, for instance when there are different possibilities to propagate a specific change, and the metrics do not provide a clear weighting. We describe this approach in detail in Chap. 4.

3.4 Comparison with Other Scenarios

Similar model transformation and synchronization scenarios appear in several development scenarios. For instance, modern software development is heavily based upon models, which are used throughout the whole software life-cycle. This is described with the term “model-driven engineering” (MDE).

Model-Driven Architecture (MDA) [OMG01] is an initiative of the Object Management Group (OMG) that uses such an MDE approach. In MDA, developers create different models of the software during the development process. Starting from platform-independent models (PIM) to platform-specific models (PSM) to the platform code, the level of abstraction continuously decreases.

The original MDA approach only uses one-time transformations when switching between the levels of abstraction, e.g., we transform from a PIM to a PSM only once. Changes may occur later to the PSM which would also affect the PIM. When developing just for one platform, it is not strictly required to update the PIM, although it is reasonable to do so for documentation purposes. More importantly, software is often to be deployed on several platforms – developers have to create different PSMs for all these platforms (cf. Fig. 3.12). Therefore, changes to a PSM should be propagated to the other PSMs via the PIM to keep the different platforms interoperable.

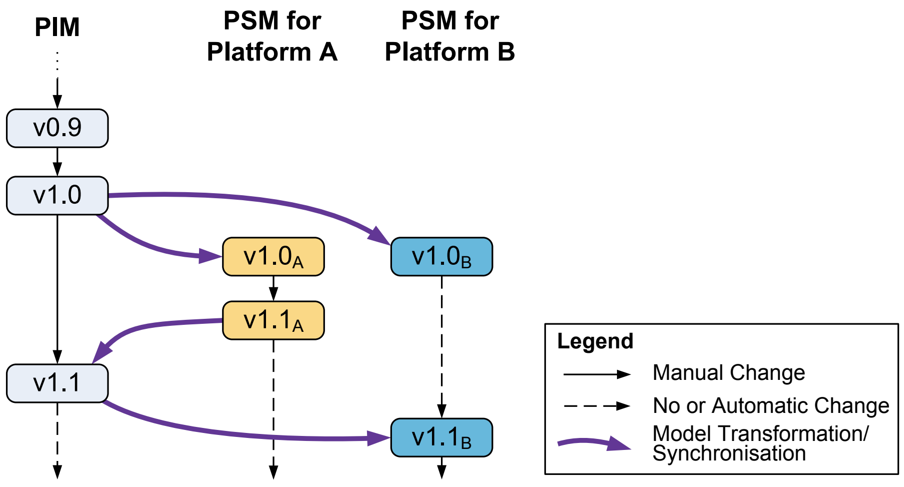


Figure 3.12: Models used in Model-Driven Architecture development

A simple example of such an MDA-like process is implementing object persistence, i.e., storing the objects of a program in a relational database management system (RDBMS). In this example, the PIM would be a class diagram of the classes that have to be stored in the database. Next, we transform this class diagram to a entity-relationship (ER) model, which serves as the PSM; this is where the actual object-relational mapping takes place. The PSM is then used to generate the (SQL) code for the respective RDBMS to create the different tables.

There are multiple ways to implement an object-relational mapping. The most common mappings are “table per class hierarchy”, “table per subclass”, and “table per concrete class”, but there exist several more [Fow03]. Each of these mappings validly maps a class hierarchy to a database schema and allows storing objects in a RDBMS. However, the different mappings have particular advantages and disadvantages regarding performance and memory efficiency.

For instance, “table per class hierarchy” just uses a single table for a class and all of its subclasses. All properties of all classes in this hierarchy will be added as columns in that table. This works efficiently for class hierarchies in which subclasses that only add methods, but no (or only few) additional properties. But when subclasses add a large number of properties, storing instances of the base class will waste memory, because the unneeded columns for the subclass properties will still use space.

Furthermore, not all object-relational mappings may be available for every RDBMS, because they may depend on features the RDBMS does not provide. As we would like to support several different kinds of RDBMS, we will need a PSM and, therefore, a transformation for each RDBMS.

For each RDBMS-specific transformation to the ER model, we have to define which types of mappings are available and which is the default mapping. Hence, the ER model (PSM) is more concrete than the class diagram (PIM); for instance, the software engineer may decide to use a performance optimized mapping only for “time-critical” classes. Such a refinement will not influence the class diagram. Nevertheless, when updating the ER model after changes to the class diagram, the refinement shall not be lost.

3.4.1 Summary

Model-based development scenarios typically have certain properties, which are

- models with differing levels of abstraction,
- models with differing viewpoints,
- information shared between several models, and
- changes occurring to these models at almost any time during the development.

Therefore, in all these development scenarios we face complex model consistency challenges.

In the next section, we present a novel approach for model transformation and synchronization. It explicitly supports mapping between models of different viewpoints and abstraction levels. It can be customized with respect to the scenario and in its amount of automation. Depending on the level of system and transformation engineers available, it may be run fully automatic or with more or less user interaction. Using metrics and heuristics, we encode expert knowledge and guidelines that have proven successful previously.

CHAPTER 4

Model Synchronization

Contents

In this chapter, we present the main contribution of this thesis. First, we introduce our novel, *improved incremental update algorithm* that prevents unnecessary loss of information in Sect. 4.1. Second, we describe a technique to allow incorporating *abstraction and refinement relations* that exist between models into the model transformation in Sect. 4.2. Third, we describe how to *synchronize concurrent changes* to both models in Sect. 4.3.

Parts of the techniques described in these sections have already been published at several conferences and other occasions [GSG+09, SEH+10, GPR11, GR12, RS12, RDS+12].

4.1 Incremental Updates

In the development process of mechatronic systems, developers use several independent, but related models. For instance, while the system model provides a discipline-spanning viewpoint on the system as a whole and also reflects the basic software architecture, it is not suited to design the details of the software. Thus, the system model is used to create a first version of the software model. As a result, the system model and the software model partially overlap in their information. In particular, structural/architectural information is subject to both models, but only the software model contains the specification of the detailed state-based behavior (including clocks, deadlines, and transition conditions). Figure 4.1 visualizes this. Whenever an engineer changes one of these models in the course of the development, we must ensure that these changes are also applied to the other model if they affect the overlapping parts.

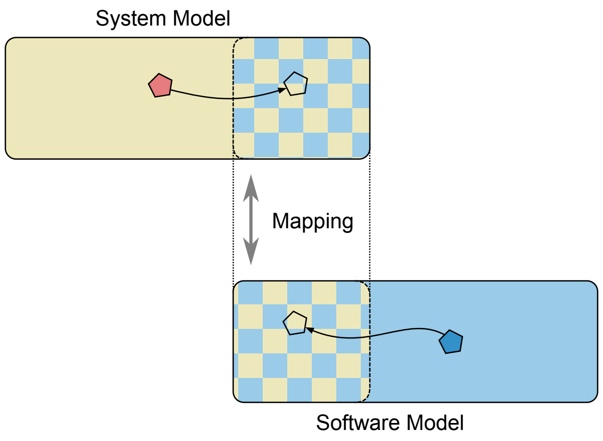


Figure 4.1: Partially overlapping models in the system development

Each of these models also contains information that is not *subject to the transformation* (cf. Fig. 4.1). This means that it is only contained in this single model, therefore also called *model-specific information*. For instance, a software engineering model contains implementation details of the software, which are – in general – unimportant for the systems engineering and for other disciplines. However, this discipline-specific information is still linked with other information in the discipline-specific model that is subject to the transformation.

In this section, we present an new algorithm to *incrementally update* related models if such a change occurs. We specifically designed this algorithm to allow dealing with models of different scopes or aspects, like the different discipline-specific models used in the development of mechatronic systems. As basis, we use an approach we developed in previous work [Rie08].

The general issue is that most incremental update algorithms do not deal with such model-specific information. More specifically, when updating models, they often unnecessarily delete such information, as they simply ignore the fact that there is information which is not in the scope of the transformation.[1]

Our algorithm solves this issue as follows. When propagating changes, our approach tries to *apply rules in a way that comes close to the state of the target model*, thereby retaining as much information of the target model as possible. More technically speaking, we do not delete elements right away when propagating changes, but instead we *mark them for deletion*. In this way, the algorithm can *reuse them later* in the incremental update process.

In general, there can be several different possibilities to reuse elements; these possibilities may vary in how much information they retain in a reasonable way. It is not always worthwhile to automatically decide on how to reuse elements. However, asking the user every time is also not a good solution. We use heuristics based on metrics to estimate the quality of the different reuse possibilities according to their success in retaining information. Basically, these metrics are a representation of expert know-how, as they formalize reuse guidelines that have proven successful previously. We discuss when these heuristics can be automatically used and where expert decisions are indispensable. In particular, our approach can be tailored from performing a fully-automatic incremental update to a complete user-interactive proceeding.

4.1.1 Related Work

After running an initial (batch) transformation, the related models are linked with each other. I.e., there exists a triple of corresponding models (the source model, the correspondence graph, and the target model). When a change occurs in one domain model, this change can be propagated by re-running the complete transformation from scratch, replacing the "opposite" domain model.

Re-running the entire transformation is very costly and, more importantly, specific information that was added to the target model would be lost. Instead, it is more reasonable to *incrementally update* only the affected parts of the model, and to keep the existing, unaffected parts. Algorithms for this problem have been described before [GW09, GH09, XSHT09, HLR06]. In the following, we explain existing incremental update approaches and their shortcomings.

First, we describe how model-to-text transformations implement incrementality. Next, we focus on model-to-model transformation approaches that (according to the classification of Czarnecki and Helsen, cf. Sect 2.2.2) follow a declarative paradigm, are able to update existing targets and to propagate changes, and have pattern-based rules.

[1]Note that such algorithms are correct, anyway, because transformation correctness is defined in terms of the transformation relation between the models. Thus, information that is not subject to the transformation will never affect correctness of the transformation (algorithm).

4.1.1.1 Model-to-Text Transformations

Incremental updates are often used in model-to-text transformations, e.g., for generating source code. A typical example is code generation from class diagrams. Class diagrams mainly contain information about properties and methods of classes and inheritance. After generating code, software engineers implement the behavior of each method manually using the generated source code. When the class diagram is changed, regenerating the code must not overwrite the manually implemented methods. The solution that is most frequently used is *user-editable blocks*. In the case of code generation from a class diagram, each method body is defined as such a user block. When regenerating the code, this block remains untouched.

While this works well for simple changes (like adding a method), it does not work more complex refactorings (like renaming a class or moving a method between classes). Moreover, it is generally not suitable for models due to the (typically large) amount of links in a model; a model could easily become invalid when simply ignoring parts of it.

4.1.1.2 Goldschmidt and Uhl (2008/2011)

Goldschmidt and Uhl [GU08, GU11] suggest using so-called retainment policies. Retainment policies specify "how a transformation rule should handle manual changes in target models." [GU08] The approach is implemented for QVT-R [OMG08], but will work for most transformation approaches that consist of transformation rules and use a declarative logic (cf. Sect. 2.2.2). They present a classification of changes that may occur, like additions or deletions of elements. Based on this classification, policies specify possible retainment actions for types of changes. Possible actions are: overwriting manual changes, only updating when there was no manual change, always retaining the manual change, or never updating the target. A retainment policy has a scope, which defines for which rules this policy should be applied. In situations where a policy advices to retain the target, it basically switches off the respective transformation rules.

The retainment rules of Goldschmidt and Uhl operate on the level of transformation rules. Thus, defining retainment rules requires knowledge of the ruleset. This approach is similar to manually defining user-editable blocks in model-to-text transformations, but provides a more fine-grained control. When larger change sets are propagated, however, this approach will likely cause information loss (in situations where no policy advises a retainment) or generate inconsistent models (due to mappings that are switched off by a retainment policy).

4.1.1.3 Giese and Wagner (2008/2009)

Giese and Wagner suggest the following algorithm [GW09] (see also Sect. 2.4.3.1 and Fig. 2.29). Let us assume we have a valid model triple, i.e., the source and target model and the correspondence graph are consistent according the TGG. When a change occurs in the source model, there may be rule applications that are no longer valid, because its graphs do not match any longer, or its attribute constraints or application conditions are violated. In this case, the rule application(s) that are violated by the change are *revoked*.[2] Revoking a rule application means that the target model elements created by the revoked rule applications are destroyed, and the source model elements become unbound. There may be rule applications that depend on the revoked rule application(s), i.e., rule applications that have one or more model elements that have been destroyed or unbound by the revocation. These rule applications are also revoked, potentially leading to a huge chain of necessary revocations.

Finally, the transformation is re-run for the unbound source model elements. During that re-run, new model elements may be created in the target model, as new rules are applied at the previously unbound source model elements. In this way, existing information that is not subject to the transformation is only retained in model parts that are untouched by the incremental update.

4.1.1.4 Giese and Hildebrandt (2009)

Giese and Hildebrandt present another algorithm [GH09] that builds upon the algorithm of Giese and Wagner described before. It tries to deal with the problem of chained rule revocations, however, mainly considering performance aspects. When there is a long chain of rule revocations, it often happens that the rules that are then applied are the same that were applied before. This is because changes are often local and only affect a few rules directly.

Observing this, Giese and Hildebrandt suggest that when a change occurs in the source model, the algorithm should try to repair the rule applications that are violated by the change. For instance, if the change is a move of an element on the source side, a rule application can be repaired by changing the link from the corresponding (target-side) element to its old (target-side) parent such that it is now linked to the new corresponding parent on the target side.

In their approach, pre-generated repair operations that are automatically derived from the TGG rules attempt these repairs. Only if such a repair operation is not possible, the rule application is revoked. In this case, the algorithm tries to apply another rule to these unbound source model elements.

However, these repair operations are only able to modify links. Whenever it is not possible to repair a rule application by changing a link, the rule is revoked and the target model elements are destroyed and cannot be reused anymore. For instance, this is the case when new elements have to be created to make another rule applicable. Thus, this approach can only prevent revoking rules in simple cases, but does not provide a general solution to the problem.

[2]Attribute changes can sometimes be propagated without revoking the rule application. Thus, the algorithm by Giese and Wagner checks whether the violation is just due to a changed attribute. If this is that case, it re-evaluates the attribute constraints of the rule and assigns them to the target model attributes. In this way, a costly revocation of the complete rule application can be avoided. Our algorithm uses a similar optimization.

4.1.1.5 Körtgen (2009/2010)

KÖRTGEN [Kör09] also describes an approach for dealing with incremental updates. She first presents a categorization of different inconsistency types that user modifications to a model can cause. Based upon this categorization and given the ruleset, she derives pre-computed repair operations, each of which can deal with a specific inconsistency type in a certain situation and a certain rule.

When changed models are synchronized, the different pre-computed repair operations are checked for applicability stepwise for each changed model element. Typically, several repair operations can be applied for a single change. In Körtgen's approach, the user has to decide every time more than one repair operation is possible. Gradually, the approach guides the user through all decisions, finally ending up in a consistent state of the models again.

Although the approach provides change operations for each type of change in the categorization, it remains unclear whether this categorization is complete. In other words, the approach covers usual changes that can happen to a model by providing corresponding operations, but no prove is given that it is able to cover all changes that may happen, especially for complex modeling languages.

Furthermore, the approach requires a large amount of user-interaction and a tight integration with the model editing tools in use. KÖRTGEN, however, mentions adding certain analyses to reduce the amount of user-selectable repair operations as a possible extension to the approach [Kör10].

4.1.1.6 Model Merging

Instead of running an incremental update, we could completely retransform the source model and thereby create a new target model afresh. This target model will obviously not contain any model-specific information. We could use a model merger to combine the freshly created target model with the previous target model. Conceptually, this will also create a target model that still contains model-specific information.

However, there are severe drawback to such a approach. Most importantly, model differencing and merging still is a difficult problem in practice, and existing tools suffer from bad merging results especially when the models and their differences become large [ELHN+10, LAS+14]. The model merger also cannot simply use the correspondence or source model to help improving the merging process. Moreover, running a complete re-transformation takes much more time than just incrementally updating changed model parts. A TGG transformation has worst-case runtime of $O(n^{k \cdot l})$, where n is the size of the host graph, k is the number of rules, and l the maximum rule size (cf. Sect. 2.3.3). An incremental update only has to check changed model elements; its worst-case runtime is therefore depends on the number of affected elements[3] n_a: $O({n_a}^{k \cdot l})$. Also the model merging gets slower when models are large.

[3]Note that "affected elements" is not the same as "changed elements": A (user) change can affect several more elements in a model transformation scenario, when it requires revoking rule applications which, in turn, cause other rule applications to be revoked.

4.1.1.7 Summary

Many existing approaches are not able to deal with complex changes and therefore cause loss of information in the target model. KÖRTGEN's approach [Kör09] can also deal with changes that are more complex. However, as the models to be synchronized differ in their viewpoints and abstraction levels, allowing for complex changes in a incremental update algorithm quickly leads to the need for making decisions how to propagate a certain change. KÖRTGEN's approach therefore requires the user to decide at every atomic decision point, leading to a high amount of fine-grained user interaction. This represents one extreme; the other is full automation, where the algorithm makes every decision.

Although we could aim for designing an algorithm that does the model synchronization fully automatically, this is not reasonable in most scenarios. Engineers should have control over all design decisions in their models, and this includes changes that are caused by other disciplines, when several alternatives are possible to implement this change. On the other hand, deciding at each atomic step of an incremental update – besides the high effort – easily leads to wrong choices, because the decisions are performed for atomic modeling elements (objects and links): Engineers may not foresee the consequences of a single decision, as they do not see the effect to larger, semantically connected units.

In the next section, we present an incremental update algorithm that tackles these drawbacks. In previous work, we presented the idea of *reusing elements* when revoking/applying rules, *instead of immediately deleting them* [Rie08]. This previous approach focuses primarily on a certain scenario (when model elements are hierarchically restructured or moved, i.e., get a new containing parent model element) and therefore lacks general applicability. It does not satisfy the completeness property for transformations with conflicting rules. Furthermore, it requires manual adaptation of the TGG ruleset by the transformation engineer. However, this previous algorithm is the basis for the approach presented in this thesis. In particular, we generalize it to make it applicable for all kinds of changes, remove the need for manual ruleset adaptations, extend it with means for incorporating expert knowledge, and allow customizing the automation amount.

4.1.2 General Approach

We present an improved incremental update algorithm

- that is able to deal with cases of complex changes,
- in which user decisions are not based on atomic elements but on larger semantic units,
- that incorporates expert knowledge encoded in metrics, and
- that is customizable in its amount of automation.

It generalizes the idea of structurally repairing rule applications such that complete structures can be repaired, independent of what kind of changes appeared. Nonetheless, it neither changes the general TGG semantics nor impairs the formal properties of a TGG.

When incrementally updating, there may be several alternatives to propagate a change. Our algorithm first calculates these different alternatives. As quality estimation heuristics, it uses (customizable) metrics that assign weightings to these alternatives. These weightings reflect how good a certain propagation possibility is in restoring the consistency without loss of data.

The algorithm may be run in full-automation mode, where it automatically chooses the "best" way to synchronize changes based on that weighting; i.e., it selects the propagation option with the best weighting. It may also run in user-interaction mode, where it asks the user for alternatives whose weightings are within a given threshold. Without a threshold, it offers fully manual change propagation, comparable with the user interaction of approach like KÖRTGEN [Kör09]. In this way, users can influence the amount of interaction according to their trust in the heuristics.

Figure 4.2 shows the activities of the algorithm. The basic loop (top of Fig. 4.2) is similar to the activities of most incremental update approaches, e.g., GIESE AND WAGNER [GW09] (cf. Sect. 2.4.3.1 and Fig. 2.29). The main extensions and improvements constitute themselves within the action apply rule (bottom of Fig. 4.2).

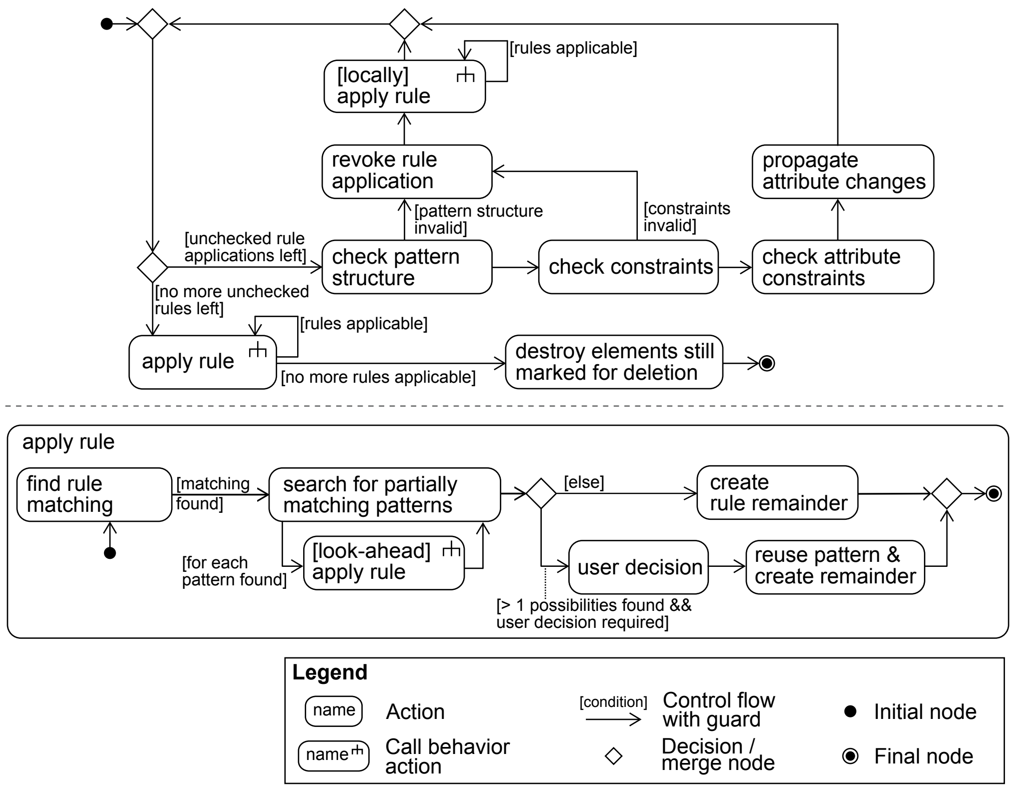


Figure 4.2: Activities of the new incremental update algorithm

The algorithm works as follows. In the first phase of the algorithm – similar to the approach GIESE AND WAGNER [GW09] –, we check for each rule application whether it may have become invalid. This includes checking the pattern structure and the constraints. Checking constraints is separated into checking for general context constraints and attribute constraints. In contrast to attribute constraints, general context constraints cannot be re-enforced; a violation therefore results in a rule revocation. Each rule application that cannot be repaired by propagating attribute changes is revoked.

As the deletion of elements should be prevented if possible, it is not reasonable to immediately destroy elements when a rule application is revoked, as we describe in previous work [Rie08]. When a rule application is revoked, we remove the bindings for all (context and produced) model elements on the one hand. On the other hand, and in contrast to other approaches, we do not actually delete the produced model elements, but *mark them for deletion.*[4]

Our algorithm can reuse these removed and marked objects and links during later rule application by explicitly *searching for matches in the set of elements that are marked for deletion.* (In order to reuse elements, the new rule of course must have an intersection with the revoked rule in the produced correspondence and target graph.) In particular, after revoking a rule application, the algorithm immediately checks if other rules are applicable as a replacement for the revoked rule. This avoids revoking subsequent rule applications that depend on the revoked rule in many cases – if we find another matching rule, the context of subsequent rule may still be valid for the new rule.[5] After that, the algorithm continues checking existing rule applications.

In the second phase, the algorithm tries applying new rules. Here, we again use the set of elements that are marked for deletion to find suitable objects, so that we do not have to create new objects. Only if deletion-marked elements cannot be reused (i.e., bound again) by a new rule application, they will be ultimately destroyed. As described before, the problem is that there may be several ways of how the elements marked for deletion can be reused. A particular challenge is therefore to determine the "best" way to reuse these elements.

The core of the approach is the search for partially matching patterns, which is described in detail in Sect. 4.1.7. It identifies which elements could potentially be reused. Optionally, we can perform a *look-ahead* if there is more than one possibility of reusing elements. In this way, we also identify effects of a particular reuse possibility on further rule applications. We use metrics to estimate how well the possibilities preserve information; for ambiguous cases, we ask the user to select one (see Sect. 4.1.6 for details).

Next, we present an example and overview our incremental update algorithm. Then we give an extended example with different ways of reusing elements and we discuss metrics to determine which reusable pattern may be best. Finally, we discuss the details of the *partially reusable pattern search.*

[4]At first glance, it seems that such a proceeding changes the TGG semantics. In Sect. 4.1.8 we argue why this is not the case. Furthermore, existing transformation specifications (TGG rulesets) do not have to be modified for our approach to work.

[5]Note that this is mainly an optimization: Even if we would revoke all dependent rule applications, we could still reuse their elements later in the second phase. This would yield the same results. However, the computational complexity would increase due to a larger set of elements marked for deletion. Furthermore, there are more ways of reusing elements, which may result in a greater chance of selecting a bad one. Our evaluation shows that such a procedure will in fact improve the runtime for some cases (cf. Sect. 6.3.1).

4.1.3 Example

The example we use in this section is based on the general exemplary process described in Sect. 3.1 and shown in Fig. 3.2. Figure 4.3 shows the parts of this general process that are relevant to this section.

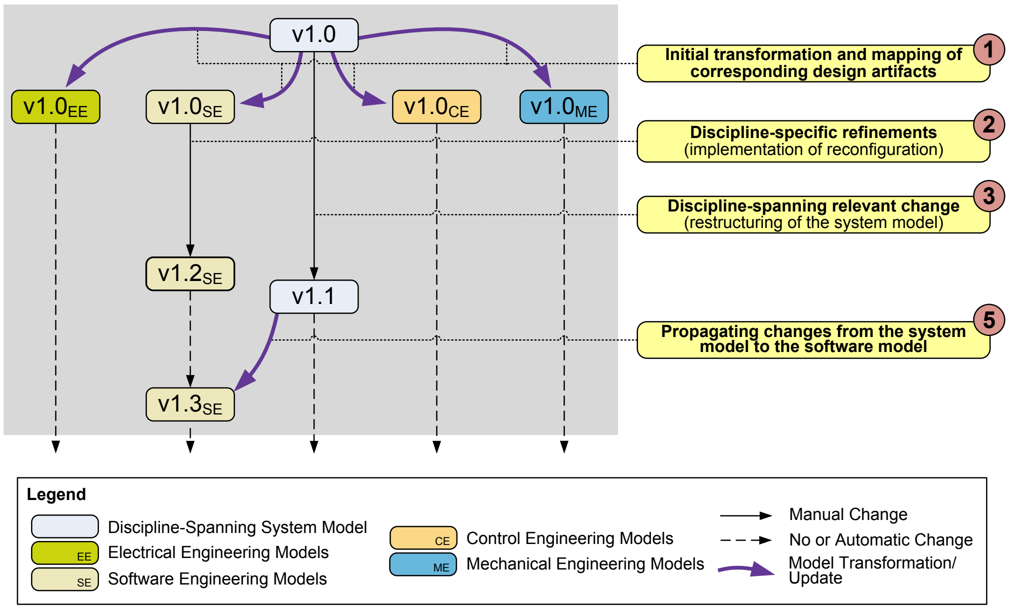

v1.0
v1.0EE
v1.0SE
v1.0CE
v1.0ME
v1.2SE
v1.1
v1.3SE
1 Initial transformation and mapping of corresponding design artifacts
2 Discipline-specific refinements (implementation of reconfiguration)
3 Discipline-spanning relevant change (restructuring of the system model)
5 Propagating changes from the system model to the software model
Legend
Discipline-Spanning System Model
EE Electrical Engineering Models
SE Software Engineering Models
CE Control Engineering Models
ME Mechanical Engineering Models
Manual Change
No or Automatic Change
Model Transformation/ Update

Figure 4.3: Evolution of different models during the development process (excerpt of Fig. 3.2)

After the initial transformation to the discipline-specific models has been performed in step 1, our models are in a consistent state, as depicted in Fig. 4.4. In particular, all elements are bound by TGG rule applications.

Next, the engineers from the disciplines start refining their models in step 2. In particular, a software engineer adds a reconfiguration chart to the MechatronicUML model. This change only affects software engineering, i.e., it is irrelevant to other disciplines and does not have to be propagated.

In parallel, a systems engineer deletes the information flow convoy state to the system element velocity controller from the CONSENS model, and with it its corresponding port (step 3 in Fig. 4.5). In contrast to the change in the software mode, this change does have an impact on other disciplines. For instance, the velocity controller in the software model must not be a hybrid component any more, but has to become a controller component.

We propagate this change to the software model in step 5 as follows. The rule applications that previously translated the (now deleted) information flow and the information flow port are now structurally invalid.[6] Therefore, these rule applications have to be revoked. All its bindings are deleted, and the

[6]We refrain from showing these TGG rules, as they simply map one information flow (resp. one information port) to one connector (resp. one port). The complete ruleset can be found in Appendix A.

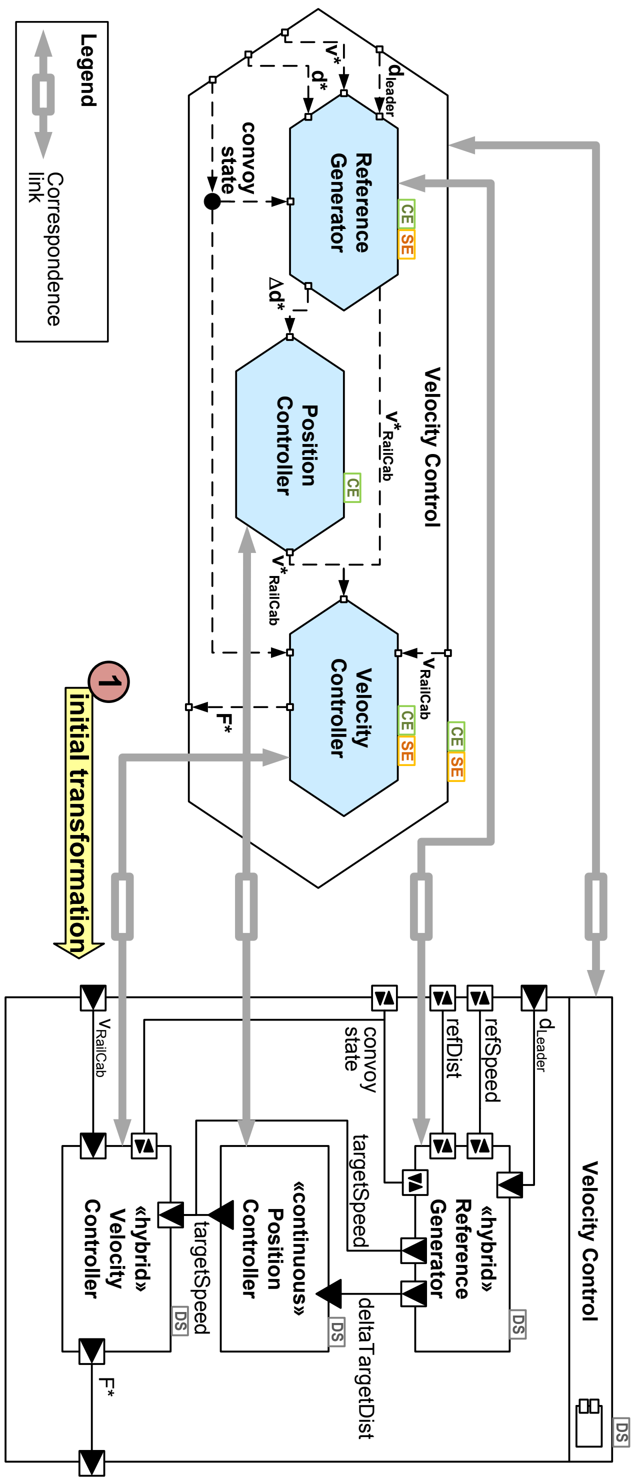


Figure 4.4: Initial transformation from the active structure to a software component diagram, here: transforming the Velocity Control system elememt

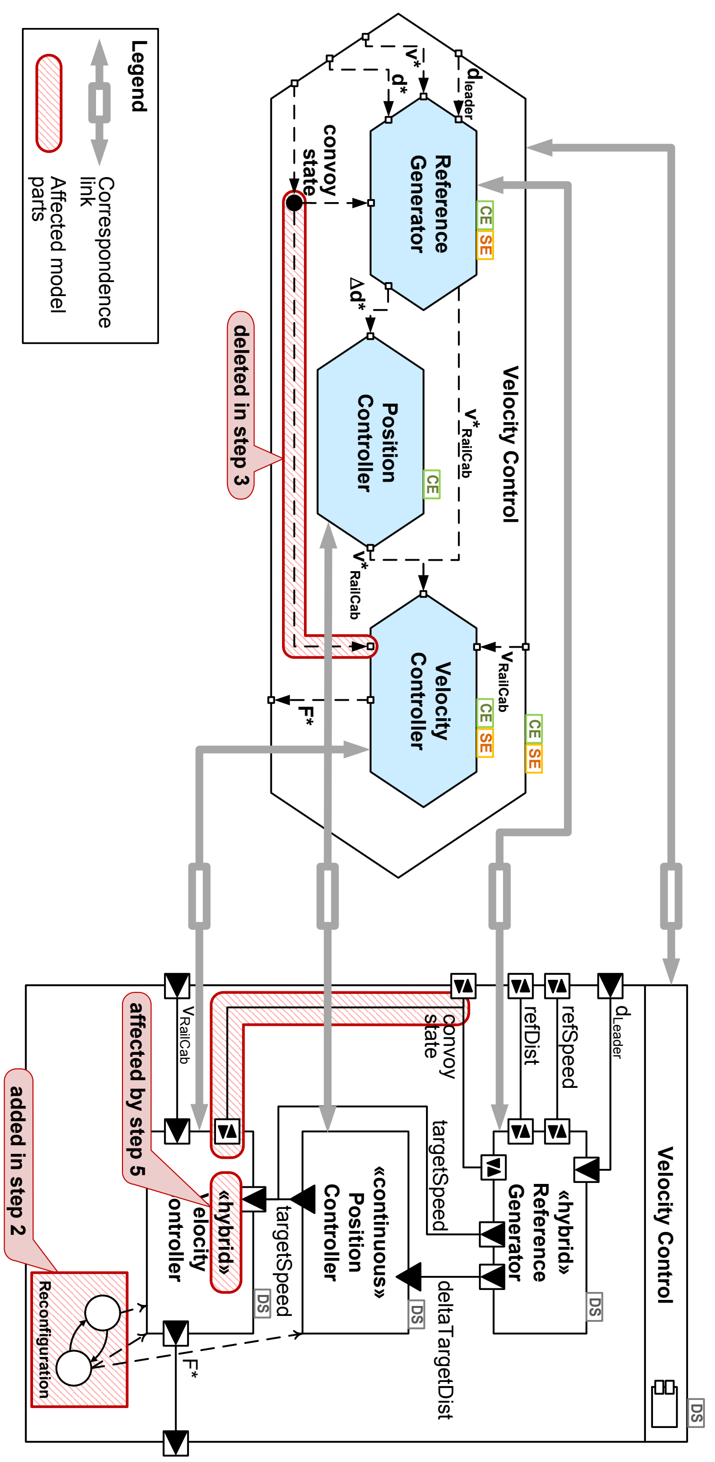


Figure 4.5: Changes to the CONSENS system model and the MechatronicUML software engineering model (step 2) after the initial transformation

correspondence and target produced objects and links are marked as deleted (denoted by dashed lines in Fig. 4.6). Figure 4.6 shows the abstract syntax of the situation after the rule revocation.

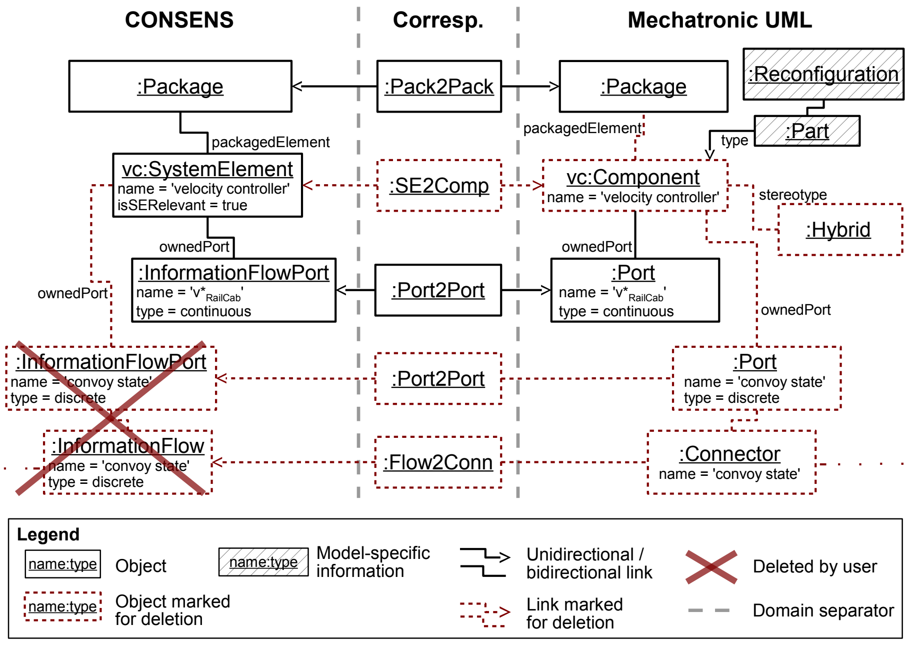


Figure 4.6: Abstract syntax after rule revocation due to deletion of a flow

The application of SystemElementToHybridComponent (Fig. 2.26) that mapped the velocity controller system element to a hybrid component also becomes invalid: `continuousPorts>0&&discretePorts>0` is violated, as there is no discrete port any more. So this rule application has to be revoked by marking its produced part (vc:Component and :Hybrid) as deleted, too.

The produced parts of the revoked rule applications served as context for other rule applications. Traditional incremental update algorithms would also revoke these dependent rule applications. Instead, we try to apply new rules immediately as a replacement for the revoked rule application. By doing so, we may be able to restore the context of depending rule applications, thereby avoiding their revocation. Besides the immanent performance increase, this reduces the risk of information loss, because less target elements are affected.

In this case, we can apply rule SystemElementToController (see Fig. 4.7). Its context is matched onto the package objects in CONSENS and MechatronicUML and the correspondence :Pack2Pack. Incrementally updating in forward direction, the produced source (CONSENS) pattern (se:SystemElement) is matched onto the velocity controller system element, as this element is now unbound due to the revocation of the rule SystemElementToHybridComponent previously. Also the constraints hold, as they require no discrete port. A normal rule application would simply create the correspondence and target patterns. Instead, our incremental update first searches for a pattern matching in the set of elements marked for deletion. Two elements in this set can be reused: Starting the search from the velocity controller system element, our algorithm finds and reuses the (deletion-marked) :SE2Comp correspondence and the velocity controller component.

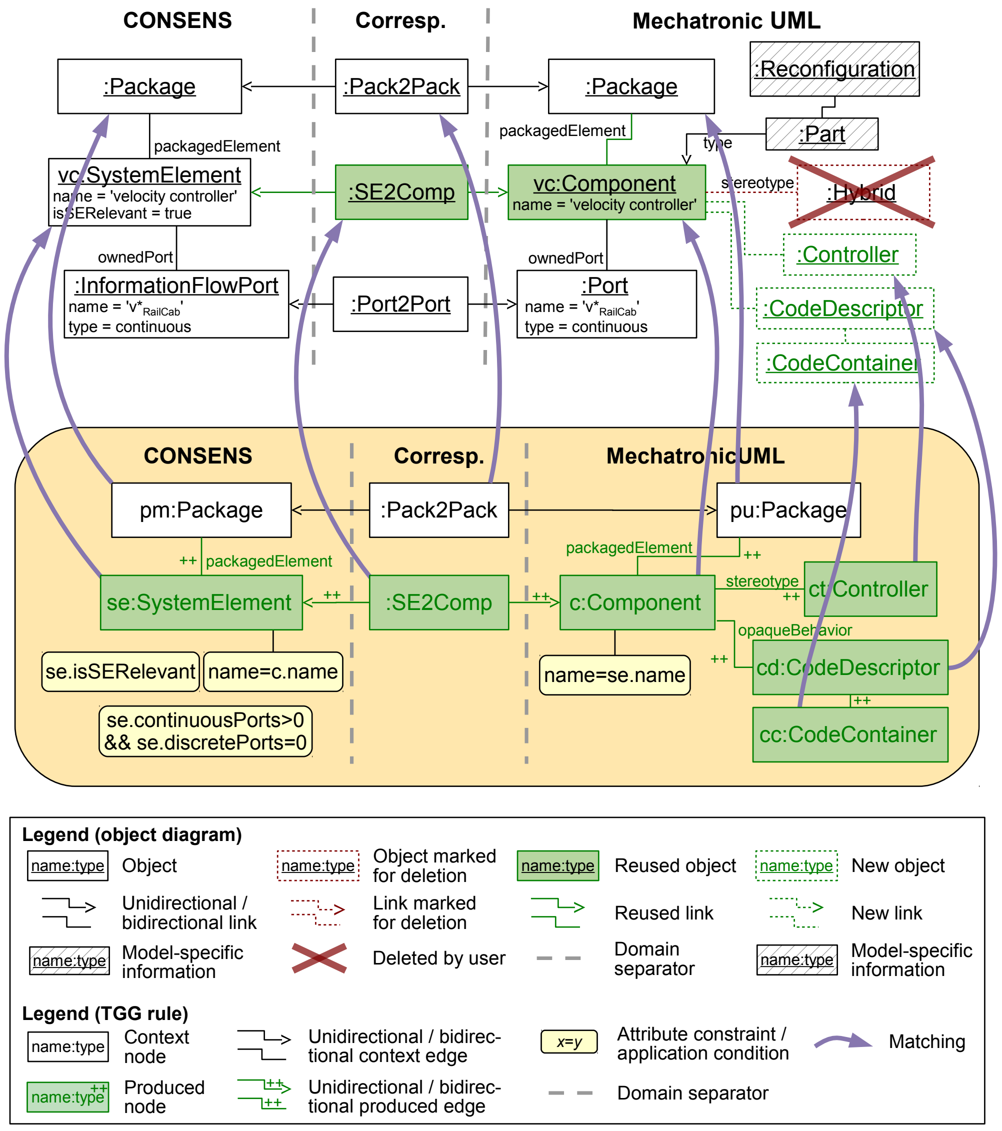


Figure 4.7: Abstract syntax after applying the rule SystemElementToController (with reusing elements)

No other previously deleted element can be reused by this rule. However, the matching is not complete yet, as there are no existing objects to match the Controller, CodeDescriptor, and CodeContainer nodes of rule the SystemElement-ToController. The algorithm uses this *partial pattern matching* anyway. Now we can apply the rule as follows. First, removing the "deleted" flag from everything that has been reused and binding these elements again. Second, because the match of the TGG rule is not yet complete, additional links and objects are created. Non-fitting links of single-valued references are moved. In this way, the target model is modified so that it matches the rule that is applied. We call this process *repair operation*.

Figure 4.7 shows the situation after the rule application. The algorithm reused the :SE2Comp correspondence and the velocity controller component (shaded in Fig. 4.7). It created new instances of Controller, CodeDescriptor, and CodeContainer, and set the appropriate links (dashed in Fig. 4.7), as no reusable element could be found for them.

The Hybrid stereotype object, which was deleted when revoking the previous rule application, could not be reused. Thus, the algorithm ultimately destroys it (crossed-out in Fig. 4.7).

By reusing the velocity controller component, which was previously marked for deletion, the algorithm prevents a dangling edge from the model-specific reconfiguration specification (hatched in Fig. 4.6 and 4.7). It also preserves any further model-specific information that is attached to the component (for instance, safety requirements). Furthermore, a deletion and recreation of the component would have rendered the context invalid for the PortToPort rule that translated the v^*_{RailCab} InformationFlowPort. Thus, traditional incremental update algorithms would have to revoke this and possibly further dependent rule applications.

4.1.4 Concept of the Incremental Update Algorithm

Let us assume we are propagating changes from source to target (forward incremental update). Then, in summary, our algorithm works as follows:

1. Iterate over all TGG rule applications in the order they were applied, and if the application has become invalid due to changes in the source model
   a. Remove the bindings of the produced graph, and mark the correspondence and target produced elements previously bound to this graph as deleted.
   b. If the same or other rules are applicable in the forward direction, i.e., the context and the source produced graph match,
      i. search for a pattern of elements marked for deletion that "best" matches part of the rule's correspondence/target graph structure
      ii. apply the rule by reusing this pattern, creating the remaining correspondence/target pattern, and enforcing attribute value constraints.

   Continue checking the next rule application or terminate if all applications have been checked.

2. Finally, effectively destroy elements that are still marked for deletion.

The concept to "mark for deletion" allows us to remember the elements that might be reusable. These elements can then be reused in step 1b-ii, and only the remainder of the pattern has to be created. This concept is the basis for intelligently reusing model elements during the update.

In the example above there was only one possible partially reusable pattern. Often there are several available partial matchings that reuse more, less or other elements. In fact, some partial matchings may reuse elements in an unintended way. Therefore, we calculate all possible partial matchings. We use heuristics to choose the most reasonable of these reuse possibilities. In the following, we present an extended example in which two reuse possibilities occur. Next, we discuss the implementation details of the partially reusable pattern search, which computes the different reuse options (step 1b-i).

4.1.5 Selection of Elements to be Reused

The heuristic for the "best" partial matching is generally to take the partial matching that reuses most elements. However, also considering existing correspondence information and object properties can be vital, as we show in the following example. Let us assume that we start with a consistent state, as in the previous example, i.e., all model elements are bound by rule applications.

First, an engineer deletes the discrete port and its information flow from the CONSENS velocity controller (as in the previous example). Second, the engineer removes the position controller's relevance flag for the software engineering discipline. Therefore the application of rule SystemElementToHybridComponent for the velocity controller and the application of rule SystemElementToController for the position controller is revoked. Figure 4.8 shows the result in form of an object diagram, with user deletions crossed-out and deletion-marked elements depicted by dashed lines.

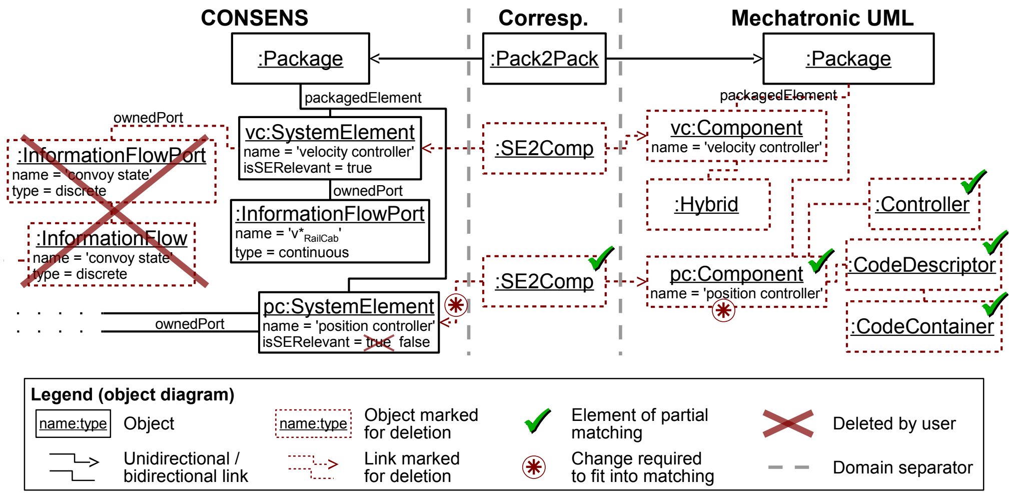


Figure 4.8: Revocation of two rules and a subsequent "wrong" partial pattern match

Again, rule SystemElementToController is now applicable at the CONSENS velocity controller. When trying to apply this rule, our algorithm searches for partial matches in the set of deleted elements. In addition to the partial pattern matching of the previous example (see Fig. 4.7), there is a second promising partial matching here. It is marked with $\surd$s in Fig. 4.8. The partial pattern matching search finds it by starting from the context :Package object of MechatronicUML. This partial matching reuses the deleted position controller component, its controller, code descriptor and code container. However, the existing correspondence node does not fit (marked with a $\circledast$): It points to the CON-

SENS position controller, but it must be connected to the CONSENS velocity controller (because the node se:SystemElement is already matched to the velocity controller). Additionally, the attribute constraint must be repaired, changing the component's name to "velocity controller" (also marked with ⊛).

Although this alternative partial matching reuses more elements than the partial matching in Fig. 4.7, it is in fact an example where the reuse is unintended: Using this partial matching would create a correspondence between elements that did not correspond to each other before. At first glance, this may not seem to be a problem, because the change propagation will adapt all links and attribute values in the target model to satisfy the constraints posed by the TGG. Thus, using this matching will also lead to a consistent state. However, as already described before, the target model may contain elements that reference these reused elements, but that are not subject to the transformation. A "wrong" reuse means that these elements now reference completely altered objects that have changed their semantics significantly. Therefore, it is reasonable to favor such partial matchings where a correspondence node is reused without changing its correspondence links. In this way, only previously corresponding elements are reused, typically resulting in the intended reuse of elements.

Note that in terms of the TGG semantics it is not relevant which of these alternatives is actually selected, i.e., in which way the algorithm reuses previously deleted elements (or whether it reuses elements at all). In every case the synchronization results in a consistent rule application. A particular reuse of elements may only be more or less harmful to the elements that are not subject to the transformation. See Sect. 4.1.8 for further information on the formal properties of our algorithm.

4.1.6 Selection Metrics

The reuse possibilities may vary in how much information they retain in a reasonable way. In this section, we describe the metrics we use to order the reuse possibilities according to their success in retaining information. These metrics represent the knowledge of experts, as they formalize reuse guidelines that have proven successful previously.

Simply counting the number of elements that will be reused is not the best way of selecting a reuse possibility, as described in the previous section. Therefore, our algorithm uses several distinct metrics to compute how to reuse elements. In our experiments, we have determined default values for these metrics that performed well in our scenario. However, as other scenarios may differ, the metrics can be customized. Table 4.1 shows the metrics that measure the reuse quality. A lower value corresponds to a higher quality.

Furthermore, we allow different amounts of user interaction when making a decision about the reuse. In particular, our approach can be tailored from performing a fully-automatic incremental update (i.e., relying completely on the metrics) to a complete user-interactive proceeding. We use a concept of *certainty* here: The more distant the (metrics-measured) quality of two reuse possibilities is, the more certain we are which one of them is better suited.

Table 4.1: Metrics for measuring the quality of a reuse possibility

Name	**Description**	**Default Value**
Object Creations	The number of (target model) objects that will be created in order to complete the produced pattern of the rule.	0.5
Attribute Modifications	The number of attributes (of target model objects) that will be changed.	0.5
ID Modifications	The number of identifying attributes (of target model objects) that will be changed. Identifying attributes are attributes like qualified names or unique identifiers.	1
Link Creations	The number of (target model) links that will be created in order to complete the produced pattern of the rule.	0.5
Link Modifications	The number of (target model) links that will be altered.	0.75
Correspondence Modifications	The number of links from the correspondence model that will be altered.	2
Disconnected Components	The number of reused model parts with no links to other reused parts.	4

The thresholds that are used to determine whether to ask the user are summarized in Tab. 4.2.

As an example, consider the following situation. There are two reuse possibilities; the first reuses more elements and is therefore considered "better" than the second. However, if both weightings are similar, it may be reasonable to ask the user. We use an adjustable threshold "Good Quality" to determine in which cases the user has to decide which reuse possibility to use. In other words, this threshold determines how close the estimated quality of both reuse possibilities has to become in order to not to rely on this metrics-based quality measurement. Its default is $qu_{rel}(p) < 0.2$, i.e., we ask the user to decide between all possibilities whose quality is within the best 20 %.

Such a threshold accounts for the inherent uncertainty of an automated, heuristics-based approach. On the other hand, the metrics have to be related to the actual quality; otherwise, such a threshold is effectively useless. We have evaluated our metrics using several examples. Nevertheless, further evaluations seem reasonable as future work, especially in application scenarios other than mechatronic system design and model-driven software development.

It may still happen that target model elements are deleted eventually, i.e., their information is lost. This can be intended, e.g., in cases where elements in the source model have also been deleted. However, it may also be due to wrong (automatic or manual) reuse decisions.

We analyze the amount of information loss in the target model at the end

Table 4.2: Thresholds for manual decisions

Name	**Formula/Description**	**Default Threshold**
Good Quality	$qu_{worst} = max(p \in P : qu_{abs}(p))$ $qu_{best} = min(p \in P : qu_{abs}(p))$ $qu_{rel}(p) = \frac{qu_{abs}(p) - qu_{best}}{qu_{worst} - qu_{best}}$ where P is the set of reuse possibilities, and $qu_{abs}(p)$ is the sum of all metrics (cf. Tab. 4.1) Let the user decide between all reuse possibilities $p \in P$ whose quality is close to the maximum quality.	$qu_{rel}(p) < 0.2$
Relative Information Loss	$il_{rel} = \frac{del_t}{unbound_s}$ where del_s is the number of deleted target elements, and $unbound_s$ is the number of newly unbound source model element Eventual loss of information in the target model due to deletion of elements in relation to uncovered source model parts due to user changes	$il_{rel} > 0.8$
Reuse Ratio	$rr = \frac{reuse_t}{reuse_t + create_t}$ where $reuse_t$ is the number of reused target elements, and $create_t$ is the number of created target elements Eventual ratio of reused target model elements (prevented loss of information)	$rr < 0.5$

of the incremental update process and compare it to the information loss in the source model (threshold “Relative Information Loss”). The rationale behind this is that the greater the information loss in the target model is compared to the loss in the source model, the more likely it is that a previous decision could have been wrong. Furthermore, we compare the amount of reused target model elements with the amount of newly created elements (“Reuse Ratio”). If only few elements are reused, this also indicates a bad reuse decision. If both conditions are met, we ask the user if he/she would like to revise previous reuse decisions[7].

[7]At the time of that earlier decisions, we cannot predict whether a certain decision will negatively affect future reuse possibilities without calculating *all* reuse possibilities of the *complete* incremental update run. Although it is theoretically possible to do so, we did not implement such an approach due to the huge negative impact on performance: The runtime and space complexity are exponential in the number of elements in every single rule, and – on top – exponential in the number of rules to reapply.

4.1.7 Partially Reusable Pattern Matching Algorithm

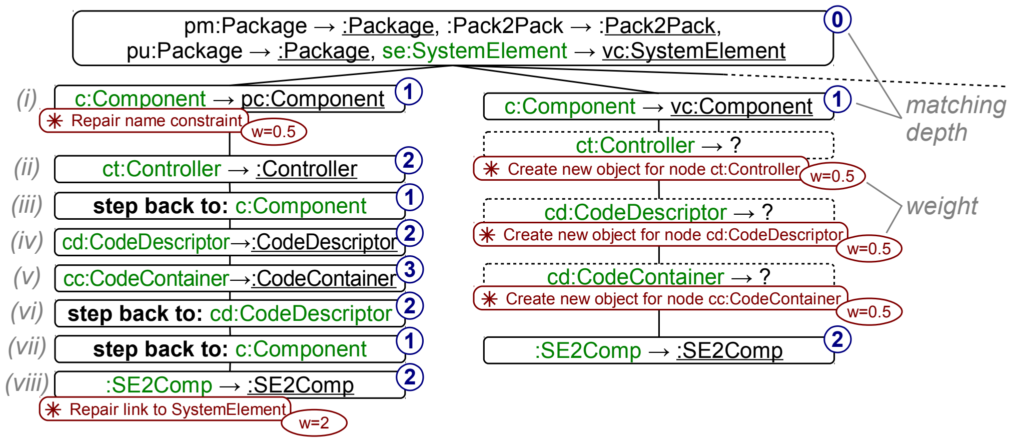
In the following, we discuss how the algorithm searches for partial matchings and describe the data structure the algorithm builds during the partially reusable pattern search.

The algorithm computes all possible partial matchings by creating a rooted tree structure. Each vertex of the tree represents a partial matching. Each edge of the tree represents a step of the pattern matching which binds a new node.

The partially reusable pattern matching algorithm starts with the matching of the context and source produced domain pattern (computed in step 1b). Thus, the root of the tree is a vertex that represents this matching. Each other vertex is labeled with a single node binding (a node-object tuple). Each vertex of the tree therefore represents a (partial) matching of the rule, recursively defined by the node binding of the vertex and those of its parent. Additionally, a vertex is labeled with its pattern matching *depth*, which is the depth in the recursion of a depth-first pattern matching algorithm, i.e., how long the taken path in the TGG rule from the start of the search to the current node is.

The resulting tree reflects the pattern-matching search: When traversing the rule, the algorithm adds a new child vertex for each successful pattern-matching step (i.e., whenever it finds a new candidate object for a node). Thus, a vertex has more than one child when there are different possibilities to match a node.

Figure 4.9 shows a part of the matching tree that is the result of a partial pattern-matching search of rule SystemElementToController (see Fig. 4.10) in the set of deleted elements from Fig. 4.8. As described, the root of this tree contains the matching of the context and the source produced domain pattern. The different bindings of this matching are shown in the form "*Node*:*NodeType* → <u>*Object*:*ObjectType*</u>", where *Node* represent a node from the TGG rule and *Object* is the matched object.



pm:Package → :Package, :Pack2Pack → :Pack2Pack,
pu:Package → :Package, se:SystemElement → vc:SystemElement
0
matching depth
weight
(i)
c:Component → pc:Component
1
Repair name constraint
w=0.5
(ii)
ct:Controller → :Controller
2
(iii)
step back to: c:Component
1
(iv)
cd:CodeDescriptor→:CodeDescriptor
2
(v)
cc:CodeContainer→:CodeContainer
3
(vi)
step back to: cd:CodeDescriptor
2
(vii)
step back to: c:Component
1
(viii)
:SE2Comp → :SE2Comp
2
Repair link to SystemElement
w=2
c:Component → vc:Component
1
ct:Controller → ?
Create new object for node ct:Controller
w=0.5
cd:CodeDescriptor → ?
Create new object for node cd:CodeDescriptor
w=0.5
cd:CodeContainer → ?
Create new object for node cc:CodeContainer
w=0.5
:SE2Comp → :SE2Comp
2

Figure 4.9: Matching tree resulting from searching the produced pattern of SystemElementToController in the set of deleted elements from Fig. 4.8

To be able to find every possible partial matching, the algorithm starts a search from every binding in the root. Let us assume it first tries to match

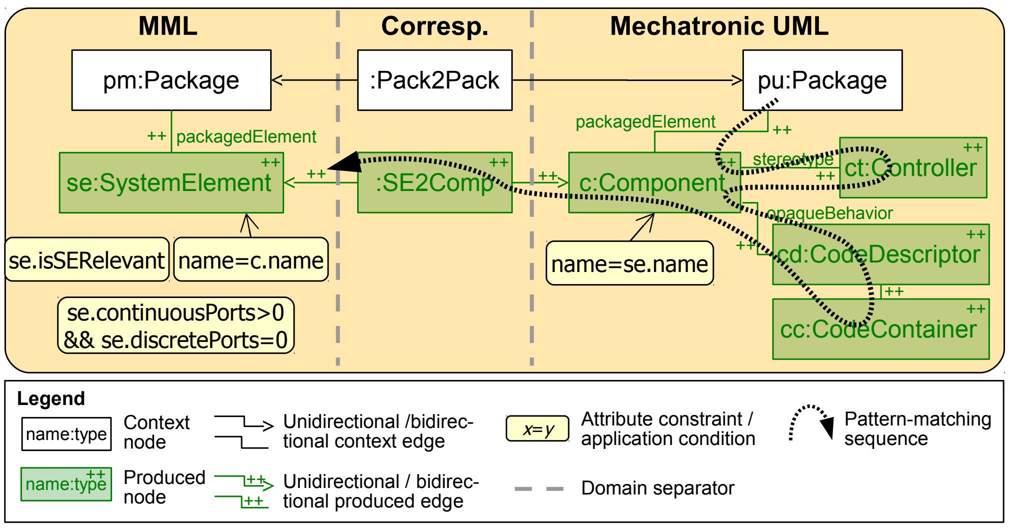


Figure 4.10: Sequence of pattern-matching steps for the left subtree of Fig. 4.9 within the TGG rule SystemElementToController

the TGG rule node pu:Package. It finds the not yet bound outgoing edge packagedElement to c:Component (see Fig. 4.10). The algorithm now has two options on how to match this node: Both the objects position controller and the velocity controller components match. So a new vertex is created for both (the left one is marked with (i) in Fig. 4.9), each with *depth* = 1 (denoted as the circled number in the upper right corner of the vertices in Fig. 4.9).

The left subtree of Fig. 4.9 contains the "wrong" partial matching possibility (marked with ✓s in Fig. 4.8). Figure 4.10 shows the matching sequence of this subtree. Here, the algorithm continues with matching the ct:Controller node to the controller object and adding a vertex with *depth* = 2 for it (ii). Then, as there is no unbound node connected to the ct:Controller node, the search must be continued at the previous node, decreasing the *depth* (iii). Here, the previous node is simply the node of the parent vertex, c:Component. Next, the cd:CodeDescriptor (iv) and the cc:CodeContainer (v) is matched. Again, as no unbound node connected to cc:CodeContainer exists, the algorithm steps back in the pattern matching, i.e., returns to the previous node, cd:CodeDescriptor (vi). There is also no unbound node connected to cd:CodeDescriptor. At this point, the previous node is not the node of the parent vertex. Therefore, the previous node is identified using the *depth* counter: we walk up the tree and select the first vertex v with v.depth $<$ $currentVertex$.depth, which is c:Component (vii).

The :SE2Comp correspondence node matches (viii), but its link to the position controller system element does not, because there is already a binding for the node se:SystemElement that binds a different object. This must be repaired if this partial matching should be applied, denoted with the ⊛. Furthermore, the attribute constraint that ensures the equality of the system element's and the component's name must be enforced by changing the name of the pc:Component (again marked with a ⊛ at the first vertex (i) of the left subtree).

The right subtree represents the other (more reasonable) partial matching from the previous example, where the ct:Controller, cd:CodeDescriptor, and ct:CodeContainer nodes could not be matched. Note that there are no real vertices for these unmatchable nodes in the tree. They are depicted dashed in Fig. 4.9 only to illustrate the repair operations needed to be performed to create a valid rule matching.

In fact, there is a third subtree (only adumbrated in the right of Fig. 4.9). The search starts at every node of the root's matching (remember it contains bindings for all context and produced source graph nodes). Thus, starting from the se:SystemElement node, the algorithm would create this third subtree which contains the same matching as the second subtree, just in opposite direction. To avoid presenting the same partial matching to the user twice, the algorithm checks whether there already is a vertex in the search tree that represents the very same matching. If this is the case, the algorithm does not add a new vertex, but instead creates an edge to the already existing vertex.[8]

Every vertex of the matching tree represents a partial matching and, at the same time, a possible repair operation. The number of reused elements is equal to the depth of a vertex in the tree (not the value of the *depth* counter), not counting the "step back to" vertices.

Listing 4.1 shows the pseudo-code of the partial pattern matching algorithm. The algorithm is an iterative depth-first search. It pushes each identified possible next matching step onto a stack. It starts a search for every binding in the context and produced source graph matching (lines 3–4). (The constructor "new Vertex(Vertex *parent*, Binding *binding*, int *depth*)" creates a new child vertex in *parent*, labeling it with *binding*.) In contrast to a regular pattern matching, it creates a new vertex for every node-binding possibility, as described above (line 10). When the pattern-matching algorithm does not find any new matching possibility (i.e., it has matched a complete subpattern), it searches the path to the root for the next vertex to continue the pattern matching (lines 11–15).

Once the tree is computed, we have to decide which of the several partial matchings (i.e., which vertex of the tree) should be used as a reuse possibility. We have discussed above that a reasonable heuristic is to select a partial matching that does not damage reusable correspondences and which reuses most elements, i.e., will require the least repair operations. In this way, it is likely that only previously related elements are reused, which is probably the intention of the user. This is also the rationale behind the high value of the "Correspondence Modification" metrics (cf. Tab. 4.1).

Using the default metrics, we have the following values for our two reuse possibilities. For the left subtree in Fig. 4.9, we have to modify an attribute (0.5) and to modify a correspondence link (2), resulting in an overall weight of 2.5. For the right subtree, we have to create three new objects, leading to an overall weight of $3 \cdot 0.5 = 1.5$. Using the default metrics, the right reuse possibility has a better (lower) weighting. With the default threshold of 0.2 for Good Quality, the algorithm will automatically select this possibility.

[8]In this case, the data structure is no tree any more, but becomes a directed acyclic graph (DAG).

Listing 4.1 Building the partial pattern matching tree

```
procedure buildPatternMatchingTree(Vertex $root$, DeletedElements $del$)
    Stack s;                                    ▷ Stack for non-recursive implementation of depth-first search
    for all Bindings $b \in root$.bindings do
        s.push(new Vertex($root$, $b$, 0));     ▷ Start a search from each context and produced source node
    end for
    while $s$ is not empty do
        Vertex $currentVertex \leftarrow s$.pop();        ▷ Get position from where the search tree is explored next
        Vertex $currentNode \leftarrow currentVertex$.binding.node;
        for all Nodes $n \in currentNode$.neighbor : $n \notin currentVertex$.boundNodes do
            for all Elements $elem \in del : elem$ matches $n \wedge elem \notin currentVertex$.boundElements do
                $s$.push(new Vertex($currentVertex$, ($n$,$elem$), $currentVertex$.depth + 1));
                                  ▷ Add new leaf to the search tree for every possible binding of neighboring nodes
            end for
        end for
        if no next Node $n$ or no next Element $elem$ was found then ▷ No further matching possible from current vertex?
            Vertex $v \leftarrow currentVertex$;
            while $v$.depth $\geq currentVertex$.depth do
                $v \leftarrow v$.parent;              ▷ Step back in the pattern matching by stepping back in the “recursion”
            end while
            $s$.push(new Vertex($currentVertex$, ($n$,$elem$), $v$.depth −1));
        end if
    end while
end procedure
```

4.1.8 Formal Properties of the Approach

As described in Sect. 2.4.2.2, there are two main formal properties of a TGG transformation algorithm: *correctness* (i.e., all produced model triples are words of the TGG) and *completeness* (i.e., for every valid source model, the algorithm will produce a valid model triple). It is important – especially when applying our improved model transformations techniques in a safety-critical context – that these formal properties are fulfilled. Otherwise, unexpected and/or defective transformation results could impair the safety of the system, eventually putting lives at risk if such errors remain undetected until the end of the development.

We argue that our improved incremental update algorithm fulfills these properties. The algorithm computes different possible alternatives to propagate a changes. One of these is then selected, either by the algorithm itself or by the user. At first glance, this may look like a contradiction to correctness, as there may be different results possible, even for TGGs that have functional behavior.

However, the core argument is that different ways in which we revoke and apply rules mainly affect the parts of the models *that are not subject to the transformation*, i.e., those parts that the TGG rules "do not speak of". Thus, a functional TGG still has functional behavior if we only consider the parts that are subject to the transformation. With the same argument, completeness is also not affected by our approach. We can employ a look-ahead for cases where a certain reuse possibility will negatively affect the applicability of certain rules.

Our algorithm only produces triples of models where the elements that are subject to the transformation form a valid triple model according to the TGG. That is because (a) when a rule is applied, reused objects and links will be modified so that they fit the rules, and (b) at the end of a incremental update run, unused objects that are marked for deletion will be actually destroyed. Therefore, after a successful run, every rule holds and no remainders of revoked rules exist. Thus, arguing informally, the fundamental TGG transformation properties correctness and completeness are unaffected by our new algorithm.

4.1.9 Summary

In this section, we presented a novel incremental update algorithm

- that is able to deal with cases of complex changes,
- whose user decisions are not based on atomic elements but on larger semantic units, and
- that is customizable in its amount of automation.

It generalizes the idea of structurally repairing rule applications such that complete structures can be repaired, independent of the kind of change. The algorithm tries to *apply rules in way that comes close to the state of the target model*, thereby retaining as much information of the target model as possible.

The algorithm may be run in full-automation mode, where it automatically chooses the "best" way to synchronize changes based on certain metrics. These metrics are a representation of expert knowledge, as they formalize reuse guidelines that have proven successful previously. It may also run in user-interaction mode, where it asks the user for alternatives whose metrics values are within a given threshold.

4.2 Abstraction and Concretion Relations

When translating between two models, the modeling languages of these models typically differ in their levels of abstraction and their purpose and viewpoint. As a result, not all changes to a model will affect the other model; the two models may still consistent in terms of the model transformation.

For instance, consider the transformation from the discipline-spanning system model to a discipline-specific model. The discipline-specific model contains more and detailed information from the discipline, whereas the discipline-spanning system model only deals with interdisciplinary information. That means that the discipline-spanning system model has a higher level of abstraction than the discipline-specific model. Such transformations are also called *vertical transformations*[9].

Thus, there exist several consistent concrete models for one abstract model. In other words, a consistency mapping from the discipline-spanning system model to the discipline-specific model is in fact a 1-to-n mapping.

As a consequence, if an engineer performs a change to their discipline-specific model, such a change may either be a *discipline-specific refinement* or a *discipline-spanning relevant change*. A discipline-specific refinement is a change to the discipline-specific model such that the consistency relation still holds. Thus, it must not be propagated to the discipline-spanning system model. In fact, most of the changes that occur during the development are discipline-specific refinements, for instance, the implementation of a software component or the specification of a controller. In contrast, a discipline-spanning relevant change places the discipline-specific model "outside" of the consistency relation. Therefore, such a change also affects the discipline-spanning system model (and possibly other disciplines' models) and must be propagated to the system model.

In the previous section, we used the notion of *information that is not subject to the transformation*. The distinction between refinements and relevant changes generalizes this concept. Changes that only affect information that is not subject to the transformation are refinements.

An illustration of such a consistency relation and how it is applied in a exemplary process is shown in Fig. 4.11. You see version 1.0 of the discipline-spanning system model in the upper left of that figure. From this version, we generate a first version 1.0 of the discipline-specific model (bottom left). Then, an engineer performs a refinement on that discipline-specific model, creating version 1.1. For instance, this change affects a model part that has no counterpart in the system model; hence, it is not subject to the consistency relation. Consequently, this version 1.1 of the discipline-specific model is still element of the consistency relation with version 1.0 of the discipline-spanning system model. That means no change propagation or consistency restoration has to be performed. Performing a relevant change, however, causes the consistency relation not to hold any more. Thus, the change to version 1.2 of the discipline-specific model must be propagated to the discipline-spanning system model in order to

[9]Horizontal transformations map between models of the same abstraction level, vertical transformations map between models of different abstraction levels [MG06].

restore the consistency, forming version 1.1 of the system model.

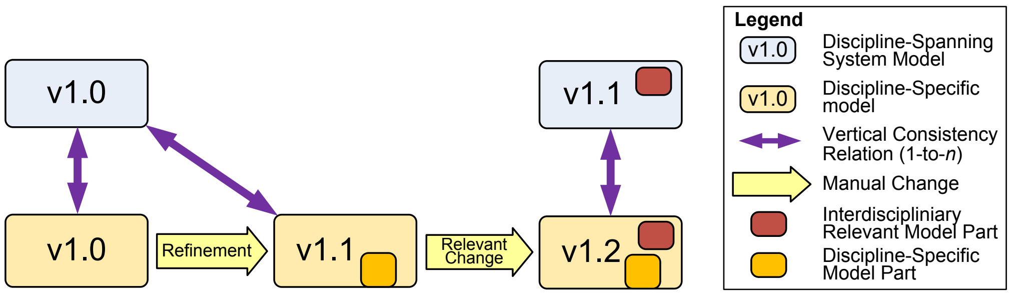

v1.0
v1.0
Refinement
v1.1
Relevant Change
v1.1
v1.2
Legend
v1.0 Discipline-Spanning System Model
v1.0 Discipline-Specific model
Vertical Consistency Relation (1-to-n)
Manual Change
Interdisciplinary Relevant Model Part
Discipline-Specific Model Part

Figure 4.11: Refinements and relevant changes in a discipline-specific model and its effects on the discipline-spanning system model

The problem, however, is that when using a 1-to-n mapping in a model transformation, the transformation becomes non-functional, i.e., there exist several consistent target models for a single source model. Control algorithms of existing model transformation approaches will either arbitrarily select one of the valid (i.e., consistent according to the consistency relation) possibilities (e.g, Giese and Hildebrandt [GH09]), or ask the user for every single decision (e.g., Körtgen [Kör09]). Obviously, the first is not feasible, especially when developing safety-critical systems where the transformation results must be deterministic and reproducible. The latter would massively increase manual tasks when transforming and synchronizing models, potentially outweighing the increased automation by using model transformation techniques.

More importantly, creating such a 1-to-n mapping is time-consuming and error-prone, as all possible refinements have to be modeled directly in the consistency relation manually. As most changes constitute refinements, the consistency relation will quickly get verbose. As existing transformation languages do not provide first-class support for modeling such non-functional relations, such a large 1-to-n consistency relation is also difficult to maintain.

To sum up the problem, defining such a mapping between two model of different abstraction levels requires a non-functional consistency relation. Existing model transformation approaches (e.g., [OMG08, GW09, HLR06]) do not provide sufficient support for that, because they mostly work for functional relations only. Even if the transformation language allows specifying non-functional mappings, it is a) difficult to develop and maintain, and b) not well supported by the synchronization algorithm, causing non-determinism or increased user interaction.

As a solution, we propose to model only a “normal”, functional 1-to-1 transformation mapping first. This transformation serves as the *default transformation* used to initially generate the (more concrete) target model. Next, we describe what changes to a discipline-specific model are considered refinements, i.e., changes that are not relevant for the discipline-specific system model. Finally, we combine the functional transformation mapping with the refinement definitions to automatically derive the 1-to-n consistency relation [RS12]. Figure 4.12 illustrates this concept.

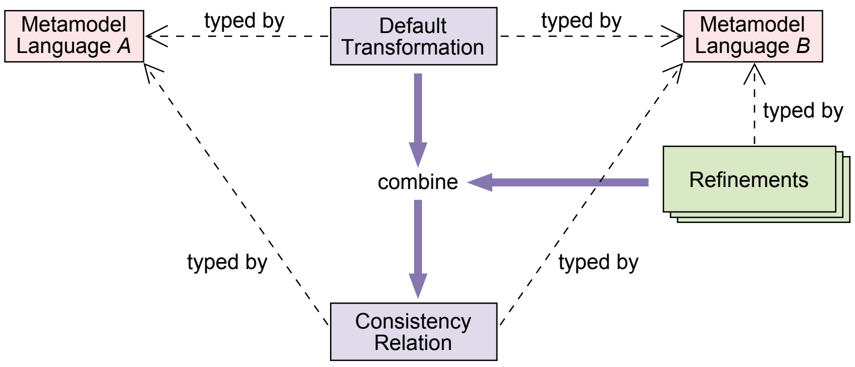


Figure 4.12: Concept overview

This follows the principle of *separation of concerns*: The initial, functional transformation describes the *general concept* of the mapping. All refinement operations are specified *independently* from this initial transformation. They are later integrated automatically into the consistency relation.

A particular advantage is that the discipline's engineers can define refinement rules using just constructs from their respective modeling language – they do not need to know details of the other modeling language in the transformation. They neither have to explicitly deal with graph transformations nor have to know details of the (initial) transformation. However, transformation experts can still access the underlying graph transformations, e.g., to fine-tune a certain mapping.

Furthermore, we can regard a refinement rule can as a special type of refactoring operation. We can use the rules to assist the discipline's engineers in finding alternative implementations. To do so, we simply have to check whether a refinement rule matches at a given model element. Applying this refinement rule will then refine the model.

When developing safety-critical systems, the semantic correctness of a transformation (i.e., that the semantics of the source model will always be equivalent to the semantics of the target model) is important. However, "showing [...] full semantics preservation of a set of model transformation rules [...] is a very difficult problem" [HKR+10], even if the semantics of both models are defined with similar means [GL12]. The problem significantly enlarges with large rule-sets, like a complete consistency relation R. Thus, our approach of separation may help proving semantics preservation. Transformation engineers only have to prove the correctness of the initial transformation I. Additionally, all refinements rules for the target model have to be true refinements, i.e., they will never affect the semantics of the source model. We think that this is easier to prove than proving semantics preservation for R, because the refinement proof can be (at least partially) automated. However, it remains to be investigated further whether this is really the case.

In the remainder of the section, we describe the proposed solution in detail. First, we formalize the problem in Sect. 4.2.2. The definition of the initial transformation function is described in Sect. 4.2.3. In Sect. 4.2.4, we present a technique for describing refinement operations. Based upon these, the overall consistency relation is derived, as shown in Sect. 4.2.5. Finally, we generalize the concept in Sect. 4.2.7 so that is applicable also for n-to-n relations.

4.2.1 Related Work

Most model transformation approaches only allow functional transformation specifications, either because a non-functional transformation will produce non-deterministic outputs, or because a non-functional specification will not work at all. The "location determination" and "rule selection" features of the Czarnecki-Helsen classification (cf. Sect. 2.2.2) describes this. Some approaches, however, also allow non-functional transformations, for instance, by letting the user decide when more than one rule is applicable.

4.2.1.1 Goldschmidt and Uhl (2008/2011)

As already described in Sect. 4.1.1, Goldschmidt and Uhl [GU08, GU11] use so-called *retainment policies* to deal with manual changes in the target model. Retainment policies specify "how a transformation rule should handle manual changes in target models" [GU08]. Policies specify possible retainment actions for types of changes, like additions or deletions of elements. Possible retainment actions are to overwrite manual changes, only update when there was no manual change, to always retain the manual change, or to never update the target. The policy's scope defines in which rules this policy should be applied.

The retainment rules of Goldschmidt and Uhl operate on the level of transformation rules. Thus, defining retainment rules requires knowledge of the ruleset. If, for instance, changing an attribute of a certain type of object is a valid refinement, we have to specify retainment rules for all transformation rules that can will affect this attribute. Furthermore, they can only define retainment policies for single syntactic elements. It is not possible to define more complex policies that are valid only for a certain graph structure or that contain further conditions. However, we also use the idea of separating the original transformation from the retainment rules ("refinement rules" in our case).

4.2.1.2 Körtgen (2009)

Körtgen [Kör09] developed a synchronization tool for the case of a simultaneous evolution of both models. Although it does not incorporate a concept to define refinement operations, it also allows having several conflicting rules, i.e., rules that are all applicable at the same position. All these rules are contained in the same ruleset, i.e., there is no separation of concerns (between the general transformation concept and possible refinements). In a step-by-step, highly interactive process, the user may decide which alternatives should be applied. Our aim is to avoid unnecessary user interaction where that is possible.

4.2.1.3 Klar et al. (2010), Lauder et al. (2012)

Klar et al. [KLKS10] and Lauder et al. [LAVS12b] and their tool eMoflon[10] support non-functional TGGs. Similar to Körtgen, there is no separation of concerns. A look-ahead of 1 is used to resolve local rule choices using a dangling edge check (i.e., they search for edges that can no longer be translated if a wrong rule is chosen). As eMoflon only supports a look-ahead of 1, the class of TGGs is limited to TGGs where DEC 1 is sufficient to resolve conflicts. If more than one rule is applicable eMoflon asks a component (which can be the user, a configuration file, or an algorithm) to decide [HLG+13]. However, they do not give further information on possible algorithms for this.

4.2.2 Problem Formalization

In a conceptual view, we have an abstract language A and a concrete language B. In our example scenario, A is the system model's metamodel, and B is the discipline-specific metamodel.[11]

To transform a word $a \in A$ to a word $b \in B$, we use an initial transformation function $I \subseteq (A \rightarrow B)$, as shown in Fig. 4.13. However, as B is more concrete than A, a consistency relation R contains more elements than I and is not a function: $I \subseteq R \subseteq (A \times B)$.

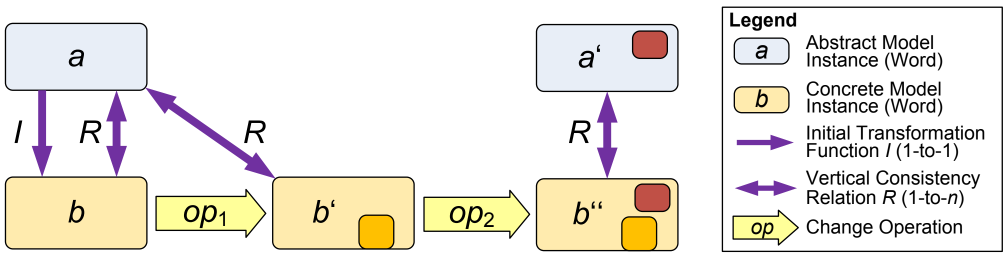

a
I
R
R
b
op1
b'
op2
a'
R
b''
Legend
a
Abstract Model Instance (Word)
b
Concrete Model Instance (Word)
Initial Transformation Function I (1-to-1)
Vertical Consistency Relation R (1-to-n)
op
Change Operation

Figure 4.13: Formalization of vertical model transformations

Paige et al. [PKP05] provide a first, simple formalization of a correct refinement: A model A is refined by a model B iff A and B are well-formed and internally consistent, and A and B "obey any cross-model consistency rules relevant to their context" [PKP05]. In our formalization, these "cross-model consistency rules relevant to their context" take the form of the final 1-to-n consistency relation.

Given a change operation op on the model b, i.e., $op \in (B \rightarrow B)$. Then, op is a consistency-preserving *refinement* iff $\forall a \in A, b \in B : (a, b) \in R \Rightarrow (a, op(b)) \in R$, i.e., both the concrete model before and after the operation map to the same abstract model [RS12]. op_1 in Fig. 4.13 is such a refinement.

[10]eMoflon Website: `http://www.emoflon.org`

[11]Note that we use "language" and "metamodel" (as well as "word" and "model") interchangeably here. In general, a *language* is considered to be a set of derivation rules that define allowed words of the language in a constructive way, i.e., by consecutively applying those rules to a start word. In contrast, a *metamodel* defines constraints that must hold on a word if it is an element of the language. For a more formal comparison of both concepts, see Amelunxen and Schürr [AS08].

In contrast, op_2 is a discipline-spanning *relevant change*. Model instance a must be changed (and becomes a') to make the consistency relation R hold again.

A refinement op can be a single, exactly defined manual change operation. However, the idea is to *generalize the description of refinements*: In this way, it becomes a universal change definition that is applicable to more than one model instance, similar to a refactoring operation.

4.2.3 Definition of the Initial Transformation Function I

The first step of defining a 1-to-n relation is to define a functional, initial transformation. This is a traditional transformation definition. It is required that this transformation is indeed functional.

As example, let as consider a synchronization of state chart behavioral models, as introduced in Sect. 3.2.3 (cf. Fig. 3.7). When transforming a CONSENS statechart to a STATEFLOW chart, the CONSENS states are mapped directly to STATEFLOW states, and logical relations in combination with its events are mapped to transitions. Figure 4.14 shows the rule for the state mapping. It further ensures that the names of the states are equal in both models.

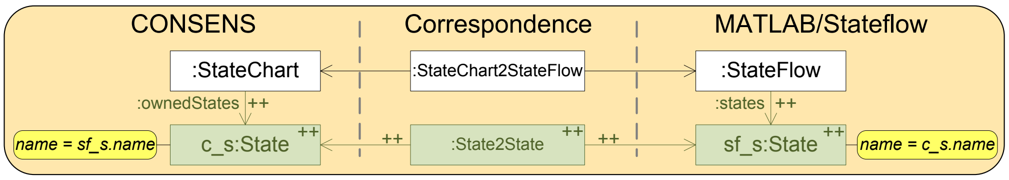


Figure 4.14: TGG Rule CONSENS State to MATLAB/Stateflow State

We defined a set of TGG rules to transform between CONSENS and MATLAB/STATEFLOW. For instance, rule Transition to Transition (Fig. 4.15) describes how transitions between states in CONSENS map to transitions in STATEFLOW. The maximum duration of a transition is represented by an annotation in STATEFLOW.

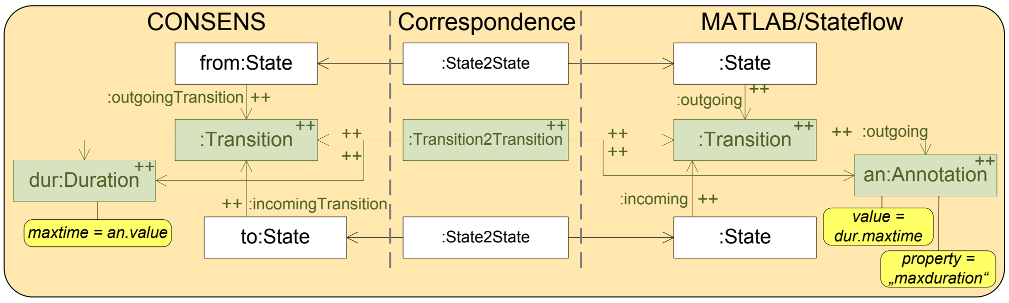


Figure 4.15: TGG Rule Transition to Transition

4.2.4 Definition of Refinement Operations

Let us consider the example process from Fig. 3.2 again. Figure 4.16 shows how the actual models evolve during this process. We use this process to give an example of a refinement and explain how to define the respective refinement rules.

After the discipline-specific models have been generated from the system model (step 1), the control engineers implement the controllers using MATLAB/Simulink and Stateflow (step 2). Especially, they have to define how the RailCab switches between controller configuration when it enters or leaves a convoy. To do so, they modify the Stateflow model by incorporating additional states that describe the fading behavior when switching between the configurations Figure 4.16 shows this modification in the first and second row of the right column “MATLAB/Stateflow” (step 2).

Such a change is considered a discipline-specific refinement, as it does not affect other disciplines. Therefore, it must be propagated neither to the discipline-spanning system model nor to the other disciplines. When using traditional model synchronization techniques, these additional states would be nevertheless propagated back to the system model: When synchronizing the models, the TGG Rule State to State (see Fig. 4.14) is applicable for the new intermediate states. The TGG rule Transition to Transition is also applicable for the new transitions. Thus, these new states and transitions in the target model, which should be discipline-specific refinements, will cause a creation of corresponding states and transitions in the source model. Consequently, we have to find a way to tag certain changes as refinements, so that they are automatically ignored by the synchronization.

Dealing with hierarchical refinements (like adding sub-states or subcomponents) can be achieved by simply ignoring everything “below” an existing element. Rieke [Rie08] and Gausemeier et al. [GSG⁺09] show how this can be achieved using *relevance annotations*. Relevance annotations mark elements as *subject to a transformation*; we have already used this notion in Sect. 4.1. However, for complex, non-hierarchical refinements as described above, this is not sufficient: The engine cannot simply ignore this change, as it still corresponds to constructs in the system model and, therefore, may be affected by future changes. Thus, we need other means to specify refinements.

We propose that discipline experts define a set of rules that describe which kinds of changes to a discipline-specific model are regarded as discipline-specific refinements. We call these *refinement rules*. A refinement rule is a model transformation rule that formally describes a particular type of refinement. Its source and target model is the same model, i.e., they perform an in-place transformation of an existing (target) model according to the Czarnecki-Helsen classification (cf. Sect. 2.2.2). We further propose to use a graph-based transformation approach, because we would like to combine it with our TGG-based synchronization, which is also graph-based.

Such a rule defines a refinement by a precondition (left-hand side) and a replacement (right-hand side). The precondition describes a situation that can be refined, and the replacement defines the actual refinement that replaces the

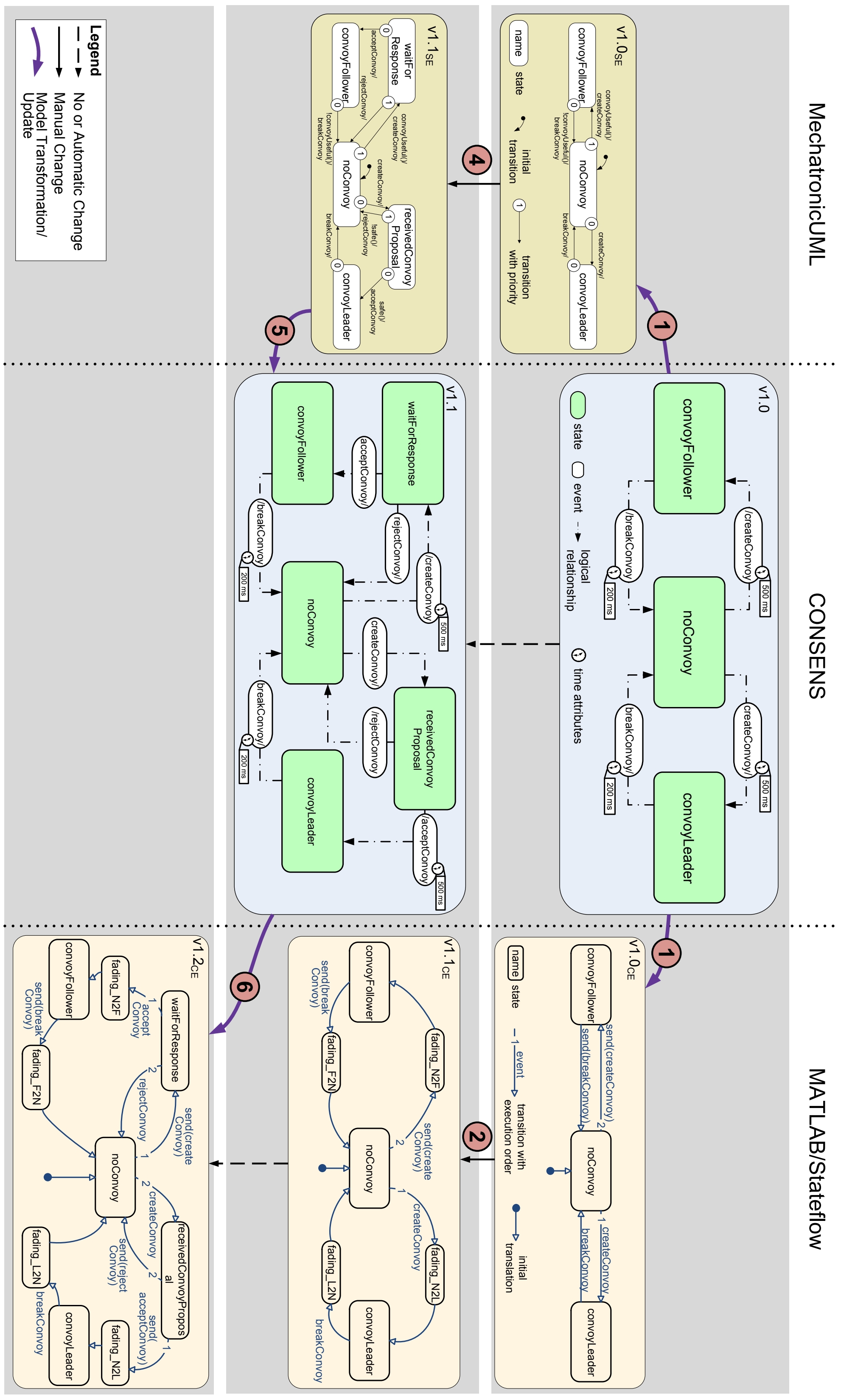


Figure 4.16: Evolution of the different models during the development process

precondition. From a viewpoint of the system model, the refinement must not change the semantics. This means that when we apply the refinement rule, the discipline-specific model still corresponds to the (unchanged) system model.

Fig. 4.17 shows a refinement rule using the concrete syntax of the modeling language Stateflow. The rule defines that adding an intermediate state is a discipline-specific refinement. It describes that a transition may be replaced by a combination of a transition, a state and another transition. In addition, a constraint ensures that the new state and transitions must not violate the maximum duration of the original transition. That means that a change is only a refinement according to this rule if the overall duration of the transition from `state1` to `state2` does not increase. Furthermore, no other incoming or outgoing transitions are allowed for the intermediate state: Otherwise, it would be possible that the new statechart allows more behavior than before – which would obviously be a relevant change.

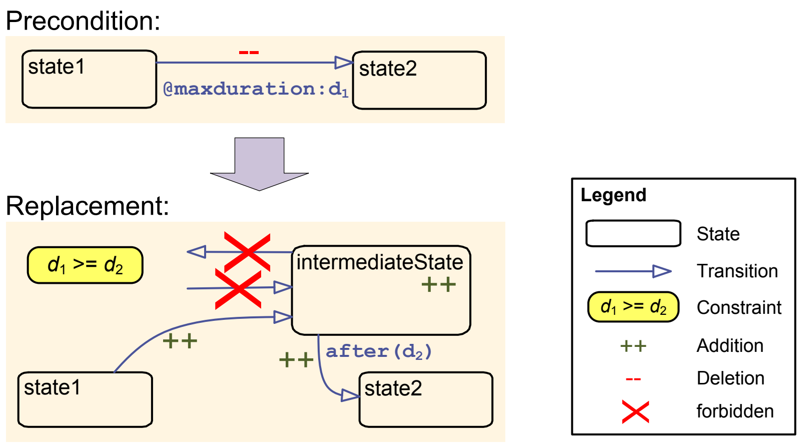


Figure 4.17: Refinement rule (concrete syntax) for adding intermediate states in the Stateflow control engineering model

This refinement rule covers the addition of the fading states (described above). Using this rule, we can identify that such an addition is a refinement. A model synchronization algorithm can use the set of refinement rules to automatically detect whether a change is a refinement.

Figure 4.18 shows the refinement rule of Fig. 4.17 in short-hand graph transformation rule notation. When choosing the language to define refinements, we sought to cover as many refinements as possible on the one hand and, on the other hand, not making the language too complex to make analyses impractical. We identified several refinements from different disciplines (e.g., fault-tolerance patterns like triple modular redundancy, functional partitioning of components, load balancing) which can be described in terms of such graph transformation rules. However, it remains to be investigated further whether we may need a more sophisticated language for other refinements that we have not identified, yet.

Our goal was that these refinement rules should be integrated into the consistency relation R, so that the model transformation engine itself can deal with

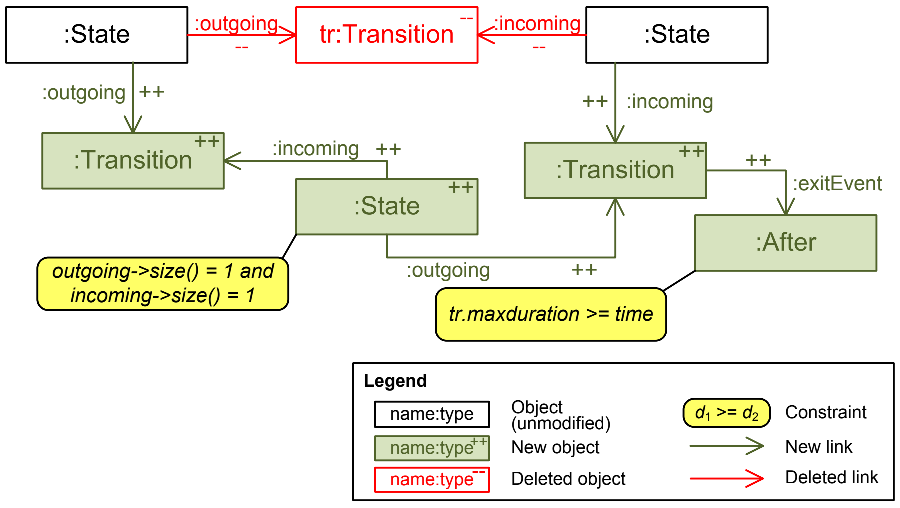


Figure 4.18: Refinement rule from Fig. 4.17 in abstract syntax

refinements without fundamental changes to the synchronization algorithm. In this way, formal properties of TGGs like correctness or completeness are still valid and we do not have to heavily modify the existing synchronization tool. We therefore add the information from the refinement rules to the TGG ruleset that defined the initial transformation I, creating an altered TGG ruleset for the consistency relation R.

One particular advantage of this approach is the *separation of concerns*: We separate the refinement definition from the transformation specification. When defining a refinement rule, the engineers do not have to think about which parts of the mapping (i.e., TGG rules) might be affected. Instead, they focus on defining the actual refinement rules, which are later automatically integrated in the model synchronization. This also helps maintaining the set of refinements, as later changes to a refinement rule are consistently and automatically applied to all affected TGG rules – the transformation engineers do not have to deal with consistently editing all affected rules manually.

4.2.5 Derivation of the Consistency Relation R

The basic idea is to check where refinement rules match in the original TGG rules in the target domain. Whenever we find a refinement rule's precondition in a TGG rule, we create a copy of that TGG rule and apply the refinement rule in this TGG rule copy. In this way, we derive new TGG rules which map the same source pattern to the refined target pattern.

Consider the refinement rule from Fig. 4.17/4.18. This refinement rule's precondition (left-hand side) matches in the target domain of the TGG rule Transition to Transition (Fig. 4.15). We now copy that TGG rule and apply the refinement rule onto its target domain. That means that we delete every TGG node and edge that match deleted elements in the refinement rule, and create new nodes and edges for everything that is created by the refinement rule.

Furthermore, we create constraints in the new TGG rule for constraints in the refinement rules. Fig. 4.19 shows the resulting refined TGG rule. This new TGG rule now matches whenever a refinement according to the refinement rule took place in the target model. This rule matches at the respective refined model elements; thus, the models are consistent in terms of the new synchronization ruleset.

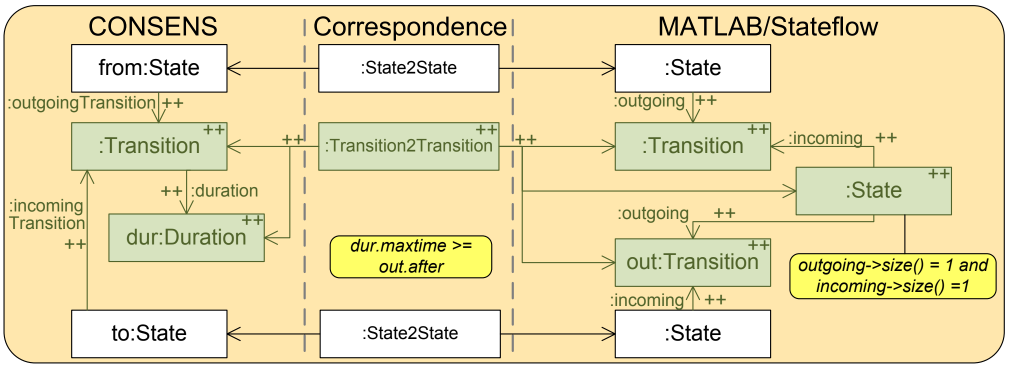


Figure 4.19: TGG Rule Transition to Transition (refined, with intermediate state)

This new rule becomes part of the overall consistency relation R. Next, we describe how the improved synchronization applies this relation R and deals with subsequent incremental updates that may affect refinements.

4.2.6 Model Synchronization with a 1-to-n Consistency Relation

Let us have a look at how the improved model synchronization algorithm deals with refinements that engineers introduced. First, we discuss how the synchronization detects that a refinement operation was applied in the target model. Next, we show how this helps avoiding information loss when the synchronization has to update refined target model parts due to source model changes.

4.2.6.1 Detecting the Refinement During Backward Update

Control engineers added fading states to the STATEFLOW model (step 2 in Fig. 4.16). As the model has changed, we run a (backward) synchronization to propagate the change to the system model, and from there to all other affected disciplines.

As described in Sect. 2.4.3.1, traditional model synchronization algorithms work in two steps: First, for everything that has been deleted (or inconsistently changed) in one model, the corresponding elements in the other model are also deleted. Second, for everything that has been added (or inconsistently changed), new corresponding elements are created. In this case, the control engineer deleted the transition from the noConvoy to the convoyLeader state and added a new fading state fading_N2L and two transitions. Thus, the synchronization would also delete the corresponding transition from the system model and then add new elements, which is what we want to avoid.

In the previous section, we proposed an novel model synchronization approach. Its main idea is not to delete such corresponding parts right away, but to mark them for deletion first, so they can be reused later, i.e., in subsequent rule applications. When creating elements during rule application, our synchronization performs a search in the set of elements marked for deletion and tries to reuse fitting elements; if they fit, they are not deleted. Only if no fitting elements that are marked for deletion can be found, new elements are created. Finally, elements marked for deletion that could not be reused are actually destroyed. For details of the improved synchronization algorithm, please refer to Sect. 4.1.

We reuse this novel synchronization here to deal with the refinements. In this example, this improved algorithm works as follows. First, as the transition from the noConvoy to the convoyLeader state has been deleted from the Stateflow model, the corresponding transition in the system model is marked for deletion. Next, we try to apply new rules. Here, the new, refined TGG rule (Fig. 4.19) is applicable: It matches the transition in the system model that has been marked for deletion, and it also matches the new fading state and the new transitions in the Stateflow model. We can now apply this rule: In the CONSENS model, we reuse the transition marked for deletion, and we bind the elements of the refinement in the Stateflow model. As result, we have applied the refined TGG rule in backward direction without performing any changes to the CONSENS model, just by reusing elements marked for deletion.

The models are now consistent in terms of the TGG. This is what we wanted to achieve: We have derived a new TGG rule that covers the refinement case described by the refinement rule. Furthermore, when later changes make the refinement invalid, e.g., when the time constraint is violated, the model transformation engine can detect this by checking the validity of the application of the refined TGG rule.

Note that we have to add precedences to the TGG rules. When propagating changes to the abstract model, we want to use these refined rules primarily. Applying the original, non-refined TGG rules would propagate the refinement to the abstract model, which we wanted to avoid.

The consistency relation R is non-functional, because it also contains rules for all refinements. Each of these additional rules constitutes a rule application conflict with its ancestor rule, because they have the same produced source part. When propagating to the concrete model (forward transformation), we have to exclude the new rules we need a functional, deterministic transformation. Thus, we only use the initial rule I set.[12]

4.2.6.2 Propagating Changes to Refined Model Parts

Let us have a look at the next steps in the development process. As explained before, the software engineers work on their model, too. They change the behavior of the software by adding the possibility to reject a convoy proposal (step 4 in Fig. 4.16). This is a discipline-spanning relevant change, as it also affects,

[12]We could also let the user decide in such rule conflicts, and use backtracking/look-ahead to show the potential outcome (see Sect. 6.1.4 for details).

for instance, the controller implementation in the control engineering. Thus, it is propagated back to the system model (step 5): Equivalently to the software model, the state diagram is extended by two states waitForResponse and receivedConvoyProposal and new transitions and messages (see v1.1 of the system model in Fig. 4.16). Instead of switching to the state convoyFollower directly after a createConvoy message is send, the follower RailCab switches to the new state waitForResponse. There, it waits for the leader RailCab to accept or to reject the convoy proposal. The leader RailCab receives the createConvoy message and changes to the new state receivedConvoyProposal, in which it decides whether it accepts or rejects the proposal. If the convoy proposal is accepted, the leader RailCab changes its state to convoyLeader and the follower RailCab changes to the state convoyFollower. If the proposal is rejected, both RailCabs return to the state noConvoy.

These changes then must be propagated to other affected disciplines. Thus, the control engineering model also has to be updated to reflect the changed communication behavior (step 6). In the example, the createConvoy/ transition was changed in the system model during the incremental update in step 5. To transform this change, a naïve synchronization would first revoke the respective rule application by deleting the corresponding elements in the Stateflow model, and then try to retransform the affected elements. However, this createConvoy/ transition in the system model corresponds to a refinement introduced in v1.1_{CE} (the combination of the transition createConvoy, the state fading_N2L and the transition to the convoyLeader state in the control engineering model, which are bound by the refined TGG rule). Revoking the rule would destroy the complete refinement (see Fig. 4.20 b)).

As such an information loss must be prevented, we again use our improved synchronization. First, we revoke rules by marking for deletion. For instance, the fading_N2L state and its incoming and outgoing transition are marked for deletion due to the revocation of the refined TGG rule Transition to Transition (Fig. 4.19).

Next, we transform the new elements in the system model to the control engineering model by applying new rules, and we try to reuse elements that have been marked for deletion by the rule revocations. In general, there may be several possibilities to reuse elements previously marked for deletion, which lead to differently updated models; all of them are consistent according to the consistency relation. We already discussed that in Sect. 4.1.5. In our example, the question is where the newly added states waitForResponse and receivedConvoyRequest should be added: before (Fig. 4.20 c)) or after the fading states (Fig. 4.20 d))? Of course, an expert can recognize that d) is the correct way of updating, as the controller strategy must not be switched before every RailCab has actually approved the formation of a convoy. An automatic synchronization, however, cannot easily decide this.

As described in Sect. 4.1, our improved synchronization algorithm explicitly computes all reuse possibilities, rates them with respect to information loss, and asks the user in ambiguous cases which of the update possibilities is the correct one. In the example, the refinement in the control engineering model that has been marked for deletion (consisting of the transition createConvoy, the state

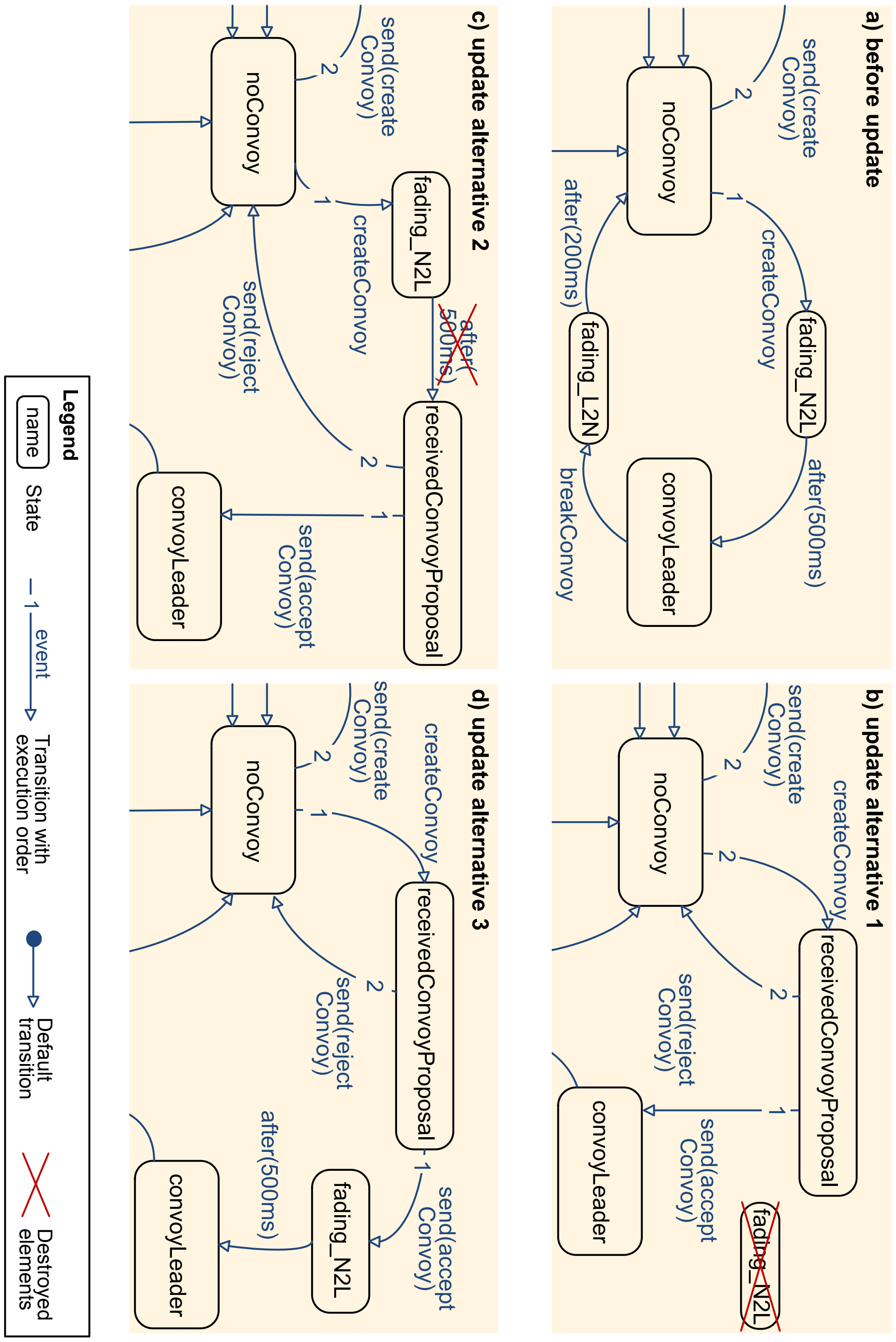


Figure 4.20: Excerpts from STATEFLOW model: a) before updating; updated in different ways: b) lost fading state, c) “wrong” propagation of the change, d) correctly updated

fading_N2L and the transition to the convoyLeader state) may be reusable as the corresponding control engineering part for three new transitions in the system model v1.1 (createConvoy/, /rejectConvoy, and /acceptConvoy). However, the deleted refinement is not reusable as is. Some additional modifications have to be made to make it reusable in the different cases. For instance, when reusing elements marked for deletion as corresponding part for the new transition createConvoy/ (which would result in Fig. 4.20 c)), the target of the outgoing transition must be modified to point to the state receivedConvoyProposal. Furthermore, we have to remove the after(500ms) exit event, as the createConvoy/ transition does not have a duration.

We can rate the quality of the different update possibilities using the metrics described in Sect. 4.1.6. The basic idea of the metrics is that the less modifications must be made, the more likely is that this is a reasonable reuse possibility. In the example, we can reuse the refinement for the transition createConvoy/ (see Fig. 4.20 c)), as the source of the transition (the noConvoy state) is the same as before, but we must alter the target state and remove after(500ms). We can also reuse the refinement for the transition /acceptConvoy (see Fig. 4.20 d)), as the target of the transition is the same (the convoyLeader state). It is, however, unreasonable to reuse the refinement for the transition /rejectConvoy (not shown in Fig. 4.20), as neither the source nor the target state is the same as before. Using the default metrics, the reuse possibilities that are depicted in Fig. 4.20 are weighted in the order b)–c)–d), with d) as the "best". Using the default values for our thresholds for manual decisions (cf. Tab. 4.2), the user has to decide between two reasonable reuse possibilities that are depicted in Fig. 4.20 c) and d), because they are similarly weighted. Alternative b) and other alternatives not shown in the figure are assigned a lower quality by the metrics, because they require more modifications to make the rule applicable. Therefore, those are not presented to the user.

4.2.7 Generalization to n-to-n Consistency Relations

The concept described in this section works only if there is a unidirectional abstraction level change between the source and the target model, i.e., the source model more (or at least equally) abstract than the target model in all of its aspects. However, in practice, we would also like to synchronize models of different viewpoints. This means that the first model can be more concrete in its core viewpoint than the other model, but it may be more abstract in the viewpoint of the second model. Hence, a 1-to-n consistency relation is not sufficient; we need a n-to-n relation to define the mapping between the two models.

Typically, we find this type of relations when directly mapping between two discipline-specific models. In our example of mechatronic system development, the overall system simulation is performed using an MATLAB/SIMULINK/STATEFLOW model. Besides the controller implementation (which are implemented using Simulink anyway), this model also includes the implementation of the discrete behavior and the message-based communication implemented in MECHATRONICUML. Both the Simulink and MECHATRONicUML models contain information that is not relevant to the other model. For instance, MechatronicUML features sophisticated model checking support including the specification of safety properties. These safety properties are only important during the development of the software: The model checker uses them to prove the safety of the system. Thus, they do not have a direct influence on the implementation system's behavior.[13] In turn, the details of a controller implementation are irrelevant to the MechatronicUML model. Each model has parts of higher *and* lower abstraction levels than the other.

However, this is not restricted to direct mappings between discipline-specific models. The MechatronicUML software model takes a logical viewpoint on the system, whereas the active structure of the system model focuses more on hardware-related aspects. Refinements in the software models are therefore typically due to functional aspects. Possible refinements in the system model often relate to the necessary hardware.

Figure 4.21 shows an abstract view on such a n-to-n consistency relations and how it maps between two models. The general idea is not only to provide an initial transformation function I_f for forward direction, but also to specify a backward transformation function I_b. TGGs are a bidirectional transformation language. Thus, we specify both of these functions within one TGG that has functional behavior for both transformation directions (cf. Sect. 2.4.2.2).

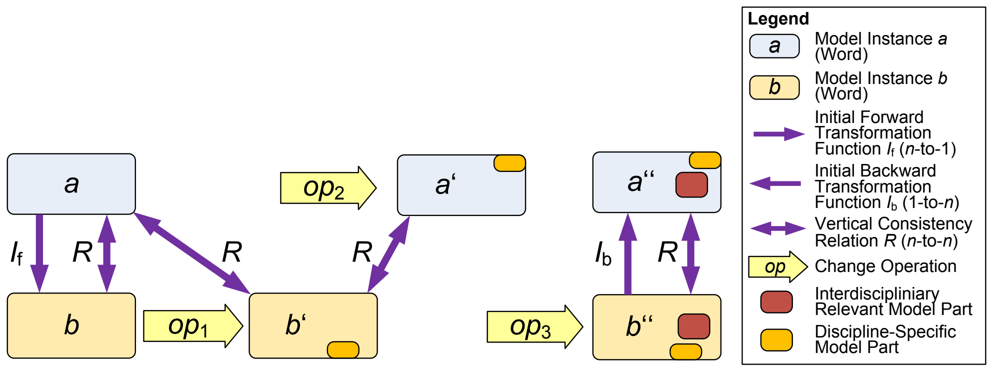

a
b
If
R
op1
b'
R
op2
a'
R
a''
Ib
R
op3
b''
Legend
Model Instance a (Word)
Model Instance b (Word)
Initial Forward Transformation Function If (n-to-1)
Initial Backward Transformation Function Ib (1-to-n)
Vertical Consistency Relation R (n-to-n)
op Change Operation
Interdisciplinary Relevant Model Part
Discipline-Specific Model Part

Figure 4.21: General formalization of n-to-n consistency relations

Similar to the 1-to-n approach, we then use these transformation functions to derive the n-to-n consistency relation. We do this by using the refinement rules for the target model to generate the target ruleset extension. Then we do the same for the source refinement rules: We apply them to the source parts of the TGG rules, leaving the target parts unaffected. This works as long as there is no source model refinement that affects a rule that is already refined by a target model refinement (and vice versa).

[13]The state-based behavior is specified in MechatronicUML and then model-checked using the safety properties. When the model checker detects a violation of a safety property, the software engineer fixes this violation by modifying the state-based behavior. Once there are no further violations, the MechatronicUML model is synchronized with the MATLAB/Simulink/Stateflow model. This allows and integrated simulation and efficient code generation for target platforms.

However, it may also happen that both source and target refinement rules affect *the same TGG rule*. As an example, again consider the mapping from the active structure to a software component model. A software component may be refined in several ways. For instance, we may perform a *functional split*, i.e., a component is split up because it fulfills two different functions. Such a change corresponds to op_1 in Fig. 4.21. In the system model, a system element may be refined such that we apply the *triple modular redundancy* pattern. We create three instances of the same system element, so that each of them computes a result independently. This results in a higher fault tolerance. Such a change typically is not relevant to the software model, as it does neither influence the interfaces of the corresponding component nor the component behavior. This change corresponds to op_2 in Fig. 4.21.

Both refinements affect the same rule SystemElement2Component. Simply creating one refined TGG rule for both source and target side, however, is insufficient: None of the rules would cover the case where both the system element *and* the software component are refined. Therefore, we also create a fourth rule that covers this case.

The case described above also shows the advantage of automatically deriving these refined rules: The number of rules required in such a case is $n \times m$. With a larger number of possible refinements on both source and target side, defining and maintaining all the refined rules will quickly become infeasible.

4.2.8 Summary

In most development processes, we find models on different abstraction levels. These models have to be kept synchronized. Due to their differing levels of abstraction, there are 1-to-n consistency relations between the different models. Existing model transformation techniques focus on functional transformation and do not provide first-class support for such non-functional relations.

In this section, we have presented a novel way to define such 1-to-n mappings. We use a traditional 1-to-1 functional mapping I. Possible refinements on the models are defined separately. These refinement rules describe what changes to a model are model-specific, i.e., do not have an influence on the other model. Together, the mapping I and the refinement rules form the 1-to-n consistency relation R.

This approach helps coping with the complexity of a non-functional mapping by separating the definition of the “core” mapping (which specifies the general concept) from the definition of refinements. Refinements can even be specified by the discipline’s engineers. This allows the systems engineers to concentrate on the basic principles of the mapping. This follows the principle of *separation of concerns*. Finally, we generalized this approach to n-to-n consistency relations.

4.3 Synchronizing Concurrent Modifications

In a distributed development environment, several engineers from different disciplines work on their models independently. As it is well-known in distributed software development with source code, this can lead to *editing conflicts*[14]: One developer changes a certain artifact (e.g., an element of a development model, a piece of code, or the text of the requirements document), and at the same time another developer changes this artifact, too. If both changes contradict each other, this is a conflict. When working with a system of interrelated models that are all modified by different developers, the conflict could also be due to a change in another model.

Existing approaches for model versioning, conflict detection and resolution often remain on a purely syntactic or text-based level, i.e., they simply use a text-based representation (e.g., XMI) of the models to perform the merging. This obviously leads to an inferior merging and may even result in errors [Rie08, BLS⁺12].

There exist some automated merging tools for models, e.g., EMF Compare [Ecl13]. Furthermore, some version control systems allow dealing with the semantics of models, like SMoVer [ASK10] and EMFStore[15] for EMF-based models, or OdysseyVCS [MCPW08] for UML models. For a more detailed overview of existing of model comparison, merging and version approaches, see, for instance, Altmanninger et al. [ASW09], Brosch et al. [BLS⁺12], or Stephan and Cordy [SC13].

Such model differencing and merging implementations do not always provide reliable results [LAS⁺14]. Graph matching is an NP-hard problem, and such methods have to rely on heuristics. Especially when dealing with large models and change sets, the internal matching heuristics cannot reliably determine corresponding model elements; this leads to worse results for larger models with more than a few changes.

Moreover, these tools do not deal with sets of interrelated models, where changes in one model may lead to changes (and, thus, potential editing conflicts) in several other models. However, such situations occur frequently in model-driven engineering. In our scenario, engineers from different disciplines edit their models simultaneously, which may lead to editing conflicts with models from other disciplines.

In this section, we present an approach how to perform a *simultaneous, bidirectional model synchronization*, that propagates changes from both models to the respective opposite model and allows resolving multi-model editing conflicts. It integrates existing model merging techniques with our model transformation and synchronization technique based on TGGs. The engineers are thereby relieved from much manual, error-prone work during the model synchronization

[14]In the context of model transformations, the term *conflict* is also used to describe a situation where more than one model transformation rule is applicable during a transformation run. When speaking about a “conflict” in this thesis and especially in this section, we mean *editing conflicts* (a conflict caused by two users concurrently editing the model). Otherwise, we explicitly use the term “rule (application) conflict”.

[15]EMFStore website: `http://eclipse.org/emfstore/`

and conflict resolution task, as the approach solves many conflicts automatically. It aims at improving the model-merging reliability, as the model merger has to deal with fewer changes. It also provides sophisticated means to ease the resolution of editing conflicts that cannot be solved automatically.

This section is structured as follows. We present a categorization of conflicts in Sect. 4.3.1. After giving an overview about the related work in Sect. 4.3.2, we describe the general concept of our approach in Sect. 4.3.3. We further extend this approach in Sect. 4.3.4 to improve the merging precision.

4.3.1 Conflict Categorization

Next, we give an overview about the dimensions of editing conflicts, different types of editing conflicts (based on the performed editing operations that lead to that conflict), and possible resolution strategies.

4.3.1.1 Dimensions of Editing Conflict in Multi-Disciplinary, Multi-User Development Processes

When working in a team of developers on a single model, conflicts may arise due to developers changing a model simultaneously. Here, we distinguish between three categories of conflicts:

- *syntactic conflicts*, i.e., changing the same model element (e.g., both developers modify a property of that element to differing values)
- *static-semantics conflicts*, i.e., changing different model elements that are related (e.g., both developers add a state with the same name to a state machine, but the metamodel requires unique state names)
- *dynamic-semantics conflicts*, i.e., changing different model elements that influence each other at runtime and thereby cause a faulty behavior of the system

Syntactic conflicts constitute already on syntax level and are therefore easy to recognize and to resolve using diff/merge tools (in many cases even text-based, e.g., using a model's XMI serialization). *Static-semantics conflicts* are more difficult to identify, as they do not constitute as a conflict in a persistence-level-based merging (e.g., a text-file-based merger). They require analyzing the static semantics of the model to be recognized (and solved). Conflicts may also arise in the *dynamic semantics*. For instance, an engineer performs a modification that increases the total weight of the system under development. Next, he or she performs a simulation to ensure that the controllers are still working correctly with the increased weight. If another engineer changes the controller strategies in the meantime, this new strategy may not work with the increased weight. Such a conflict can only be identified with the help of manual or automated conformance tests.

Furthermore, two simultaneous changes can constitute a conflict on syntactic or static-semantics level, but if we consider the dynamic semantics, the conflict vanishes. An example for such a *pseudo conflict* is when two developers model the same behavior in two different ways. In such a case, the conflict can be automatically resolved by arbitrarily selecting one change. For a more detailed discussion on the categories of conflicts, we refer to Altmanniger and Pierantonio [AP11].

Model transformations operate on syntactic and static-semantics level. In this thesis, we therefore focus in syntactic and static-semantics conflicts. A detailed discussion on dynamic-semantics conflicts is outside the scope of this thesis.

Model differencing and merging still is a difficult problem in practice [ELHN+10, LAS+14]. Using a simple revision control system (like Apache Subversion[16] or Git[17]) cannot solve the problem, as such a system does not have any knowledge about the inter-model dependencies[18].

The problem even enlarges when working with interconnected models. For instance, in our scenario of mechatronic system development, the conflict could be due to a change in another discipline's model. This adds a new dimension to the problem.

Direct, single-model conflicts are easy to solve using model differencing and merging tools. If the resolution of the conflict requires user interaction, typically all involved developers can perform this resolution.

In the case of *multi-model conflicts*, different developers may change different models that are technically unrelated (i.e., in terms of persistence in a file system or database), but connected with a model transformation/mapping. Conflict resolution for multi-model conflicts requires performing a model synchronization and a (potentially manual) conflict resolution using model differencing and merging tools. As several disciplines are involved in such conflicts, ideally a system engineer performs the conflict resolution in consultation with the discipline engineers. As the conflict affects several models, its resolution has to be consistently applied to all models.

Fig. 4.22 summarizes the two dimensions of editing conflicts and the necessary resolution approaches. For instance, while syntactic single-model conflicts may be resolved using simple text-based differencing and merging techniques, the resolution of static-semantics, multi-model conflicts requires static-semantics aware merging in combination with model synchronization techniques.

4.3.1.2 Editing Conflict Types

When two developers modify the same model, conflicts can be classified by the editing operations that caused the conflict. The type of conflict also determines the conflict resolution strategy, i.e., whether and how it is possible to solve the conflict automatically.

Most of the conflicts that may arise in situation where synchronized models are simultaneously modified can be directly mapped to conflicts in a single model: After propagating changes from the source model to the target model, such conflicts constitute in the form of single-model conflicts. Examples are

[16]`https://subversion.apache.org/`

[17]`http://git-scm.com/`

[18]In fact, even static-semantics conflicts cannot be solved automatically, because such systems also do not know the static semantics of the models.

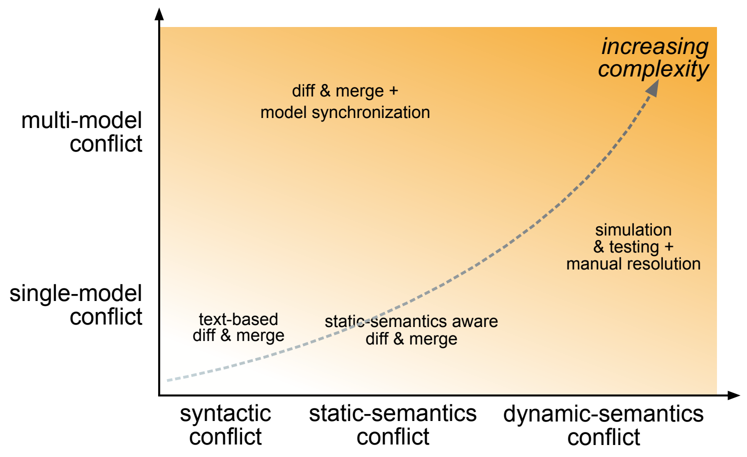

increasing complexity
diff & merge + model synchronization
multi-model conflict
single-model conflict
simulation & testing + manual resolution
text-based diff & merge
static-semantics aware diff & merge
syntactic conflict
static-semantics conflict
dynamic-semantics conflict

Figure 4.22: Dimensions of editing conflicts and its resolution approaches

simultaneously changing an attribute or a reference of two corresponding source and target model objects, or moving corresponding objects to another position. The model merger can solve some kinds of conflicts automatically, some require a user decision.

We used several examples of model-based development processes[19] to derive a categorization of conflicts and what resolution strategies are reasonable [Gos10].

In the following, we assume that the developers A and B change their models a resp. b. Table 4.3 shows an overview of the different types of conflicts that can be caused by a simultaneous evolution of two synchronized models. If there is more than one resolution strategy, the default strategy is printed in bold.

Generally, it is similar if either two developers change the same model in a conflicting way or two developers change two separate but related models such they are in conflict. The solution strategies are the same. The difference in these cases is that, in the latter case, the conflict constitutes only after an incremental update, whereas in the first case the conflict becomes visible as soon as both developer try to commit their changes to a repository.

4.3.1.3 Conflict Resolution Strategies

One simple, naive strategy to deal with inconsistencies is to require an immediate conflict resolution, such that the developers reestablish the consistency by merging the changes, possibly overwriting one of the changes. However, in distributed, highly multi-disciplinary development processes like the development of mechatronic systems, immediate resolution of interdisciplinary conflicts often requires several developers to discuss the issue, diverting those developers away from their main tasks. If interdisciplinary consistency is not strictly necessary, it

[19]Examples considered include: development of the RailCab system (active structure, behavior–states, behavior–activities, MechatronicUML, MATLAB/Simulink and Stateflow), development of the TGG Interpreter (primarily EMF class diagrams), AUTOSAR development (software architecture models)

Editing operations	Description	Resolution(s) (**Default**)
Modify – Modify (Attribute)	Both A and B changed a value of an attribute. These attributes directly correspond to each other.	Manual resolution
Modify – Delete	A modified an object that was deleted by B.	**Manual resolution**; Enforce deletion
Add – Add	Both A and B added an object at the same position.	Automatic resolution possible if both objects could also be added after each other (i.e., two additional objects are allowed, and the objects are not identical). Manual resolution required otherwise.
Use – Delete	A added a new use (e.g., a link) of an object that was deleted by B.	Manual resolution
Use – Move	A added a new use (e.g., a link) of an object that was moved by B.	The link can be automatically redirected to the new position of the object.
Modify (Object) – Move	A modified an object that was moved by B.	The modifications can be applied to the moved object automatically.
Move – Delete	A moved an object that was deleted by B.	**Manual resolution**; Enforce deletion
Move – Move	A moved an object that was also moved by B.	Manual resolution

Table 4.3: Types of conflicts by causing editing operations (adapted from [Gos10])

may be more reasonable to keep that inconsistency for a certain amount of time. Moreover, especially in early phases when there is no "consolidated view on the system under development yet, the conflicts might provide valuable information on the various intentions of the modelers" [WLS+13]. Thus, it is reasonable to tolerate inconsistencies in certain situations. In the literature, the term "living with inconsistencies" [EC01, GMT99, HN98, Bal91] has been used to describe this.

Delaying the inconsistency fixing (i.e., resolving several of such inconsistencies all at once) has the advantage of a reduced overhead due to less interdisciplinary conflict resolution meetings, where the different developers have to agree on solutions. However, it comes with the drawback of an increased difference between the models for which the consistency must be reestablished. This will make it harder for both automatic tools and the engineers to fix these inconsistencies.

There exist some automated merging tools for models in general, e.g., EMF Compare [Ecl13]. Other approaches focus on specific kinds of models. For instance, the approach of Gerth at al. [GKLE11] is targeted towards process models, and Brosch et al. [BSWK12] present a special conflict visualization for UML models. These tools can resolve some of such conflicts automatically, but user interaction is still necessary in many cases. Moreover, dealing with conflicts in an interdisciplinary, multi-model setting requires a tight integration of the model merging with the model transformation, especially if conflicts are due to changes in another discipline's model. Therefore, we need both improved automated model conflict resolution approaches as well as a better support for the users to visualize and resolve conflicts that cannot be resolved manually. Furthermore, current model merging tools suffer from worsening matching results in case of large differences in the model versions.

Comparison, merging and conflict resolution for (single) models is a research topic in itself [RK10, RK11, BCE+06], and the tools in use involve complex algorithms. Recognizing that conflict resolution on (single) models is a difficult task already, we decided not to directly incorporate conflict detection and merging into the model synchronization algorithm. Instead, we use an external dedicated model differencing and merging tool to help resolving conflicts. In this way, we could also easily integrate tools dedicated to certain languages/metamodel if reasonable, like, for instance, a special conflict visualization for UML models [BSWK12]. Then the results from this merging tool are used as an input for the model synchronization process.

Using a pure model merging approach has some drawbacks, namely performance and merging precision. First, we need to run two transformations instead of performing an immediate, bidirectional synchronization. Second, we need to perform the actual model differencing and merging. The more changes to the models have to be merged, the less precise the model merger is. As it does not have any knowledge about the correspondences between the models, it cannot use this information to help resolving conflicts. To improve both the performance and the merging precision, we integrate the model merger and the synchronization algorithms such that we try to automatically synchronize non-conflicting changes using our improved model synchronization and then using the model merger for the remainder.

Next, we have a look at related approaches for dealing with concurrent modifications in related models.

4.3.2 Related Work

In the following we give a short overview of the related work. We focus on transformation techniques that are incremental and provide a destructive propagation method (cf. Sect. 2.2.2).

4.3.2.1 Xiong et al. (2009, 2013)

Xiong et al. [XLH+07, XSHT09, XSHT13] present an approach how to achieve a bidirectional synchronization using unidirectional incremental updates (using ATL as transformation language). The idea is to first propagate the source model changes to a new version of the target model. Then Xiong et al. use a model merging tool to combine the new target model with the user-edited target model. Next, they run an incremental backwards transformation to propagate the changes to the source model. Finally, they run a conformance test to ensure all user changes in the source model are still present.

Our approach is based on this idea. We use the same basic propagation principle of first propagating the changes from one model and then using a model merger to merge the results and resolve potential conflicts. Xiong's developed his approach for ATL transformations. In contrast to ATL, TGGs are typically bidirectional by construction. Thus, we do not need the conformance tests Xiong performs at the end of the synchronization.

4.3.2.2 Körtgen (2009)

Körtgen [Kör09] also allows synchronizing simultaneous changes to both models in her TGG-based solution. Similar to conflicting rules in a transformation, editing conflicts have also to resolved in a manual fine-grained manner. However, there is no explicit support for directly conflicting changes. The user has to resolve these conflicts within the model integration environment. Thus, it typically can only be performed by a synchronization engineer and not by engineers of the affected discipline(s).

4.3.2.3 Conclusion

Both approaches have their advantages and drawbacks. In Körtgen's approach, the engineers use the model integration environment to solve the conflicts in both models (with the help of the transformation specification). However, system engineers with in-depth knowledge of the transformation to the different disciplines are typically only rarely available. Hence, it is reasonable to move the conflict resolution at least partially to a discipline-specific model, so that the discipline's engineers can perform it with the help of specialized merging tools. On the other hand, performing model merging only based on the system model or a discipline's model has performance and precision drawbacks.

4.3.3 Model Comparison for Merging Concurrent Modifications

The general idea of our approach is to use *model merging* to synchronize the concurrent modifications in the models. Our approach is based on the approach of Xiong [XLH+07, XSHT09, XSHT13], which was developed for ATL, a hybrid model transformation language. We adapted this approach to fit to a TGG-based model synchronization.

Figure 4.23 visualizes the idea of our approach. A precondition is that all potentially conflicting changes by different developers to the single models have been merged (no single-model conflicts left). This can be achieved using model-merging techniques like EMF Compare. In the scenario depicted in Fig. 4.23, this means that all changes from different software engineers have been merged into version 1.2 of the software model, and all changes to the system model have been merged into version 1.1 of the system model.

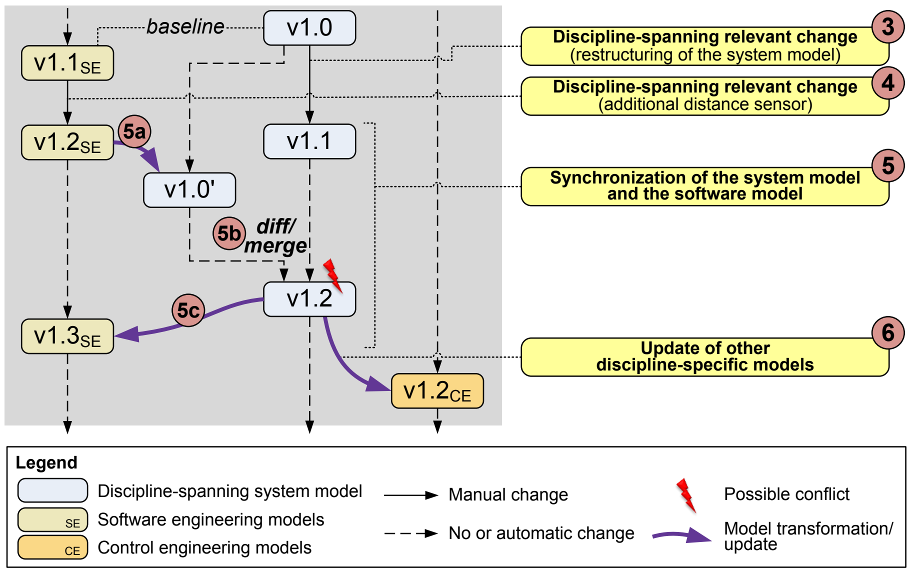

baseline
v1.0
v1.1SE
v1.2SE
5a
v1.1
v1.0'
5b
diff/ merge
v1.2
5c
v1.3SE
v1.2CE
3
Discipline-spanning relevant change (restructuring of the system model)
4
Discipline-spanning relevant change (additional distance sensor)
5
Synchronization of the system model and the software model
6
Update of other discipline-specific models
Legend
Discipline-spanning system model
SE Software engineering models
CE Control engineering models
Manual change
No or automatic change
Possible conflict
Model transformation/ update

Figure 4.23: Using model merging to synchronize simultaneous changes

We first propagate the changes from the discipline-specific software model to the discipline-spanning system model (step 5a).[20] To do so, we use version 1.0 of the system model and version 1.1_{SE} of the software model – the model versions that we last synchronized – as baseline[21]. Since version 1.0 of the system model

[20]The direction in which the changes are propagated is not important. It is, however, reasonable that the merging is performed on the model that the engineer performing the synchronization knows best. In cases where different disciplines may be affected, the a system engineer should perform the conflict resolution using the discipline-spanning system model and in consultation with the discipline engineers.

[21]A baseline is a point in time that serves as a basis for defining changes. In a baseline, all development artifacts are typically in a consistent state. When merging different versions of an artifact, this baseline is typically the most recent common ancestor, i.e., the version from which the changed artifacts were derived.

contains no changes, no conflicts could arise during this propagation. The result is version $1.1'$ of the system model, which now contains only the changes from the software model, but not the changes that have been performed by the system engineers in the meantime.

Next, we use an external model comparison/merging tool to merge the changes of both versions (step 5b). The result is version 1.2 of the system model. In this step, conflicts may arise. The merger may be able to solve some of these conflicts automatically, but some of them require user interaction.

Once all conflicts are resolved, we can use this merged system model 1.2 to perform a propagation to the software model (step 5c). We get version 1.3_{SE} of the software model as a result, which contains both the changes from the software model and the system model itself. We set these final models as the baseline for future synchronization. Furthermore, the changes can be propagated to other affected discipline-specific models (step 6).

Goschin [Gos10] implemented this approach in the course of his Bachelor Thesis.

A drawback of this approach is that the model comparison/merging tool requires a baseline version of both the source and the target model to perform a three-way merge. Usually, the engineers simply overwrite the baseline versions with new versions that contain their changes. Therefore, we have to explicitly store the last common ancestors for source and target model to be used in future model synchronization runs.

The model synchronization from the system model to the different discipline-specific models do not always take place at the same time. For instance, it may happen that only the software model is synchronized with the system model, but the synchronization with the control engineering model is delayed. We must store such a baseline for every model mapping, because the baseline for the mapping between the system model and the software model is different to the baseline from the system model to control engineering.

However, in most engineering processes, revision control tools are in use. These tools already store the complete version history of all models. Although we did not implement it, integrating our model synchronization with revision control could be used to retrieve the required baseline versions. In this way, no additional storage space will be required.

4.3.3.1 Automatic Resolution

Next, we illustrate automatic conflict resolution capabilities with an example that Fig. 4.24 shows. On the left side of that figure, the system engineer restructures the system model such that the Distance Processing and the Distance Sensor system elements are moved into a new Distance Measurement system element to improve the encapsulation of system functions and to allow a better reuse of that solution. On the right side, you see that the software engineer also performs changes to their model: Due to safety concerns, they introduce a second Distance Sensor (cf. Sect. 3.1).

These changes affect model elements that correspond to each other: The Distance Processing component/system element is affected both by the restruc-

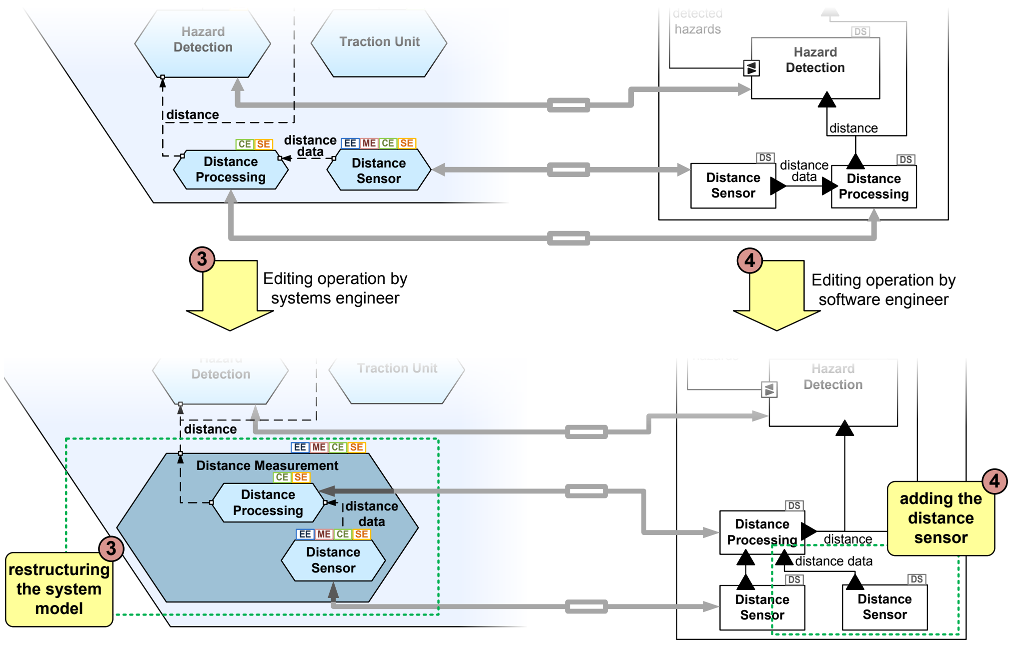

Hazard Detection
Traction Unit
distance
distance data
Distance Processing
Distance Sensor
detected hazards
Editing operation by systems engineer
Editing operation by software engineer
Distance Measurement
restructuring the system model
adding the distance sensor
3
4

Figure 4.24: Changing software engineering model and system model

turing and by the new Distance Sensor (as it requires to add a new port to that Distance Processing component).

However, this conflict can be resolved automatically. To do so, our approach first transforms the changes of the software model to the system model (step 5a in Fig. 4.23), using the baseline version of the system model (because this was the last version for which we can be sure that it was consistent with the last synchronized version of the software model). Figure 4.25 shows the result.

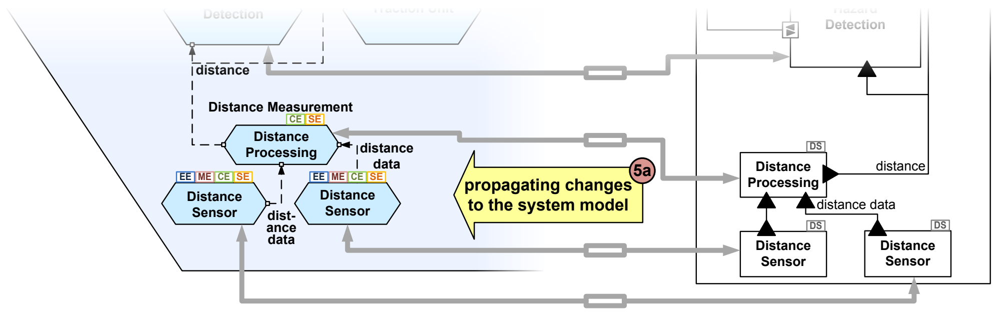


Figure 4.25: Automatically propagating software engineering changes to the system model

Now, our approach performs the actual differencing and merging (step 5b), as illustrated in Fig. 4.26. The manually changed system model is shown on the left, and the system model update with the changes from the software model is shown on the right side of that figure. Using an dedicated model merger, we are

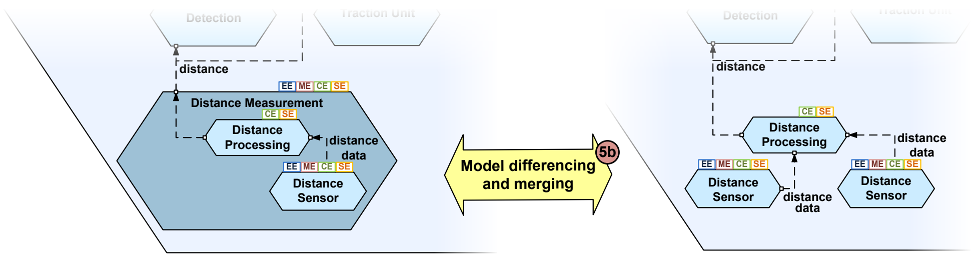


Figure 4.26: Merging the manual changes from the system model and the propagated changes from the software model

able to merge these changes without a conflict, as the changes are of type *Modify (Object) – Move*, which is a conflict type that can be automatically merged.

Finally, we use the merged model to consistently update the software model (step 5c). Fig. 4.27 shows the result of the process.

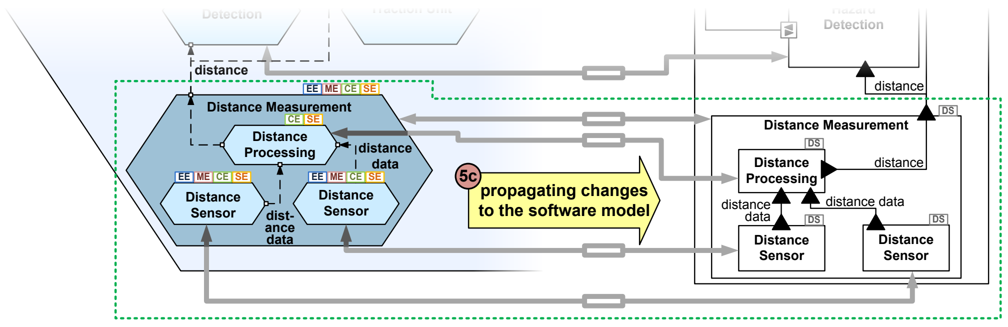


Figure 4.27: Result: Automatically merged software engineering model and system model

4.3.3.2 Interactive Resolution

Our approach decouples the model synchronization from merging and resolution of editing conflicts. As the interface between the model synchronization and the model comparison just consists of the different model versions, every model comparison approach can be integrated easily. We could even use different comparison tools for different types of models.

One particular advantage is that we can rely on existing solutions of the external model comparison tools. These tools are specialized on the comparison and conflict resolution and provide sophisticated means to visualize the conflicts and help the engineers resolving them.

4.3.4 Improving Conflict Resolution

The approach described above is generally capable of handling all kinds of conflicts, either by automatic or manual (user-interactive) resolution. However, model comparison and merging is a difficult problem in practice that relies heavily on heuristics [LAS+14]. When developing safety-critical systems, a consequence is to thoroughly review the merging results. This induces an additional manual effort, which we would like to minimize.

The imperfect merging process is mainly due to the heuristics that have to be used if models differ significantly. The correspondences that exist between the different models can help to improve the merging: If the model comparison algorithm is not sure whether two model elements in two versions of the system model originate from the same model element, maybe the corresponding model elements in the software model can be matched with a higher precision. This, in turn, would increase the certainty for the model elements in the system model.

However, this information is not available to the model merging tool. One approach is, therefore, to allow the model comparison/merging tool to access this correspondence information. This requires providing an interface for the comparison tool, and extending the comparison tool such that it accesses this information (i.e., also performs a model comparison for corresponding elements in the other model).

On the other hand, this increases the effort for the comparison as well as the coupling between the model comparison and the model synchronization technique. As explained above, we aim for minimal dependencies between both. Thus, we decided to pursuit another approach. Instead of providing the model merger with additional information, we try making the job of the model merger easier by making the differences between the model versions that have to be compared smaller. This reduces the difficulty of the differencing/merging process and, thus, increases both accuracy and time efficiency.

However, integrating sophisticated merging facilities into the model synchronization approach is also inadvisable due to the complexity of that problem. Thus, we only synchronize source model changes that can be propagated without affecting target model changes. Generally, the idea is that, for each source model change, we determine which rule application(s) are affected, and whether manipulating these rules application(s) may affect target model changes. If no changed target model element is affected, we may immediately propagate the change using model synchronization techniques. If we cannot be sure not to affect changes in the target model, we leave the merging/conflict resolution to the model comparison (“pessimistic synchronization”). Figure 4.28 illustrates this approach.

For the model comparison to be able to perform a three-way merge, we need a common ancestor of the system model that was changed by the system engineer (v1.1) and the system model that was updated with all the changes from the software model (v1.0”). Version 1.0 is such an ancestor; however, it does not contain the non-conflicting changes from the software model. The model comparison/merging tool would thus identify all these non-conflicting changes in both branches – an unnecessary effort. Thus, we generate a *virtual common ancestor*. In step 5a, we do this by propagating non-conflicting changes not only to version 1.1 (creating v1.1’), but also to version v1.0 (creating v1.0’). This model v1.0’ serves as the virtual common ancestor for the model comparison/merging tool. Next, we propagate the remaining, potentially conflicting

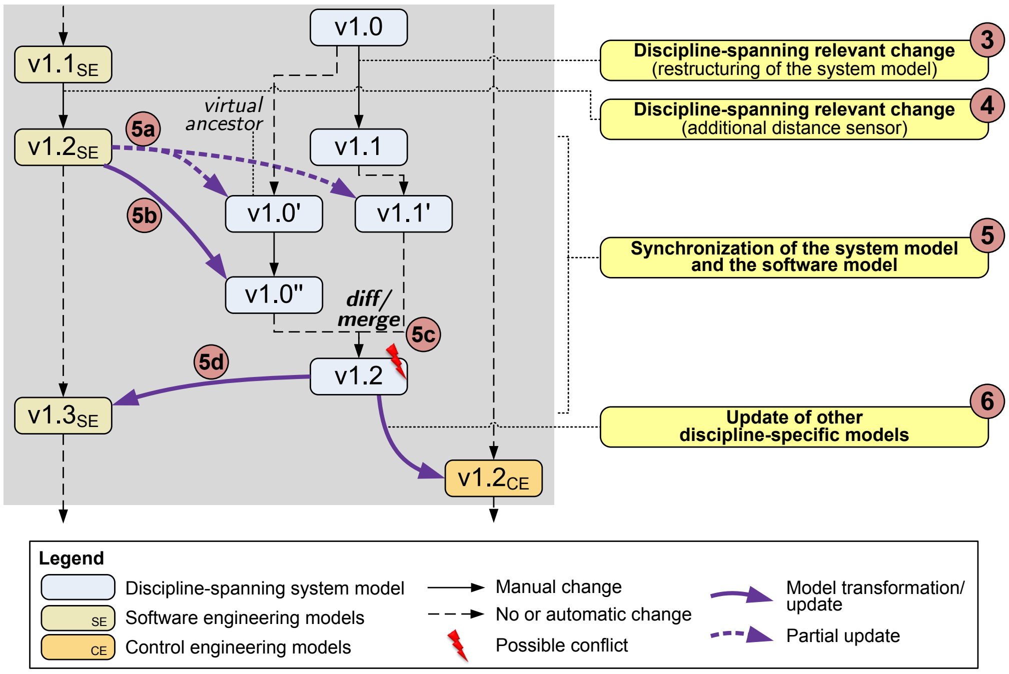

v1.0
v1.1SE
v1.2SE
virtual ancestor
v1.1
v1.0'
v1.1'
v1.0"
diff/ merge
v1.2
v1.3SE
v1.2CE
5a
5b
5c
5d
3 Discipline-spanning relevant change (restructuring of the system model)
4 Discipline-spanning relevant change (additional distance sensor)
5 Synchronization of the system model and the software model
6 Update of other discipline-specific models
Legend
Discipline-spanning system model
SE Software engineering models
CE Control engineering models
Manual change
No or automatic change
Possible conflict
Model transformation/ update
Partial update

Figure 4.28: Overview of the improved merging and conflict resolution approach

changes to create version 1.0" of the system model (step 5b). In this way, only the potentially conflicting changes will be processed during comparison/merging (step 5c). Eventually, we update the software model with the merging results (step 5d).

It is crucial that only non-conflicting changes are propagated in step 5a. However, determining whether a change could be propagated without affecting changed target model parts is not a straightforward task. Let us assume a software engineer changed something in the software model, now regarded as source model. The result is that one or more rule applications are not valid any more. The problem is that a rule application may not only influence its target produced parts. Other rule applications may rely on that rule application, i.e., they use parts of its produced parts as context. Propagating the change by revoking the rule application will also render these dependent rule applications inapplicable.

Therefore, before revoking a rule application, we must check that there are no changes in the target model either in the affected rule application or in other, dependent rule applications. In many cases, such a pessimistic approach will effectively prevent the synchronization of many changes, because a change in one of the “top” rule applications (i.e., applications of a rule on which many other rules depend) will overlap large parts of the model. As depicted in Fig. 4.29, the rule application that translated the changed source model element (top-left red hexagon) may influence a large area of the target model (dark-purple triangle in Fig. 4.29). A change in that area of the target model (right red hexagon) can therefore be in conflict with that source model change.

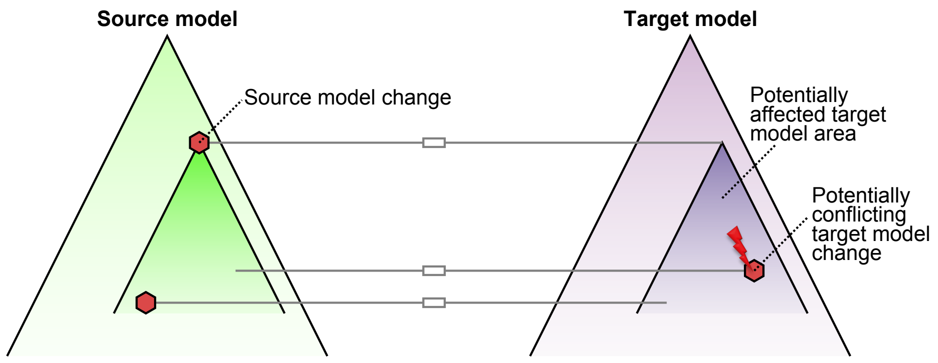

Source model
Target model
Source model change
Potentially affected target model area
Potentially conflicting target model change

Figure 4.29: Changes in "top" rule applications may impede synchronization

As all single-model editing conflicts in the source model have been resolved before, further changes in the source model (bottom-left red hexagon), however, may not be in conflict with that first change. It is, however, reasonable to propagate the changes in a top-down order, as Giese and Hildebrandt [GH09] point out.

However, as described in Sect. 4.1.2, we can propagate many changes by either re-enforcing attribute constraints, or revoking a rule application and immediately applying the same or another rule at that position. For example, if an attribute was changed, the synchronization algorithm tries to re-evaluate and re-enforce the attribute constraints. This does not revoke the rule application, and no dependent rule applications are affected. Even if revoking a rule application cannot be avoided due to structural changes in the source model, immediately re-applying a rule at that position often reestablishes the context of dependent rules.

Figure 4.30 shows the overall activities of our improved approach for synchronizing concurrent changes. After identifying changes and determining which changes may be in conflict with each other, we propagate non-conflicting source changes to the target model in order to create a virtual ancestor. Here we call the actual incremental update algorithm (*call behavior*). We reuse the incremental update algorithm presented in Sect. 4.1, but in a way such that we only propagate non-conflicting changes. Figure 4.31 shows the details of this sub-activity.

The algorithm first checks for each change in the source model whether the corresponding rule application still holds (graph structure and constraints). Attribute changes can easily be propagated. If a rule application has become invalid, we revoke this rule. However, we do not delete elements right away, but we again use the *mark for deletion* facilities (cf. Sect. 4.1). Next, we try applying the same or other rules as a replacement for the revoked rule application (potentially reusing elements marked for deletion).

We use a rule application dependency graph to calculate if there exist dependent rule applications that are affected by target model changes. In case there are no such target model changes, the change can be propagated without further

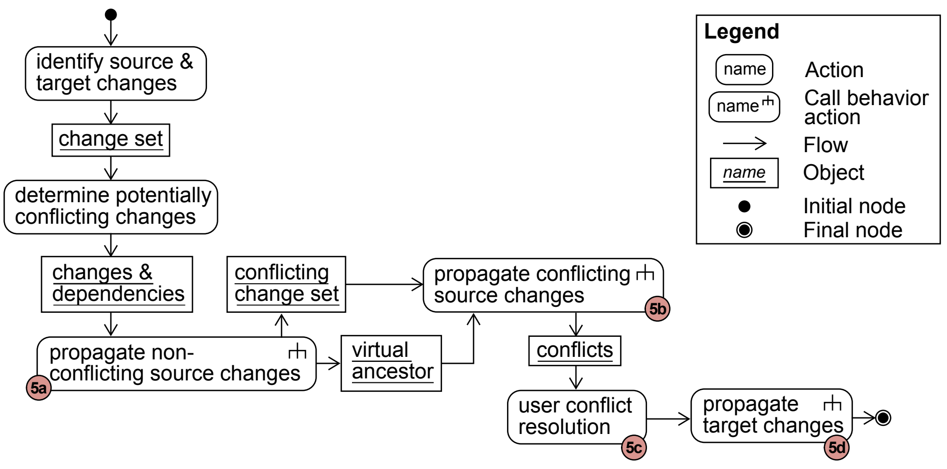


Figure 4.30: Activity diagram of the improved approach for synchronizing concurrent changes

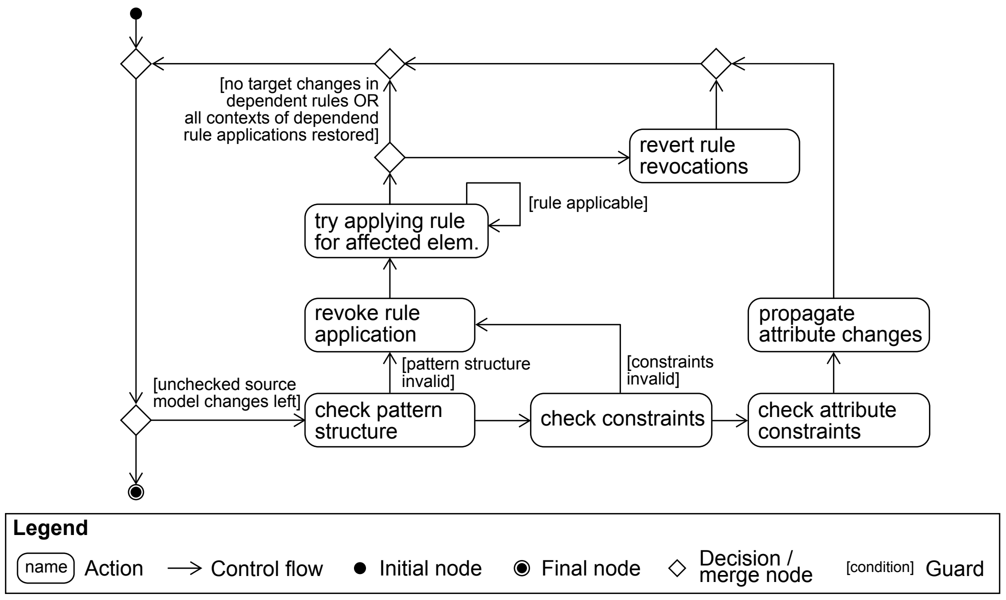

[no target changes in dependent rules OR all contexts of dependend rule applications restored]
revert rule revocations
[rule applicable]
try applying rule for affected elem.
revoke rule application
propagate attribute changes
[pattern structure invalid]
[constraints invalid]
[unchecked source model changes left]
check pattern structure
check constraints
check attribute constraints
Legend
name Action
Control flow
Initial node
Final node
Decision / merge node
[condition] Guard

Figure 4.31: Activity diagram of the activitiy propagate non-conflicting source changes

precautions.[22] If there are target model changes in dependent rule applications, we have to make sure that these are not affected by the propagation, i.e., we have successfully reestablished the context and constraints for all dependent rule applications. Whenever we are unable to reestablish the validity of dependent rule application, propagating the change could easily overwrite changes in the target model, because we also have to revoke some of the dependent rule applications that contain target model changes. Thus, we have to revert the rule revocation in such a case. The potential editing conflict has to be solved by the model comparison/merging tool then.

As described above, we also apply this partial incremental update to the unchanged baseline target model version 1.0 (cf. Fig. 4.28). The result is the new virtual common ancestor. We use the basic approach described in Sect. 4.3.3 for the remaining, potentially conflicting changes. If no unprocessed changes remain, we skip this step.

In contrast to the basic approach, this improved synchronization of concurrent changes requires a third synchronization run (in addition to the two incremental update runs). This is the partial incremental update of non-conflicting changes (step 5a).[23]. On the other hand, smaller differences between the models that the model comparison/merging tool has to process should lead to improved merging results. Evaluation results indicate that this proceeding will in fact increase the merging precision, but only for some types of models that are large and have large change sets (cf. Sect. 6.3.2).

4.3.5 Summary

In this section, we presented an approach how to integrate existing model merging techniques with our model transformation and synchronization technique based on TGGs. All editing conflicts are mapped into a single model using our novel incremental update algorithm, performing a merging of the non-conflicting model parts. We then use an existing model comparison/merging tool for the resolution of editing conflicts in this model. Our approach tries to improve the model merging reliability, as the model merger has to deal with fewer changes. By using sophisticated means of existing, specialized model merging tools, we ease the resolution of editing conflicts that cannot be solved automatically. The engineers are thereby relieved from much error-prone work during the model synchronization and conflict resolution task.

[22]Further source model changes in dependent rule applications cannot cause editing conflicts. Thus, the synchronization can also propagate them.

[23]Although this incremental update has to update two models (v1.0 and v1.1), it is sufficient to perform the rule matching for model v1.1 only. Because only non-conflicting changes are processed, the propagation is exactly the same for models v1.0' and v1.1'. Hence, we can simultaneously modify both models when applying/revoking rule, which causes only a relatively small additional effort.

CHAPTER 5

TGG Extensions

Contents

This chapter introduces a) extensions to the formalism of Triple Graph Grammars (TGGs) that increase their expressiveness, and b) concepts that ease the development of TGG-based model transformations. The presented extensions were required to implement the mappings from the CONSENS language to the models of software engineering and control engineering that are presented in this thesis. However, they are no “special purpose” extensions, i.e., they can be applied in a wide range of transformation scenarios. We furthermore developed additional concepts that allow a more intuitive representation of a transformation specification and means for transformation execution analysis. They help transformation engineers to implement and maintain TGG-based model transformations.

First, we introduce how we implemented attribute constraints and applications conditions in Sect. 5.1. Section 5.2 presents the concept of nested transformation, where a TGG may have other TGGs embedded in its context. Section 5.3 describes how to use the concrete syntax of the modeling languages within a TGG rule. Finally, we present a debugging concept for TGGs in Sect. 5.4.

5.1 Constraints and Application Conditions

In their most basic form, Triple Graph Grammars just describe how to map between typed-graph-based structures. A single TGG rule describes a (partial) mapping between certain elements. In terms of MOF [OMG06] this means that we can only map between object structures, but can neither define anything on their *attributes* nor explicitly define patterns that *must not* exist.

As most models make heavy use of object attributes, allowing arguing on attributes is crucial for the practical applicability of TGGs. Thus, attribute constraints have been suggested as extensions to TGGs [GLO09, AVS12, LHGO12]. We describe how we implemented attribute constraints in Sect. 5.1.1.

Allowing negative patterns is also important from a practical perspective. To allow such additional constraints, we extended the concept of attribute constraints to general constraints. Similar to attribute constraints, these constraints must hold after a rule has been applied, but also after the transformation is complete and after potential future incremental updates. In other words, constraints in applied rules are *invariants* on the resulting model triple. Details how to process such general constraints can be found in Sect. 5.1.2.

So far, the semantics of application conditions is only formally defined for simple attribute value constraints. The semantics of general application conditions (that argue also on object structures) requires further attention, as a naive evaluation of such application conditions may result in incorrect transformations. Thus, we developed an advanced concept of such application conditions, which is described in Sect. 5.1.3

5.1.1 Attribute Value Constraints

Attribute value constraints allow transforming attribute values between the two models. Generally, these are equations with the two corresponding attributes on each side. For instance, given a source node a whose class has an integer attribute *count*, and a target node b whose class has an integer attribute *number*, a simple attribute constraint could be $a.count = b.number$. Such a constraint defines that for a rule to be correctly applied, (besides a correct structural isomorphic matching) the attributes of the bound model elements must be equal.

When interpreting a TGG for transformation, we also need to consider these attribute constraints. When transforming in forward direction, we match the source model elements and create new target (and correspondence) model elements. Therefore, we need to interpret attribute constraints as an assignments on the respective target model element: `b.number := a.count`. In backward direction, the assignment is `a.count := b.number`.

In this case, the assignments can be easily computed, and it is obvious that both assignments will make the original attribute constraint hold. However, this is not possible in general. As an example, consider a mapping between two different types of Java-like class models: The source model has just one String attribute for the class name, fullyQualifiedName, whereas the target model has two different attributes, package and classname. A mapping between these attributes can be described with the following equation: $b.fullyQualifiedName = a.package + '.' + a.classname$. Such a attribute constraint can only be converted into a target side assignment for forward transformations (`b.fullyQualifiedName := a.package + '.'  + a.classname`), but not for the backward direction: We need to split the string at the dot, but as dots may also appear in the elsewhere in the string, this is ambiguous.

Nevertheless, such attribute dependencies are often necessary in many real-world transformation scenarios. One practical solution is to manually separate an attribute constraint into two *attribute assignments*, one for the forward and one for the backward direction. However, this somewhat breaks the declarative nature of TGGs, as the attribute processing is handled in a operational manner. In particular, it is up to the developer to ensure the bidirectionality of the attribute assignment pair. These issues have been described earlier [KW07, LHGO12].

In contrast, Lambers et al. [LHGO12] describe an approach how use a (declarative) attribute constraint to automatically derive operational attribute assignments. If, as in the case of an inverse string concatenation, this is not possible, they suggest using a constraint solver to generate *all* possible attribute values. Then the user has to decide which values to use. Anjorin et al. [AVS12] describe a similar approach with a focus on the formal properties of TGGs.

Such an approach ensures the bidirectionality of a TGG. A drawback of such declarative attribute constraints is that they are either not as expressive as manually defined operational attribute assignments, or introduce a potentially large amount of user interaction.

In our example, a classname may not contain dots ("."). Thus, the "splitting point" of the fullyQualifiedName is easy to determine: it is simply the last dot in the string. Hence, the operational attribute assignment for backward direction can be specified manually. In contrast to a declarative specification, no user interaction is required.

In practical cases, such constraints on the metamodels and/or domain-specific knowledge often help creating operational attribute assignments for both directions that do not impair bidirectionality. Therefore, we decided to implement these operational attribute assignments in our TGG Interpreter. As a consequence, it is the task of the transformation engineer to ensure that the attribute assignment work bidirectionally.

5.1.2 General Constraints

Basic graph transformations only support a "positive" context (left-hand side), i.e., they define which graph structure must exist to make the rule applicable. However, often the applicability of rules has to be further restricted [SK08a]. Therefore, we would like to include additional conditions to the rules that reflect these restrictions. Mostly this is the case when we would like to define conditions that *must not be present* in the host model. Such conditions are typically called *negative application conditions (NACs)* or simply *application conditions*. Habel et al. [HHT96] introduced them for graph grammars.

Furthermore, a rule may depend on attributes of an object to be matched, which cannot be easily represented using a graph structure.[1] For instance, the rule in Fig. 5.1 contains such an attribute application condition ("se.isSERelevant") that restricts the application of that rule to cases where the se:SystemElement is relevant to the software engineering discipline.

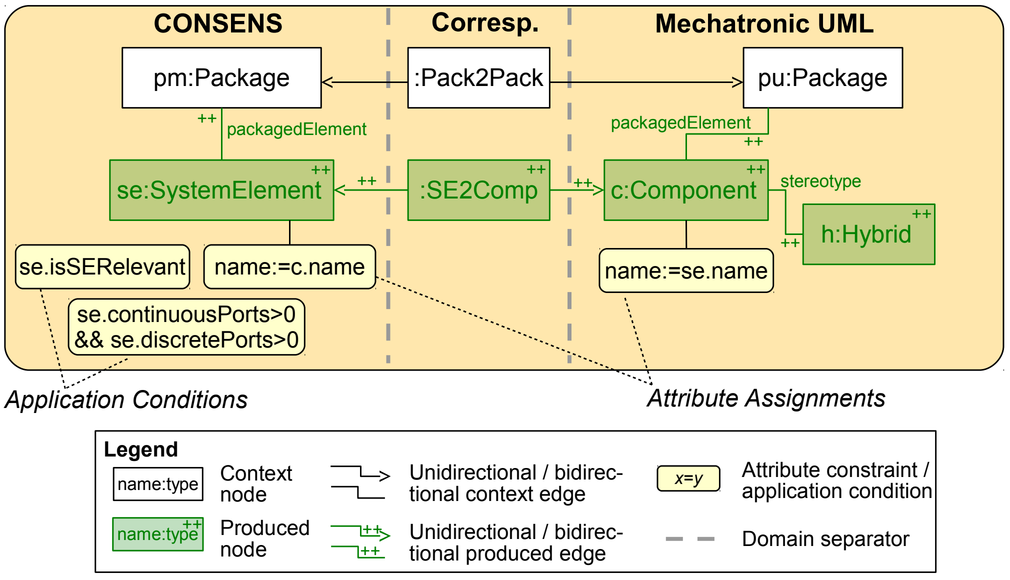


Figure 5.1: TGG Rule SystemElement2HybridComponent: examples of attribute assignments and application conditions

Most TGG approaches process such application conditions only during the rule application [GLO09, AVS12, LHGO12]. Consequently, they do not have to hold later on in the transformation process. This means that application conditions only guide the rule application process, but we cannot use them to define *invariants* that must also hold after applying the rule or when the transformation is finished.

In contrast, Wagner considers such conditions as invariants [Wag09]. *Invariant constraints* specify additional requirements on the relation the rule defines. Thus, such constraints must *always* hold, i.e., also after the transformation run is complete. Kindler and Wagner [KW07] describe such generalized constraints. Attribute constraints, as described in the previous section, are a special kind of such constraints. However, constraints may also refer to structural properties of the models, not only to attributes. Let us consider the active structure of the system model. System elements can have so-called templates, which is a special way of typing a system element. For instance, there could

[1]Attributed graphs and attributed graph grammars [EPT04] can be used for representing object attributes. Our concept builds on this formal concept of attributed graphs. It is further extended with respect to applicability to real-world scenarios; however, we do not provide a fully formal extension here.

exist several identical sensors in a system. All of these sensors then reference a template which models the general properties of this kind of sensor, e.g., input and output types and the weight and dimensions. When mapping to MechatronicUML, a system element with such a template becomes a component instance, and the template becomes the component type (cf. rule SystemElementWithTemplate2StructuredComponent in Fig. A.10). A system element without a template is mapped to both a component type and instance, i.e., it is represented by a "singleton component" in MechatronicUML (cf. rule SystemElementNoTemplate2StructuredComponent in Fig. A.7). It is important that these constraints are considered as invariants, so that they will be checked in subsequent incremental updates. Otherwise, adding a template to a system element later on would not revoke the rule and, therefore, would not change the MechatronicUML model.

We need both interpretations in different scenarios, and there are rulesets which require both interpretations. Thus, we allow both application conditions (which are checked just before applying a rule) and invariant constraints (which must always hold) in our TGG Interpreter. Note that it may require rule backtracking when an invariant constraint does not hold any more at the end of a transformation. For further details on the implementation, we refer to Sect. 6.1.7.

5.1.3 Transformation Semantics of Application Conditions

In the standard case of a (initial, non-incremental) forward transformation, we have a complete source model given. This is in contrast to the graph production semantics of TGGs (i.e., evolution of both models in parallel), where both source and target models are produced simultaneously. When executing a forward transformation, we emulate the simultaneous production semantics by explicitly binding source model element to the produced source pattern (cf. Sect. 2.4.1).

Application conditions argue on properties of the models. When evaluating an application condition on a completely existing source model, the result can be different from an evaluation in a simultaneous production scenario where the source model does not completely exist, yet. Consequently, we also have to emulate the simultaneous production semantics when evaluating application conditions in order to keep the transformation *correct* (cf. Sect. 2.4.2.2). In particular, the parts of the source model that are not translated yet must be ignored when evaluating a constraint, because they would not exist in the simultaneous production case. Otherwise, the semantics of a transformation would differ from the semantics of a simultaneous production, potentially resulting in an incorrect transformation.

As described earlier, we must distinguish between *application conditions* and *constraints*. The former are just checked before a rule is applied – they are *not a part of the context* (left-hand side). Thus, they do have to hold neither after the rule was applied nor at subsequent incremental updates or model synchronization runs. They only allow guiding the rule application process, e.g., ensuring functional behavior or avoiding "dead ends" when a wrong rule could be applied otherwise.[2]

A typical use case for application conditions is *ordered lists.* When a model element contains several sub-elements whose order is relevant, this cannot be processed using a classical, declarative TGGs: First, we cannot ensure that the elements are processed "in order", and second, we cannot define different rules for the first, second, ..., and last element [GKRT08]. For instance, when mapping between CONSENS/Behavior–States and MATLAB/Stateflow, the numbering of the priorities of outgoing transitions is differing; we have to process the lists of transitions in a well-defined order to keep the same semantics.

We described that problem in 2008 [GKRT08], and introduced a first approach for this problem. We annotate TGG edges with special constraints "«first»", "«next»", and "«last»" for the first element, all centrical elements, and the last element in an ordered list, respectively. However, this approach only partially solves this problem. While we know the first and the last element, we cannot ensure the order of all other elements. As a result, we for example cannot reverse the order of a list when transforming it to the target model.

We propose the following solution. When evaluating an application condition, the evaluation ignores elements of the source model that have *not been translated, yet*, i.e., they are *not yet bound.* For instance, when evaluating the position in an ordered list, the position is calculated only using the elements that have already been translated. An application condition `list->sizeOf() = 0` would therefore also hold for an non-empty list in case no element has been translated, yet. Obviously, such an application condition would not hold any more after the rule has been applied. Consequently, we also cannot simply reevaluate application conditions when checking an existing rule application in later incremental updates. Instead, we also only consider elements of the source model that were bound before this (currently checked) rule was applied when incrementally updating.

In combination with invariant constraints, we can also ensure that the first element in a list is also translated first: We can use an invariant constraint `list->indexOf(e) = 1` where `e` is the element to be translated.[3]

Section 6.1.7.2 describes how we implemented these semantics of constraints and application conditions in our TGG Interpreter. The described approach preserves the simultaneous production semantics of TGGs also in the transformation scenario where we have a source model given. In other words, this extended TGG semantics is *correct*, which we will prove informally next.

[2]When two or more rules are applicable for a certain element, most TGG approaches arbitrarily select one of the applicable rules. In such a situation, application conditions can further restrict the applicability of the respective rules, such that a rule can be selected deterministically. Such application conditions can also be generated automatically for avoiding dead-ends (situations where no further rules could be applied for elements that are not translated, yet), e.g., by performing analyses like critical pair analysis and identifying such dead-ends [HEGO10]. We refer also to Sect. 6.1.4 for other means to deal with non-deterministic rulesets.

[3]Different from most programming languages, OCL uses a 1-based index for ordered sets and sequences. The first element has an index of 1, not 0.

5.1.4 Correctness of Application Condition and Constraint Semantics

In Sect. 2.4.2.2, we defined correctness as follows:

Definition 11 (correctness of a transformation algorithm)**.** A TGG transformation algorithm is *correct* iff all model triples that it will produce can also be produced by a sequence of TGG rule applications starting from the axiom.

Given the correctness of the general transformation algorithm (cf., for instance, [Sch95]), we now prove (by contradiction) that a transformation extended by constraints and application conditions (called "extended semantics" in the following) is also correct.

Assume that the extended semantics is incorrect. This means that there exist at least one model triple H produced using this extended transformation semantics that cannot be derived using the simultaneous evolution semantics. If the model triple H cannot be derived using simultaneous evolution, there must be at least one applied rule that would not be applicable when simultaneously producing both models.

More formally, we have a set of TGG rules $p_1, p_2, \ldots \in M$ and an axiom A, and a pair of two models $H = (H_s \times H_t)$. As SCHÜRR described [Sch95], when transforming a given source model H_s into a target model H_t, we first interpret each rule p_i such that we try to match the complete source part (and the correspondence/target context) $p_{i,\text{match}}$ in the existing models, and then create the remainder of the rule $p_{i,\text{create}}$. With the axiom A, its target part A_t, and an incorrectly applied rule p_f:

$$\exists p_f \in M : G_0 = (H_s \times A_t) \xrightarrow{\dots} G_i = (H_s \times G_{t,i}) \xrightarrow{p_{f,\text{match}}} G_i$$

$$\xrightarrow{p_{f,\text{create}}} G_{i+1} \xrightarrow{\dots} H = (H_s \times H_t)$$

$$A \xrightarrow{\dots} G_i = (H_{s,i} \times G_{t,i}) \overset{p_f}{\nrightarrow} G_{i+1}$$

The problem could be either an application condition or a constraint that differs in its evaluation between the two semantics. Furthermore, the differing evaluation could only be due to the source model, as the target model is produced similarly in both cases. We deal with both cases separately.

If the problem is an application condition violating correctness, this application condition must hinder the application of p_f in the simultaneous evolution case although it allows applying $p_{f,\text{match}}$ in the transformation case.[4] We defined the evaluation semantics of application conditions such that only *bound elements* are considered. By definition, bound elements are the elements that would have already been produced if simultaneously evolving the models. Thus, the evaluation of the application condition will be the same in both cases.

If the problem is a constraint violating correctness, similarly, this constraint must hinder the application of p_f in the simultaneous evolution case although

[4]As the *match* rule resembles the context of the *create* rule, the *create* rule is applicable iff the *create* rule is applicable. Thus, we only have to consider one of them.

it allows applying $p_{f,\mathrm{match}}$ in the transformation case. In fact, if the constraint refers to properties of the source model that would have not been accessible in the simultaneous evolution case (e.g., because the element does not exist, yet), this constraint could be violated before p_f is applied. However, as we defined constraints so that they must *hold on the transformation result*, a temporary violation of a constraint during the rule application/derivation process is acceptable. Thus, a constraint will not hinder an application of a rule. As all model elements are created at the end of both a transformation and a simultaneous evolution, the evaluation of a constraint does not differ between these two cases.[5] Thus, there could be no difference in the semantics due to constraints.

As shown, the violation of correctness could not be due to the novel semantics of application conditions and constraints. Given the correctness of the original TGG semantics, this is a contradiction. Therefore, our assumption that the extended semantics is incorrect is wrong.

5.2 Combinatoric Distributions

In general, the expressiveness of TGG is similar to other declarative model-to-model transformation languages like QVT-R [GK10]. However, one particular problem arises in TGGs when, for instance, trying to define a TGG mapping between typed and non-typed languages. Figure 5.2 illustrates such a scenario.

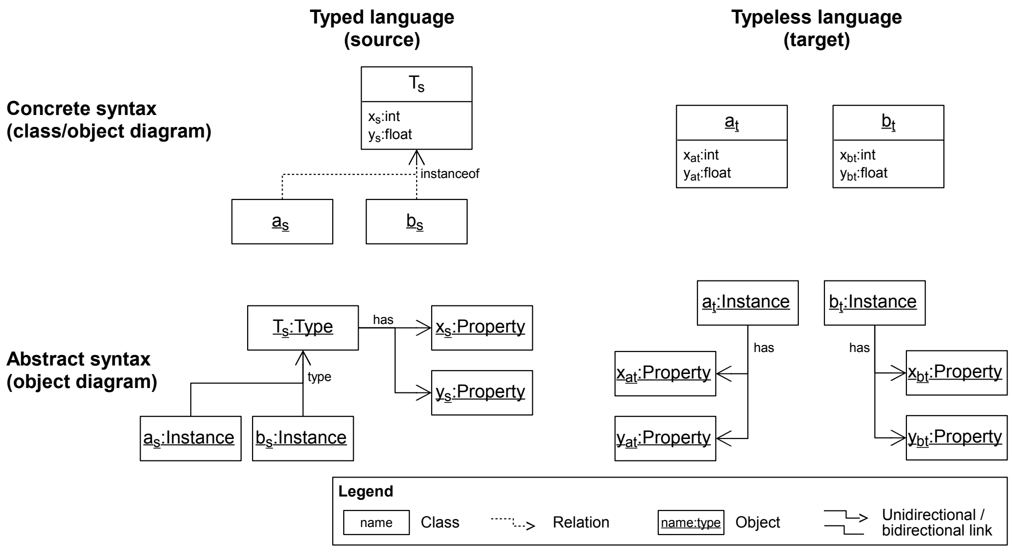


Figure 5.2: Distributing type properties into non-typed instances

[5]Note that depending on the constraint enforcement strategy, there could be cases where a certain transformation algorithm would produce an invalid result. However, the transformation algorithm has to check if all constraints hold at the end of the transformation. If this is not the case, it has to backtrack over the rule applications and/or constraint enforcements in order to find a valid transformation result. Details on the backtracking strategy can be found in Sect. 6.1.4.

As shown in the figure, we want to transform a typed language (source) into a language that does not have a (more complex) typing concept (target). There are two instances $\underline{\mathsf{a_s}}$, $\underline{\mathsf{b_s}}$ of a type $\mathsf{T_s}$ in the source model. The type $\mathsf{T_s}$ has two properties: an attribute $\mathsf{x_s}$:int and an attribute $\mathsf{y_s}$:float. As the target language does not have a type concept, we have to distribute all properties of the (source model) type into the concrete instances $\underline{\mathsf{a_t}}$, $\underline{\mathsf{b_t}}$ in the target model, as shown on the right side of Fig. 5.2.

We find such mappings, for instance, in the transformation from the (typed) CONSENS language to MATLAB/Simulink, which only allows data types, but no complex, structured types.[6] In general, where languages differ in their type system – from simple data types like in C [ISO9899] to complex polymorphism and higher-order type operators like in Haskell [Mar10] or Scala [Ode13] – mapping between languages typically requires the distribution of properties.

Figure 5.3 illustrates the problem with defining such a mapping in terms of a regular TGG. Rule Instance2Instance (top) translates the instances. As a type in the source model has no corresponding part in the target model, rule Type2Nothing (middle) simply binds the respective source model element, but does not produce any target model elements. The bottom rule TypeProp2InstanceProp now has to distribute the different type properties into the target instances. However, as we have a bind-only-once semantics for produced source nodes, we can match one property only once. Consequently, we can only translate it into one target model instance. Every further instance will not receive these properties.

This problem is a result of a lack of expressiveness due to the bind-only-once sematics of TGGs. It arises whenever not single elements or element patterns are mapped to a target model pattern, but *all combinations of more than one element* should be mapped. In the following, we call this kind of mapping principle *combinatoric distribution*.

To be able to cover the described type of mapping, we would like to extend the TGG formalism. TGGs are context-sensitive grammars (type-1 in the Chomsky hierarchy) [Sch95]. However, TGGs should not become unrestricted, because deciding whether a given graph can be generated by an unrestricted grammar is equivalent to the Halting problem and is therefore undecidable.[7] Instead, we aim for finding an extension in expressiveness that only covers this particular type of mappings.

In the following, we present different solutions for this problem and discuss on advantages and disadvantages.

[6]MATLAB/Simulink allows defining so-called library blocks, which can then be used as blocks in actual models. Although this looks like a type concept at first glance, it is basically a simple link between an block and a library element. This link is removed when the user edits the actual block; as a result, the block will become a simple copy of the library element with no typing information left.

[7]Note that using OCL as a language for constraints and attribute assignments (cf. Sects. 5.1 and 6.1.7) already makes TGGs potentially undecidable because OCL is undecidable. However, a restricted form of OCL called *OCL-Lite* is decidable in EXPTIME [QACT12a, QACT12b].

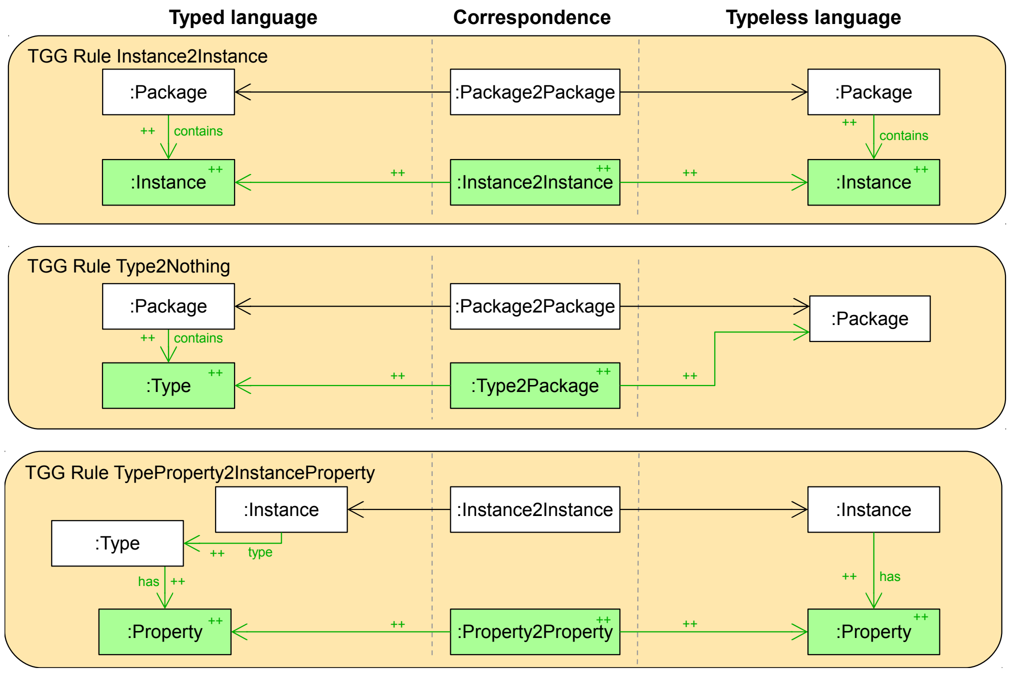


Figure 5.3: Using regular TGGs for a distribution of properties

5.2.1 Reusable Nodes and Application Conditions

Fig. 5.4 shows a simple solution: *reusable source nodes* in combination with application conditions for the source model element that has to be distributed. Reusable nodes (and edges), as introduced by Greenyer and Kindler [GK10] and extended by Greenyer and Rieke [GR12], are a combination of produced and context nodes. They serve either as a context or a produced node, i.e., a reusable node can also be specified by creating two structurally identical rules, one with a produced node and one with a context node. A transformation engine may nondeterministically decide to interpret a reusable node as a produced node or as a context node.[8] The semantics of reusable edges is equivalent. In the graphical syntax of TGG rules, reusable nodes and edges are gray and are annotated with "##".

Here, the source property is a reusable node (ps:Property). On a second application of the rule, this node can be interpreted as a context node, leaving the source produced part of the rule empty. As a result, the rule would be applicable infinitely often, creating an infinite number of properties in the target model. Therefore, we use an additional application condition to prevent the rule from being applied if there already exists a corresponding target property. This works well in forward transformations and incremental updates.

The problem, however, are incremental updates in backwards direction. If a user deletes a property from a (target) instance, there are two consistent ways of propagating such a change:

[8]It may therefore happen that a transformation engine has to backtrack over rule applications, switching between "produced" and "context" in a certain situation.

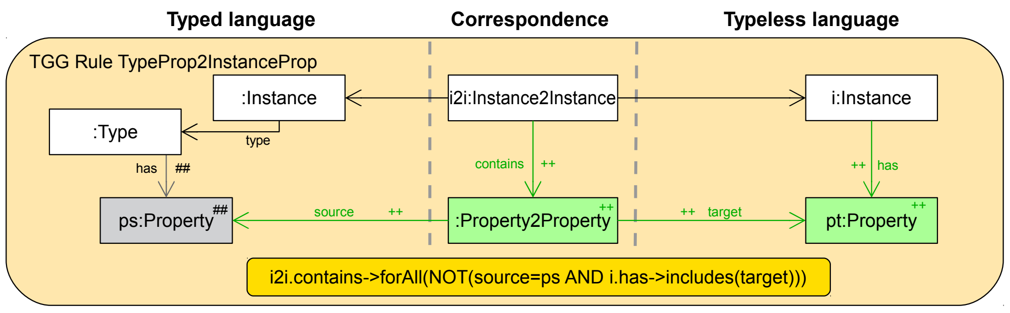


Figure 5.4: Reusable source nodes and constraints for combinatoric distribution (attribute constraints omitted)

1. Remove the property from the (source) type, and remove it also for the other target instances.
2. Remove the type from the (source) instance, and create a new type only for this instance.

In general, both are valid ways of change propagation; therefore, we have to ask the user (or define a default case). However, this is not possible with reusable nodes, as we explain in the following.

If the deleted target property belongs to the first rule application where the reusable source node ps:Property was interpreted as produced node, rule revocation would result in deleting the source model property. All other rule application, which interpreted the reusable node as context, are therefore also invalid and have to be revoked. Thus, all target properties are also deleted (first way of propagation). Although this produces a result that is correct, we cannot choose the second way of propagation.

However, with the type node interpreted as a context node, the change would simply result in revoking the rule, which changes nothing on the source model (cf. Fig. 5.5). Even worse, a future forward incremental update run would reapply the rule, effectively undoing the change in the target model.

Allowing an empty source produced part in a forward transformation violates correctness in later backwards incremental updates (the same holds for empty target produced parts in a backward transformation). Therefore, we do not allow such rules to be applied in the respective transformation direction. Consequently, using reusable nodes in combination with constraints does not solve the problem for scenarios like ours, where we want to incrementally update the models later on. For simple batch transformation scenarios, however, it is the easiest solution, as it relies only on the expressiveness of the constraint language.

5.2.2 Child Transformations

Before starting the type property distribution for one instance, we would like to "forget" that we may have already translated the type properties for a different instance. In other words, we want to remove all bindings for the (source) type properties from our bookkeeping. We propose to open a *new transformation*

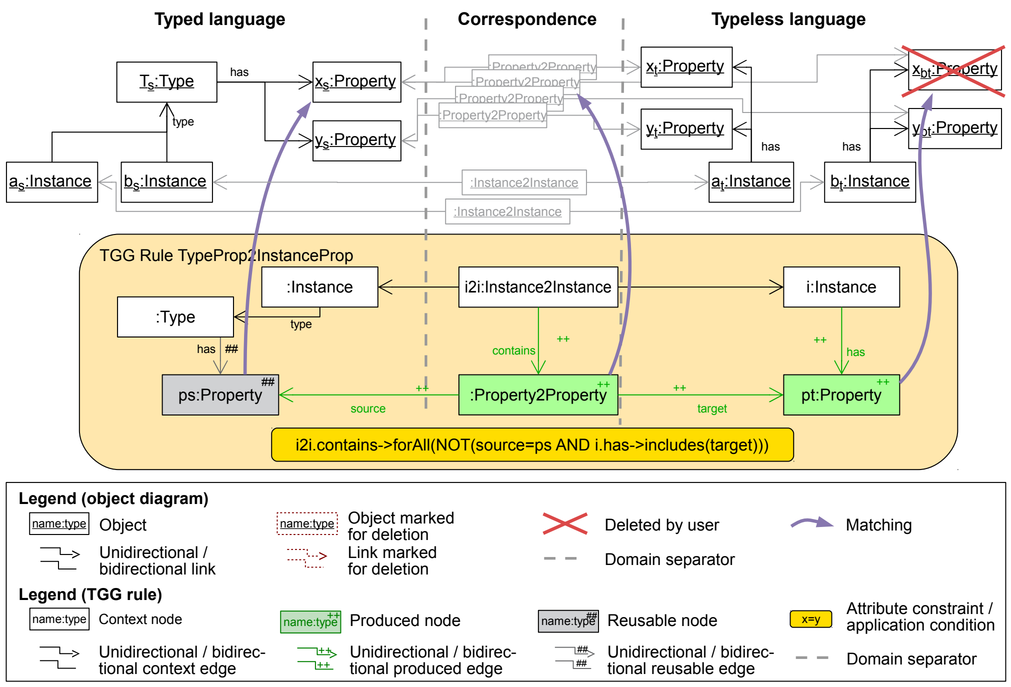


Figure 5.5: Backward incremental update: Inconsistent state

binding context in such a situation, or, informally speaking, to *start a child transformation* nested into the parent transformation.

To branch into a child transformation, we explicitly define where such a nested transformation can start in the parent transformation. Every produced node can be declared as such a *branching point*. For each valid matching of such a node, we open a new transformation context.

All existing node and edge bindings from the parent transformation are kept when opening a context. However, the bindings created in the new context are *not* carried over to the parent transformation. That means that when we apply the branching rule of the parent transformation the next time, all source elements translated in the new context can be translated again. To ensure that the new transformation context starts at the branching node, we mark one node of the first rule of the child transformation as the counterpart of the branching point.

Fig. 5.6 shows a ruleset using such an branching point. In general, it resembles the rule that translated the types (Fig. 5.3 center). The nested transformation only consists of the rule that distributes the different type properties into the target instance (Fig. 5.3 bottom). It can be applied as often as properties exist in the type.

Let us reconsider the example from Fig. 5.5. Figure 5.7 shows this example using the ruleset with the nested transformation. The rule TypeProp2InstanceProp was applied four times. Two applications cover the object xs:Property with a produced node. This is possible because every application of the rule Type2Nothing opens a new child transformation, starting with the rule

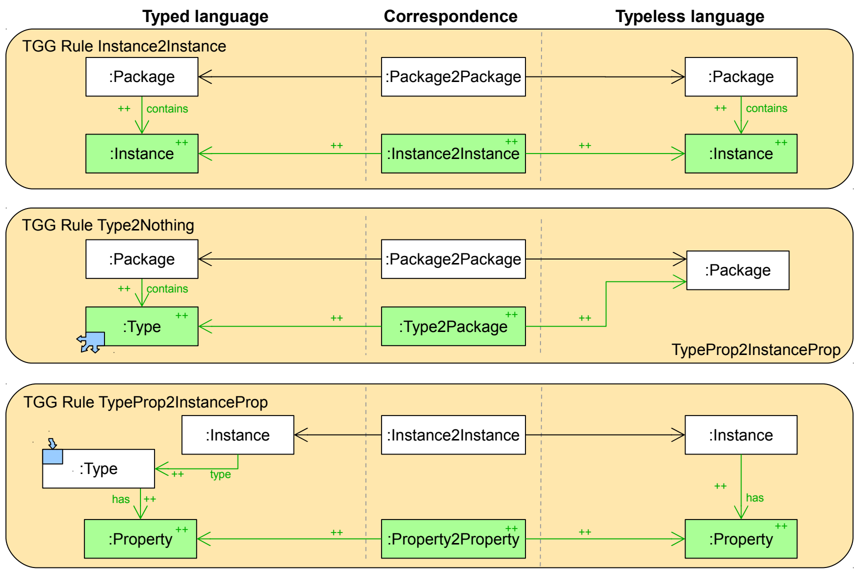


Figure 5.6: Branching point to start a nested (child) transformation

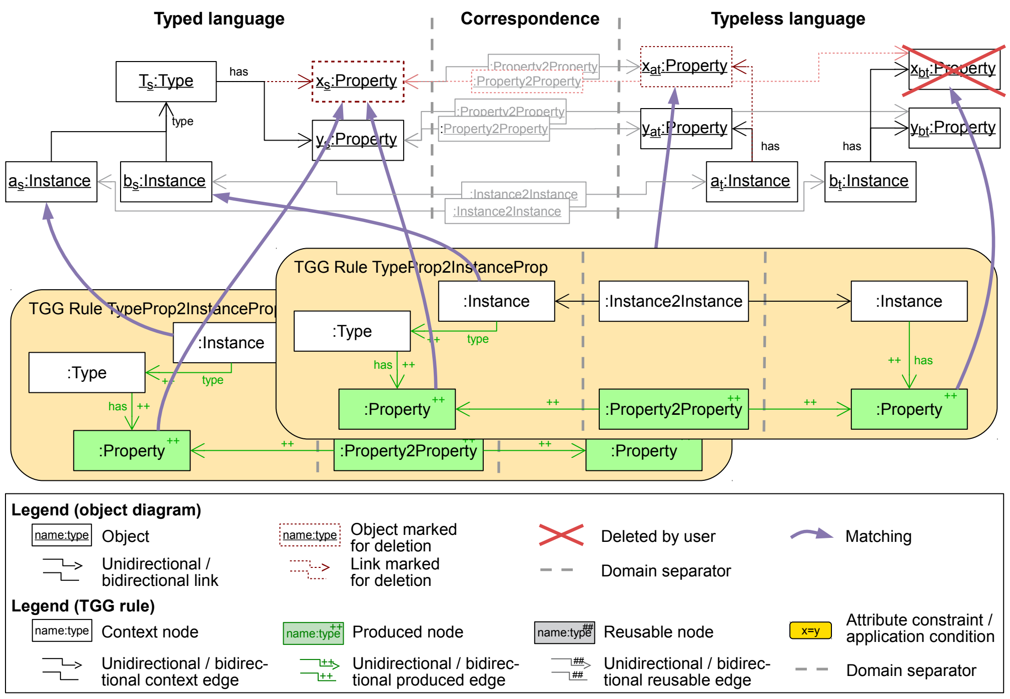


Figure 5.7: Backward incremental update: Correct incremental updates using a nested transformation

TypeProp2InstanceProp. Consequently, the object xs:Property can be bound by a produced node as often as a new child transformation is started – two times in this case. When a user deletes the x_{bt}:Property, one of these two rule applications becomes invalid (the topmost application in Fig. 5.7). Revoking this rule application means deleting x_s:Property as well as the correspondence object (denoted as dashed red border in the figure). As a result, also the other rule application that translated the x_s:Property to the x_{at}:Property becomes invalid. Revoking this rule removes the x_{at}:Property.

Concerning expressiveness and decidability, the child transformation itself is obviously context-sensitive, because it is a regular TGG. It can be decided using a linear-bounded nondeterministic Turing machine. When starting a new child transformation from the parent transformation, this is similar to a call to a subroutine, or combining two Turing machines. As we do not need to store any data from the child transformation within the parent transformation (because no binding information is carried over to the parent transformation), we do not need more than a linear-bounded tape for the nondeterministic Turing machine. Thus, an extended TGG with child transformations is still context-sensitive, i.e., decidable in NSPACE(n).

5.3 Concrete-Syntax-Based TGG Rules

Where the abstract syntax (i.e., the metamodel of a language) is most important from a tool-centered view, the concrete syntax of a language is highly critical for its users, as it represents the "user interface" to the language. According to Voelter et al., "a DSL will only be successful if and when it uses notations that directly fit the domain" [VBD+13]. Simplicity and conciseness are essential for a good concrete syntax.

A large number of model transformation languages is simply based upon a textual syntax for specifying the relations, for instance QVT-Operational [OMG08], ATL[9], Epsilon[10], JTL [CREP10], and VIATRA2[11]. In contrast, TGGs are typically represented in a graphical form, as explained in Sect. 2.4.

Discussions between graphical and textual syntaxes are "often heavily biased by previous experience, prejudice and tool capabilities" [VBD+13]. However, graphical notations are generally considered good for describing structural relationships like the highly-structured graph patterns of TGG rules.

To the best of our knowledge, no dedicated user acceptance study has been performed concerning the concrete syntax of TGGs. However, we have applied TGG transformations for different application scenarios and case studies, and many engineers (e.g., student workers, research colleagues) with differing previous knowledge about model transformation have specified transformations using our TGG Interpreter tool suite. According to the feedback we re-

[9]http://www.eclipse.org/atl/
[10]http://www.eclipse.org/epsilon/
[11]http://eclipse.org/viatra2/

ceived from these users, specifying a model transformation with TGG suffers from three main issues:

1. It is difficult to understand the potentially complex interplay between the rules.
2. Rules quickly become verbose and, hence, difficult to understand.
3. Understanding a rule may be difficult, as it requires reading the labels of nodes and edges, because the used metamodel elements are just referenced by their (textual) name.

The first issue is mainly due to the declarative character of TGGs. First, most engineers are unfamiliar with declarative languages in general, because typical computer science education as well as most software engineering projects focus on imperative programming. Second, there is no explicit visualization of the dependencies between rules. We address this issue by a) visualizing rule dependencies, and b) providing debugging facilities to help the transformation engineer understand the course of a transformation. See Sect. 5.4 for details on this topic.

In typical real-world examples, model transformations are used for non-trivial mappings, which means that the complexity of the mapping is high. Consequently, the TGG rules often contain a large number of nodes, edges and constraints. This is the main reason for the second issue.

The third issue is due to the concrete syntax of TGG rules, which is based upon the abstract syntax of the modeling languages the TGG maps between.[12] A concrete syntax of a modeling language can mask parts of the (potentially extensive and complex) abstract syntax, or can represent it more intuitively.

The latter two issues hold room for improvements. In the following, we describe how we modify the concrete syntax of the TGG rules, such that we use the concrete syntaxes of the modeling languages (instead of the abstract syntaxes).

5.3.1 General Approach

Voelter et al. mention four concerns that may be addressed when designing a concrete syntax: *writability*, *readability*, *learnability*, and *effectiveness*. These concerns may be at least partially in conflict with each other. For instance, a well-writable concrete syntax may not be equally good in its readability. One loophole for this problem is to create *more than one concrete syntax*, each of them tailored to different concerns or users [VBD$^+$13].

In the previous section, we described that one particular difficulty with TGGs is the low readability/comprehensibility of rules. In the following, we argue why creating a second, alternative syntax for TGG rules can mitigate this problem.

TGG rules can be either represented using the classical graph production syntax or the compact notation (cf. Fig. 2.25 on page 43 and Fig. 2.26 on page 44). The compact notation allows a quicker access to the rule due to less

[12]It is important not to confuse TGGs as a language with the modeling languages the TGG speaks about. TGGs have a graphical concrete syntax; however, this graphical concrete syntax reuses the abstract syntax of the modeling languages (in an object-diagram-like style).

elements shown. Furthermore, representing corresponding nodes/edges from the lhs and rhs by a single visual element increases comprehensibility, because the user can easily capture the morphism between the lhs and the rhs. Therefore, it is generally reasonable to use this compact notation for reading and writing TGG rules.

In general, the compact, abstract-syntax-based TGG syntax provides a good writability and effectiveness for experienced users. However, considering the size that complex TGG rules have (for instance, see Fig. B.13 on page 222), beginners as well as experts may have problems quickly understand the mapping a such a rule. This is because they are unable to cope with the number of elements, and they are unable to identify the essential parts of the rule, because they cannot easily map the abstract nodes and edges of the rule to elements of the modeling languages.

Users of a language typically work only with the concrete syntax of a language. In many cases, they have only limited knowledge of the abstract syntax of their modeling language. Consequently, displaying at least parts of a model-to-model mapping also with the help of the concrete syntax is likely to improve the comprehensibility of the mapping for the users. On the other hand, a transformation engineer has to understand the deeper technical aspects of a mapping. An abstract-syntax-based view of a mapping (like in classic TGGs) is probably better suited during transformation development.

We provide an *additional concrete-syntax-based view* for TGG rules that can be used to quickly understand the mapping of an existing rule. This view should be automatically generated using knowledge about the concrete syntax of a modeling language. Such an approach mainly addresses the readability and learnability aspects of a concrete syntax.

5.3.2 Related Work

Kindler and Wagner [KW07] already suggested using the concrete syntax in TGG rules. However, they neither describe how to achieve such concrete syntax TGG rules, nor did they implement it in their tool. They mention that using only modeling tools based on the EMF/GMF technology may simplify the development of such a TGG rule editor.

Körtgen [Kör09] presents a rule editor that uses concrete syntax. She gives examples where this concrete syntax looks briefer and more comprehensible than an abstract syntax rule. However, this rule editor was specifically designed for the two modeling language she used; no generalized concept how to use concrete syntax in TGG rules is given.

Grønmo and Møller-Pedersen [Grø09, GMP09] present a framework to specify concrete-syntax-based transformation rules, called “concrete syntax-based graph transformation” (CGT). These rules are not executed directly, but translated into traditional graph transformation rules. The authors tested the concept on several modeling languages, and they claim it is a generic concept applicable to arbitrary modeling languages.

Baar and Whittle [BW07] identify four main differences between models and patterns that describe models. First, objects in patterns must be labeled with a unique variable. Second, as patterns usually represents an incomplete model, well-formedness rules must be ignored. Third, objects in a pattern may be typed by abstract classes, whereas model objects are always typed by non-abstract classes. Forth, properties of objects in a patterns can contain variables, model object only have literals as values. Using this observations, they automatically transform a given metamodel into a metamodel that also works for patterns; e.g., this is done by relaxing constraints, adding a property for the label, or making all classes non-abstract. However, the problem remains that for all those changes, the concrete syntax has to be defined manually. For instance, they have to create a visual representation for abstract classes.

5.3.3 Concept

The compact notation of TGG rules consist of a context and a produced part, which are syntactically discriminated by color (b/w and green) and the "++" markings for produced elements. Generally, rules resemble the appearance of class or object diagrams. Such diagrams do not contain color, so using green syntactic elements is easily possible; also they leave room for stereotyping, allowing "++" annotations.

However, the concrete syntax of a certain language may use color as a semantically significant attribute. So coloring the visual elements of the modeling language cannot be used to distinguish between context and produced node elements. Moreover, there may be no space for additional annotations. Hence, simply transferring the "produced element" notation of abstract syntax TGG rules to concrete-syntax-based rules is not possible.

There are two possible alternatives:

- Using a different visual distinguishing feature not used in typical visual modeling languages, or
- "Boxing" the modeling language elements into classical TGG syntax.

Considering the first alternative, finding such an common unused visual feature is not trivial. For two-dimensional visual concrete syntaxes, we could use a third dimension. However, this adds a new visual complexity to the rules' syntax. This added complexity could undo the positive effects of the concrete syntax for readability.

Thus, we chose the latter alternative, "boxing" the visual elements into classical TGG nodes. Instead of showing the nodes type name inside the nodes, we use the concrete visual syntax. Figure 5.8 illustrates this idea.

Nodes may also have names, which are use to reference the node's attributes from a constraint or application condition. Elements of modeling languages often also have a name, which may also be shown in the concrete syntax of a modeling language. Therefore, we reuse the name's representation for showing the name of the TGG node.

Links between two objects can also have a more complex visual representation than a simple line. The most common case are containments: Elements may be hierarchically contained in a parent element. On the metamodeling level, such containments are just a special kind of reference. Usually, contained elements are also shown inside the parent element in the concrete syntax. This

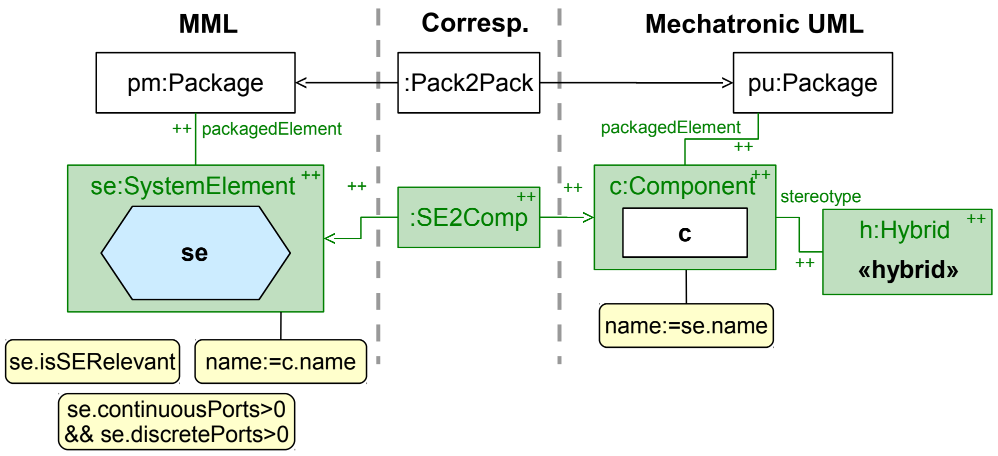


Figure 5.8: Using the concrete syntax of a modeling language in TGG rules

is, however, difficult to combine with the introduced "boxing" TGG syntax. We could simply draw the contained element into the boxed parent element. But if the parent and the contained element do not belong to the same graph pattern (i.e., one of the is a context element and the other a produced elements), the user cannot easily recognize this fact. Drawing a TGG node that boxes the contained element inside the parent element can also be confusing, because the TGG node's concrete syntax may be similar to elements of the concrete syntax of the modeling language. Consequently, we do not draw elements inside parent elements.

In rare cases, links may be also represented with node-like visual elements. To cover such cases, we suggest using annotations similar to speech bubbles, although we did not implement this into our software.

As mentioned above, we designed this concept as an alternative syntax for TGG rules, primarily intended for getting a quick overview about a rule. It is likely that it will not ease writing a rule, because the concrete syntax of the modeling language is only used in the rule's visual representation, but was not integrated into the editing tools. Possible future work could be to allow the rule designer to create TGG nodes using editing tools which also show the concrete syntax of the modeling language. This could improve rule writability especially for novice users.

5.4 TGG Debugging

Debugging is generally considered as a key aspect of software development. Providing debugging support is essential for software development environments – locating bugs in complex software is not a trivial task. It is also one core part of providing high quality software. According to Hailpern and Santhanam, "in a typical commercial development organization, the cost of [...] appropriate debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost." [HS02]

Debuggers of imperative programming languages all follow similar principles [Ros96]. Typically, they cover all or most of the following features:

- running a program step by step (*single-stepping* or program animation)
- stopping (breaking) (pausing the program to examine the current state), mainly using *breakpoints*
- showing *values of variables*
- modify the *program state*, e.g., by changing a value of a variable or the instruction pointer

Furthermore, modern debuggers provide advanced tools like a query processor, symbol resolver or an expression interpreter. Some debugger even allow reverse debugging (also known as "historical debugging" or "backwards debugging"), which allows the software engineer to reverse the execution steps of a program.

Model transformation languages can be regarded as specialized programming languages. Thus, they also need debugging support. Model transformation languages/tools are applied for mappings that are more complex. This, again, increases the need of sophisticated debugging facilities, as complex mappings tend to contain more errors and flaws. However, only little research has focused on this topic [MV11].

In a study of 59 anecdotal (imperative program) debugging experiences, EISENSTADT found that "53 % of the difficulties are attributable to just two sources: large temporal or spatial chasms between the root cause and the symptom, and bugs that rendered debugging tools inapplicable" [Eis97]. These two core problem sources are evident in current TGG tools as well. First, a transformation engineer typically has to run the complete transformation to find out that the result is incorrect. They have no direct clue which rule(s) may have cause the problem. Second and most importantly, there are no debugging facilities on the level of the TGG ruleset.[13] That means, the debugging takes place either using log output, inside a TGG INTERPRETER source code, or using code that was generated from the TGG. Similarly, you could try to find a bug in a Java program by debugging the resulting byte code and the implementation of the virtual machine.

Furthermore, TGGs are a declarative specification technique: The ruleset describes *conditions* on the result, but not *how* they can be achieved. The rules have to hold after a transformation, but the way in which they are enforced may differ from implementation to implementation. This has immediate consequences for the debugger. For instance, the definition of an "execution step" cannot be easily defined, as it also can differ between the different implementations. Hence, the debugging approach should implementation-agnostic, and it should form a good compromise between representing the declarative specification and the actual execution.

First, we give an overview about related work in Sect. 5.4.1. In Sect. 5.4.2, we introduce a debugging concept for TGGs that provides debugging means on the level of TGG rules, but also reflects the execution strategy of the model transformation algorithm. It supports stopping a transformation using breakpoints, stepwise execution of the matching process and inspection of the state of

[13]This gap between the debugging level and the modeling level can also be found in many industrial applications of domain-specfic modeling approach in general [MV11].

the transformation. Ackermann [Ack10] partially implemented this approach in his Bachelor's thesis. This live-debugging is complemented with static analyses on the TGG ruleset, which allow identifying potential bugs already at design time. An evaluation of our TGG debugging concept can be found in Sect. 6.3.3.

5.4.1 Related Work

Debugging concepts and debuggers exist for different model transformation techniques.

Mannadiar and Vangheluwe [MV11] provide an overview about debugging concepts in domain-specific modeling, also covering model transformations. They translate the ideas of tradition imperative program debugging (like execution control, runtime variables, breakpoints, and stack traces) to model transformation debugging, but limit themselves to a brief summary of existing ideas and concepts.

Dhoolia et al. [DMSS10] present a debugging concept tailored towards model-to-text (M2T) transformations. It is based upon the technique of *dynamic tainting*. The approach associates so-called taint marks with source model elements. These marks are propagated to the output text. In this way, the user can see which source model elements contribute to a text block in the output. With such an approach the user of a transformation is able to find errors that are due to faults in the source model. On the other hand, it is not the best way for finding bugs in the transformation specification itself. However, conveying information about the relationship between source and target model elements is important for many scenarios, including debugging a transformation specification. In a model-to-model transformation, similar information can be gathered using a trace model (like the correspondence graph in TGGs).

Geiger [Gei08] describes an approach how to debug Story Diagrams on model level. Story Diagrams [vDHP$^+$12, Zün05] are an in-place model transformation technique whose graphical syntax is similar to TGGs. Basically, they map code-level constructs to their model-level counterparts. When hitting a breakpoint, the model-level construct (e.g., the node of a story diagram) is highlighted in the concrete syntax editor. While this works well for imperative model transformation techniques (where the transformation definition is more or less an abstract view of the actual transformation execution flow), such a mapping cannot be established for declarative approaches.

Forensic debugging, as described by Hibberd et al. [HLR07], records all atomic operations that a running model transformation performs. After the transformation finishes, it can be replayed using classical debugging facilities like step-wise execution. Furthermore, such an approach allows *reverse debugging*, i.e., stepping backwards in time of the execution. A drawback of forensic debugging is that it requires memory to store all execution steps of the transformation, and that no alterations to a running transformation can be made (e.g., changing an object's property). Furthermore, the transformation engine has to allow recording every atomic operation (rule selection, single matching steps, object and link creations etc.).

In his doctoral thesis, SCHOENBOECK [SKK+10, Sch12] presents an extensive debugging concept for model transformations. It is based upon the novel formalism of Transformation Nets (TNs), which is a DSL on top of Colored Petri Nets (CPNs) [Sch12]. TNs serve as a runtime model for model transformations. SCHOENBOECK uses these TNs to perform static analyses on the transformation specification, e.g., identifying code smells, and to execute the transformation in a step-wise manner or query information during the transformation. The approach allows applying sophisticated debugging means independent from a concrete model transformation approach. Each transformation formalism has to be mapped onto TNs to allow applying this debugging approach; in his thesis, he implemented this mapping for QVT Relations. As a consequence, the transformation engineer has to learn the concept of TNs, besides profound knowledge of the transformation approach. This introduces a gap between the transformation specification and the debugger.

SEIFERT AND KATSCHER [SK08b] present a first debugging concept for TGG-based transformations. They argue that first a common notion of an execution step must be defined, as it is unclear for TGG model transformations. The propose using *debug events*, which are defined as a combination of one or more *subjects* (e.g., a TGG node) and a *context* (e.g., matching). We refine this definition of an execution step for our debugging concept. Although they mention that they plan on implementing and evaluating this concept, to the best of our knowledge, no such study or implementation exists, yet.

Until now, no other TGG implementation besides the TGG INTERPRETER provides debugging support [LAS+14].

5.4.2 Debugging Concept

In this section, we present our concept for TGG debugging. After describing the general idea and the details, we discuss how transformation developers can find different types of bugs using this debugging concept.

When running a transformation that is described in a declarative, rule-based language, the general layout of the different transformation algorithms is similar:

```
Repeat
     1. Check whether a rule is applicable at a certain
        position (‘precondition check’).
     2. If the rule passed the precondition check, apply
        it (‘enforce’).
until no further rules are applicable.
```

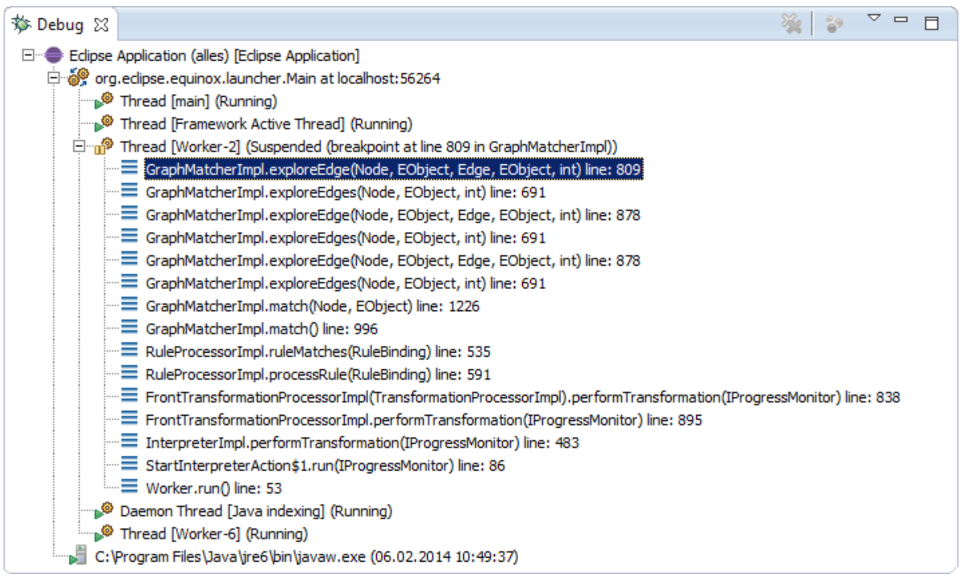
Besides that general layout, algorithms differ in several aspects:

- *Rule scheduling:* The order in which the rules are applied is typically not (always) determined. Thus, the algorithm has to schedule the application of the rules (cf. Sect. 2.2.2).
- *Location determination:* Besides the order of the rules, it may also be undefined where in the model(s) the rule should be checked and applied (cf. Sect. 2.2.2).

- *Matching strategy:* Determining whether a rule is applicable ("precondition check") requires to perform a pattern matching. As pattern matching is a time-complex problem, different heuristics are applied to guide the matching.
- *Application strategy:* Similarly, applying a rule (i.e., creating object/links, or destroying model objects/links while incrementally updating) can be performed in different order.

The debugging concept should appropriately reflect these potential differences between transformation engines. Even if a TGG debugger is only used in combination with a single transformation engine, the transformation engineer may not know or know exactly how this engine behaves. Consequently, the debugger should convey the behavior of the engine to the transformation engineer.

Similar to a debugger of a general purpose programming language, our debugging concepts consists of different layers of granularity. In traditional imperative program debugging, this is called the "call stack" (see Fig. 5.9). This directly reflects the actual program execution, where each executed function call creates a new stack frame.



```
Debug
Eclipse Application (alles) [Eclipse Application]
  org.eclipse.equinox.launcher.Main at localhost:56264
    Thread [main] (Running)
    Thread [Framework Active Thread] (Running)
    Thread [Worker-2] (Suspended (breakpoint at line 809 in GraphMatcherImpl))
      GraphMatcherImpl.exploreEdge(Node, EObject, Edge, EObject, int) line: 809
      GraphMatcherImpl.exploreEdges(Node, EObject, int) line: 691
      GraphMatcherImpl.exploreEdge(Node, EObject, Edge, EObject, int) line: 878
      GraphMatcherImpl.exploreEdges(Node, EObject, int) line: 691
      GraphMatcherImpl.exploreEdge(Node, EObject, Edge, EObject, int) line: 878
      GraphMatcherImpl.exploreEdges(Node, EObject, int) line: 691
      GraphMatcherImpl.match(Node, EObject) line: 1226
      GraphMatcherImpl.match() line: 996
      RuleProcessorImpl.ruleMatches(RuleBinding) line: 535
      RuleProcessorImpl.processRule(RuleBinding) line: 591
      FrontTransformationProcessorImpl(TransformationProcessorImpl).performTransformation(IProgressMonitor) line: 838
      FrontTransformationProcessorImpl.performTransformation(IProgressMonitor) line: 895
      InterpreterImpl.performTransformation(IProgressMonitor) line: 483
      StartInterpreterAction$1.run(IProgressMonitor) line: 86
      Worker.run() line: 53
    Daemon Thread [Java indexing] (Running)
    Thread [Worker-6] (Running)
  C:\Program Files\Java\jre6\bin\javaw.exe (06.02.2014 10:49:37)
```

Figure 5.9: Stack frames in a Java debugger

Although the pattern matching should be conveyed to the transformation engineer, the actual debugging of the matching process should *not* be directly based on the engine's implementation. For instance, the debugger should not show the stack frames of a depth-first search it performs for pattern matching, like in Fig. 5.9, where we see three (recursive) calls within the depth-first search inside the GraphMatcher. Instead, it should focus on the current state of the *pattern* matched so far, as this is the most important information for the transformation engineer.

We developed a hierarchy of transformation execution steps, which we use to structure the debugging events. This hierarchy is not build upon the stack frames within the engine during execution, but on the *logical execution steps* of a transformation. Figure 5.10 shows this hierarchy of transformation execution steps.

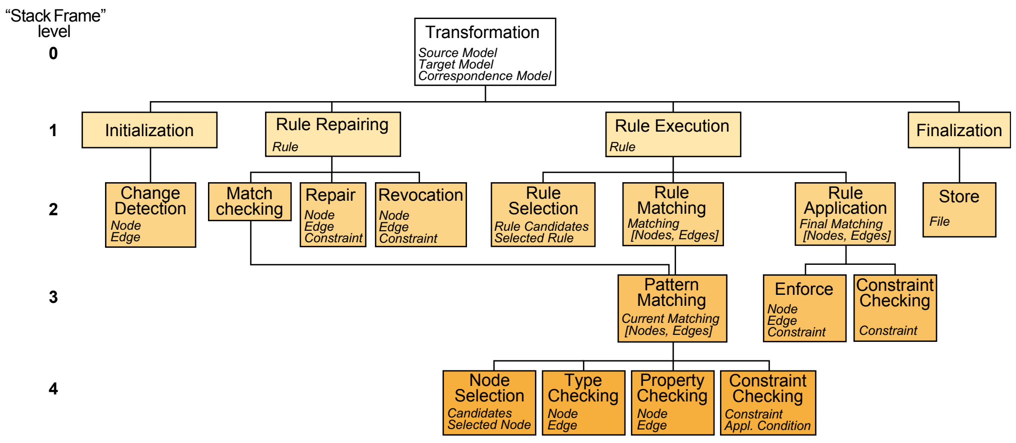


Figure 5.10: Hierarchy of transformation execution steps

On top, there is the complete transformation itself. An execution step on level n consists of other, more fine-grained execution steps on level $n+1$. For instance, a transformation (level 0) consists of the steps initialization, rule repairing, rule execution, and finalization (level 1). These steps again have more detailed substeps.

When debugging an imperative program, there are different sets of variables in each stack frame. Similarly, each execution step of a transformation has different properties associated with it. These properties are also shown in Fig. 5.10. For instance, properties of the transformation are its input, output, and correspondence model. During pattern matching, the properties are the variables of the pattern, i.e., the nodes and edges of the current TGG rule. The values of a variable is the object (or the link) matched to that node (or edge).

All properties of the hierarchy represent the current state of a transformation. A transformation engineer can use these properties to inspect the transformation when it is suspended, similar to classical debuggers.

This set of execution steps can also be extended based on special features of a transformation engine. For instance, when processing changes for an incremental update, an engine can alter the order in which the changes are processed, e.g., such that changes are ordered "top-down" to improve performance [GH09]. In such a case, it is reasonable to add an additional execution step processing ordering below rule repairing. The execution steps shown in Fig. 5.10 can be regarded as a common subset reflecting the capabilities of most engines.

5.4.2.1 Debugging Events

Developers can use breakpoints to pause a running program when a certain *event* occurs. Examples of such events in imperative programming languages are *line hits* (which are the subject of *line breakpoints* that pause the execution when the execution reaches a particular line in the source code), or *exception throws* (which are the subject of *exception breakpoints* that pause the execution when a particular exception is thrown).

Seifert and Katscher [SK08b] provide a first idea of a concept for events in model transformations. They define an event by a *subject* and a *context.* For instance, the subject can be a node, a rule, or a constraint. The context is the situation in which the subject is processed, e.g., pattern matching or rule application. However, they do not elaborate further on this idea.

For our debugging concept, we reuse this subject-context idea and further extend it. It is obvious that not every combination of subject/context is reasonable. For instance, a constraint (= subject) cannot be processed during rule application (= context). However, our hierarchy of execution steps already defines which subjects belong to which contexts. If we think of an *execution step* as the *context*, the *properties* of an execution step are its valid *subjects.*

Consider the pattern matching execution step. Its properties are the nodes and edges already matched. In this context, we would like to allow the transformation developer to interrupt the transformation

- when a particular node/edge is matched onto an object/link, and
- when a particular object/link is matched by a node/edge.

In other words, we want to monitor a particular property for changes, and we want to detect when particular values are to be assigned.

To summarize, we define a model transformation debugging event by

1. a transformation step, e.g., pattern matching,
2. a property, e.g., a node to be matched, and
3. a value, e.g., an object matched to that node.

5.4.2.2 Breakpoints

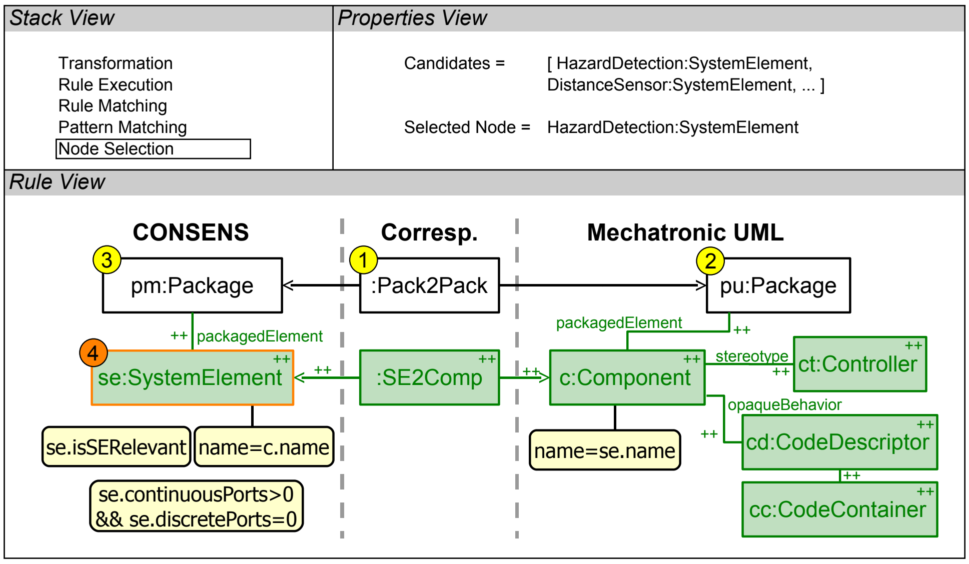
Breakpoints are defined using debugging events. A breakpoint has to define a transformation step in which it may hit. The property and the value are optional. For instance, a breakpoint could hit every time a particular node is to be matched regardless of the object that it is matched on. This provides a transformation engineer with enough flexibility in defining breakpoints, allowing to debug all aspects of a model transformation.

Whenever an event occurs during the execution of a transformation, the transformation engine sends information on occurring events to the debugger. The debugger then checks whether one of the defined breakpoint fits to the current event. In this case, it suspends the running transformation and shows its current state in the debug view, using a visualization of the execution step "stack frames" and a view of the properties.

5.4.2.3 Visualization of the Transformation State

Every state of the transformation can be represented showing the current execution step "stack frames", the respective properties and their values. The debug view shows this information at the top. However, just presenting a simple list of properties may not be intuitive in every transformation step. Therefore, we suggest more specific views for some transformation steps.

Considering intra-rule bugs, the most important view is the pattern matching view. It is based on the concrete syntax of the TGG rules (see Fig. 5.11). When the execution of the pattern matching hits a breakpoint, the lower part of the view shows the TGG rule that is currently matched. Its nodes are annotated with their current matching state. Matched nodes get a yellow circle with a number added to a corner. The number shows the depth of the matching progress, indicating the order in which the matching took place. The node currently processed is additionally shown with a orange border. Unmatched nodes are shown without annotations.


Stack View

Transformation
Rule Execution
Rule Matching
Pattern Matching
Node Selection

Properties View

Candidates = [HazardDetection:SystemElement, DistanceSensor:SystemElement, ...]

Selected Node = HazardDetection:SystemElement

Rule View

Figure 5.11: Mock-up of the Pattern Matching Debug View

For visualizing rule-related information, we use a rule dependency view. This view is shown for the steps rule repairing, rule execution, and rule selection (see Fig. 5.12). It contains a representation of all rules in the ruleset. The rules are connected based on their dependency: The produced part of rule $R2$ may be (partially) contained in the context of rule $R1$. I.e., $R1$ depends on $R2$[14]. These dependencies are computed by a static analysis of the ruleset. More specifically, we search for pairwise overlaps in the produced and context parts of rules. For implementation details on the analysis, we refer to Sect. 6.1.2.

5.4.2.4 Stepwise Execution

In most imperative debuggers, we find the following functions to stepwise execute the program.

[14]A rule $R1$ can depend on several rules $R2, R3, \ldots$. However, this does not mean that all of these rule have to be successfully applied in order to apply $R1$. For instance, if both $R2$ and $R3$ have the same produced part, it is sufficient that only one of them was applied before.

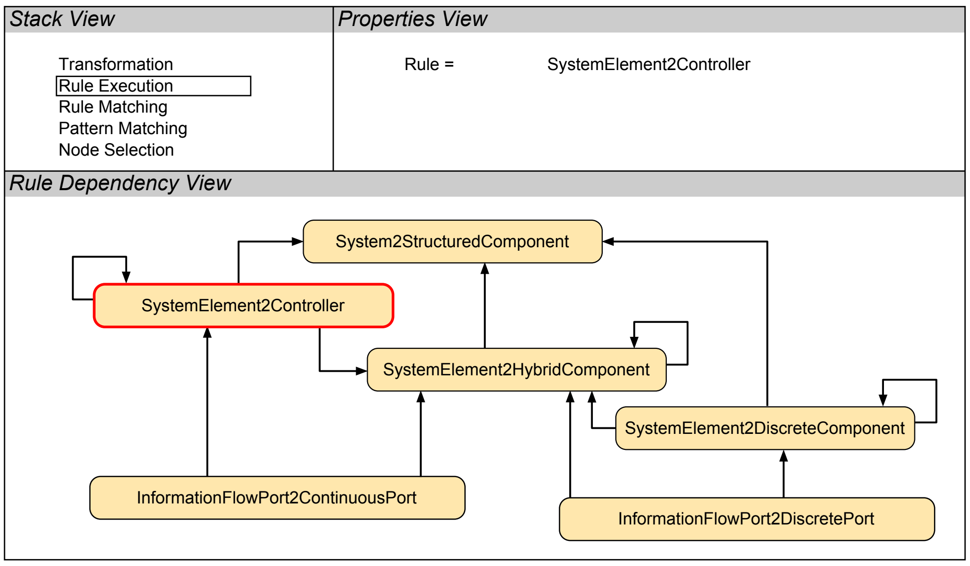


Figure 5.12: Mock-up of the Rule Dependency Debug View

- *Step In*: If the current execution pointer is at a function call, we call this function (opening a new stack frame), step into it and halt at the first instruction of that function. If not, the behavior is like *Step Over*.
- *Step Over*: Execute the current instruction (regardless of its nature) and halt at the next instruction on the current function (stack frame).
- *Step Out*: Resume the execution of the function until it returns to its caller. Halt at the next instruction of the caller function.
- *Resume*: Resume the execution.

Similar to this classical imperative program debugging, we provide means for a stepwise execution of a transformation when it is suspended. Basically, they resemble the behavior of the respective functions of imperative debuggers, transferred to our "stack frame" concept of execution steps (cf. Fig. 5.10). For instance, using the *Step In* function when in pattern matching, we descend to level 4 of the hierarchy, performing the type checking etc. Using *Step Over* instead, the execution steps of level 4 would be executed en bloc, halting again at the next pattern matching step (i.e., a new node or edge to be matched).

Using these stepwise execution facilities, a transformation engineer can easily switch between the hierarchy levels, depending on the current aspect that is debugged.

CHAPTER 6

Realization and Evaluation

Contents

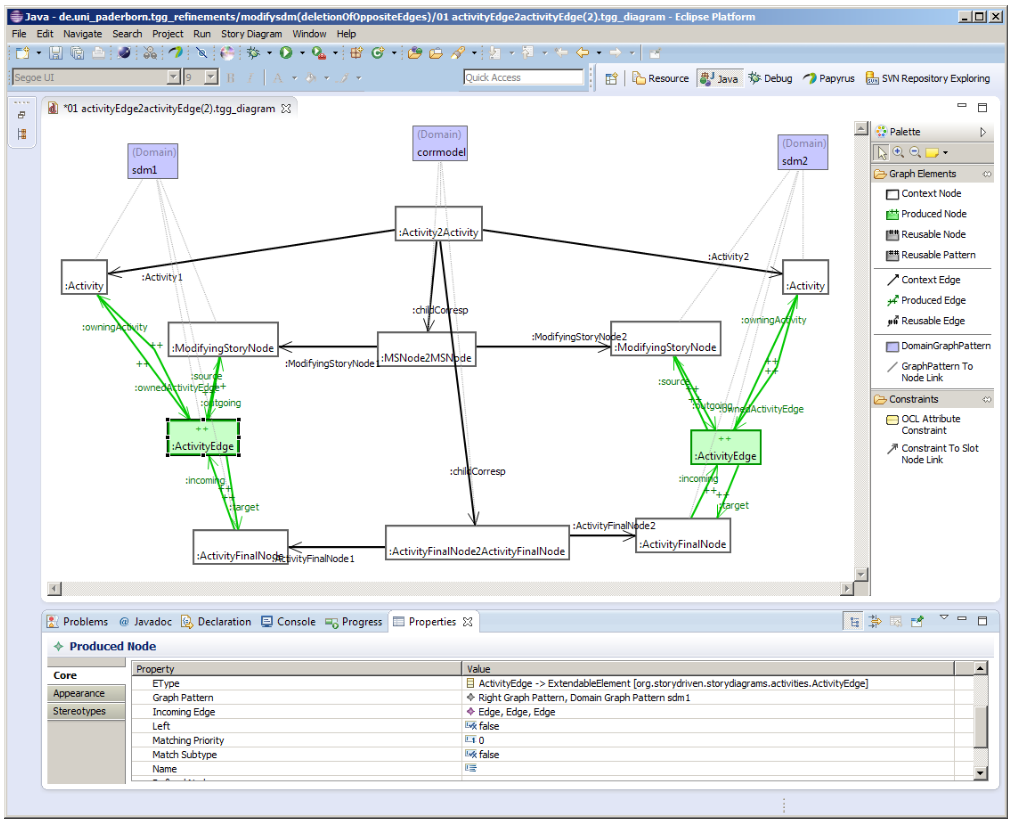
In this chapter we describe how we implemented the approaches described in the previous chapters and present some evaluation results. In Sect. 6.1, we provide details on the software architecture, implementation, and algorithms. Section 6.2 gives an overview about the model transformations implemented in the scope of this thesis. Finally, Sect. 6.3 provides evaluation results, e.g., on the performance of our algorithm.

6.1 Implementation

We implemented our approach using the TGG Interpreter tool suite [UPB14]. The TGG Interpreter is a model transformation engine that uses Eclipse/EMF/ GMF technology. It consists of the model transformation engine ("interpreter"), which performs the actual transformations, and transformation specification tools, which can be used to graphically model the TGG rules.

A first implementation of a TGG INTERPRETER goes back to 2004 [KRW04]. This tool was redesigned in 2006 by GREENYER [Gre06]. The TGG INTERPRETER implementation that was used in this thesis was completely rewritten in 2009, using the experience gained with the first interpreter. In this thesis, we always refer to this latest implementation with the term "TGG INTERPRETER".

Figure 6.1 shows the rule editor that is part of the TGG INTERPRETER tool suite. After executing a transformation, the TGG INTERPRETER shows an overview of the rules that have been checked, revoked and/or applied (see Fig. 6.2).



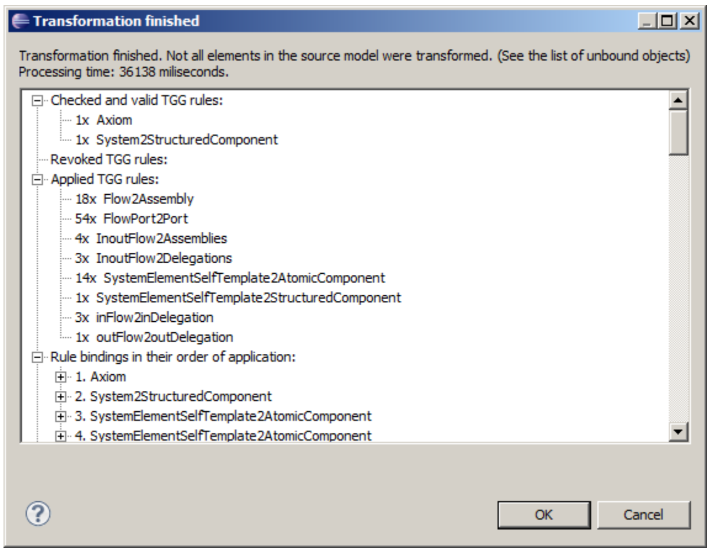
Produced Node

Property	Value
EType	ActivityEdge -> ExtendableElement [org.storydriven.storydiagrams.activities.ActivityEdge]
Graph Pattern	Right Graph Pattern, Domain Graph Pattern sdm1
Incoming Edge	Edge, Edge, Edge
Left	false
Matching Priority	0
Match Subtype	false
Name	

Figure 6.1: Screenshot of the rule editor of the TGG INTERPRETER tool suite

The TGG INTERPRETER tool suite has an architecture that consists of several components. This architecture is shown in Fig. 6.3. The colors denote whether a component was created or modified in the course of this thesis, or whether it already existed before.

In general, each component also forms an Eclipse plug-in. Eclipse plug-ins are singletons, i.e., there exists at most one instance. Interfaces between components are implemented using the *extension point mechanism*. A plug-in can define an *extension point*. Other plug-ins can register at that extension point; they are called *extensions*. Typically, an extension point also specifies



```
Transformation finished

Transformation finished. Not all elements in the source model were transformed. (See the list of unbound objects)
Processing time: 36138 miliseconds.

⊟ Checked and valid TGG rules:
    1x  Axiom
    1x  System2StructuredComponent
  Revoked TGG rules:
⊟ Applied TGG rules:
    18x  Flow2Assembly
    54x  FlowPort2Port
    4x  InoutFlow2Assemblies
    3x  InoutFlow2Delegations
    14x  SystemElementSelfTemplate2AtomicComponent
    1x  SystemElementSelfTemplate2StructuredComponent
    3x  inFlow2inDelegation
    1x  outFlow2outDelegation
⊟ Rule bindings in their order of application:
  ⊞ 1. Axiom
  ⊞ 2. System2StructuredComponent
  ⊞ 3. SystemElementSelfTemplate2AtomicComponent
  ⊞ 4. SystemElementSelfTemplate2AtomicComponent

OK    Cancel
```

Figure 6.2: Incremental update result overview dialog

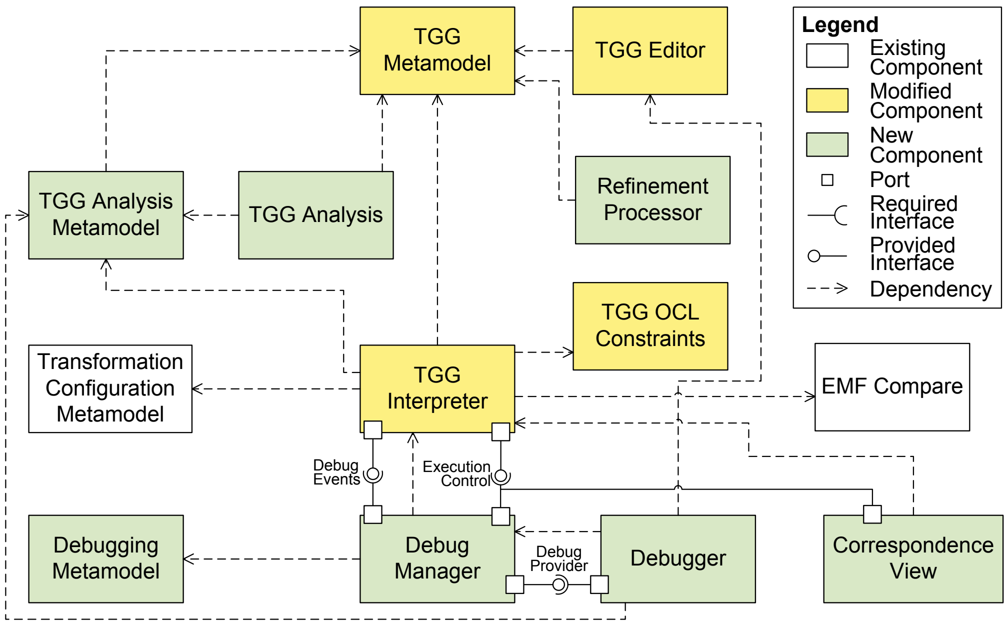


Figure 6.3: Architecture of the TGG Interpreter Tool Suite

an interface that an extension has to fulfill. For instance, there is a Debug Manager component that is extended by the Debugger (cf. Fig. 6.3). The Debug Manager specifies an interface that the Debugger has to implement. Therefore, the Debugger provides this interface, and the Debug Manager requires it.[1]

When designing the architecture, we focused on modularity and tried to minimize dependencies between different components. Especially, we separated the components necessary for executing a transformation (mainly the TGG Interpreter) from the TGG design components (TGG Editor, TGG Analysis, TGG Refinement Processor). When integrating a transformation into a software product, only the TGG Interpreter component and the TGG Metamodel have to be deployed.

In the course of this thesis, the TGG INTERPRETER tool suite has been extended at several points. The implementation of these features are described in the following sections. The novel incremental synchronization algorithm was implemented in the TGG Interpreter. It is described in Sect. 6.1.1. Section 6.1.2 explains the static analyses on the TGG ruleset, which form the TGG Analysis and TGG Analysis Metamodel components. The Correspondence View (described in Sect. 6.1.3 was implemented as a new component. Section 6.1.4 describes rule backtracking facilities. The Refinement Processor integrates refinement rules into an existing (functional) ruleset, as described in Sect. 6.1.5. We incorporated extensions to the syntax into the TGG Editor, mainly the usage of the concrete syntaxes of modeling languages (cf. Sect. 6.1.6). The TGG OCL Constraints component performs the constraint processing, as described in Sect. 6.1.7. Finally, Sect. 6.1.8 describes the implementation of the debugging facilities (Debug Manager, Debugger, Debugging Metamodel).

6.1.1 Incremental Bidirectional Synchronization Algorithm

Figure 6.4 shows the activities of the complete model synchronization algorithm that includes bidirectional, conflict-resolving change propagation. The background colors in this figure denote the different phases of the approach:

1. *Non-conflicting forward propagation:* First all source model changes that are guaranteed to be non-conflicting are propagated to a new version of the target model. We use a slightly modified version of the incremental update algorithm here. This algorithm revokes rule applications that have become invalid due to source model changes. This can create a chain of revocations if further rule applications depend on this revoked application. Therefore, we try to reapply the same or another rule at this position in order to repair the context of dependent rule applications. A revocation is undone if contexts of dependent rule applications that contain target model changes cannot be repaired. See Sect. 4.3.4 for details. Finally, we try applying new rules for elements added or not yet translated (call behavior apply rule). It may happen that we ask the user to choose between different reuse possibilities. Note that these decisions are *not* due to editing conflicts; they help minimizing the amount of information loss due to the information not

[1]Note that there also is an interface that allows the extension to register at the extension point. This is omitted in Fig. 6.3 for brevity.

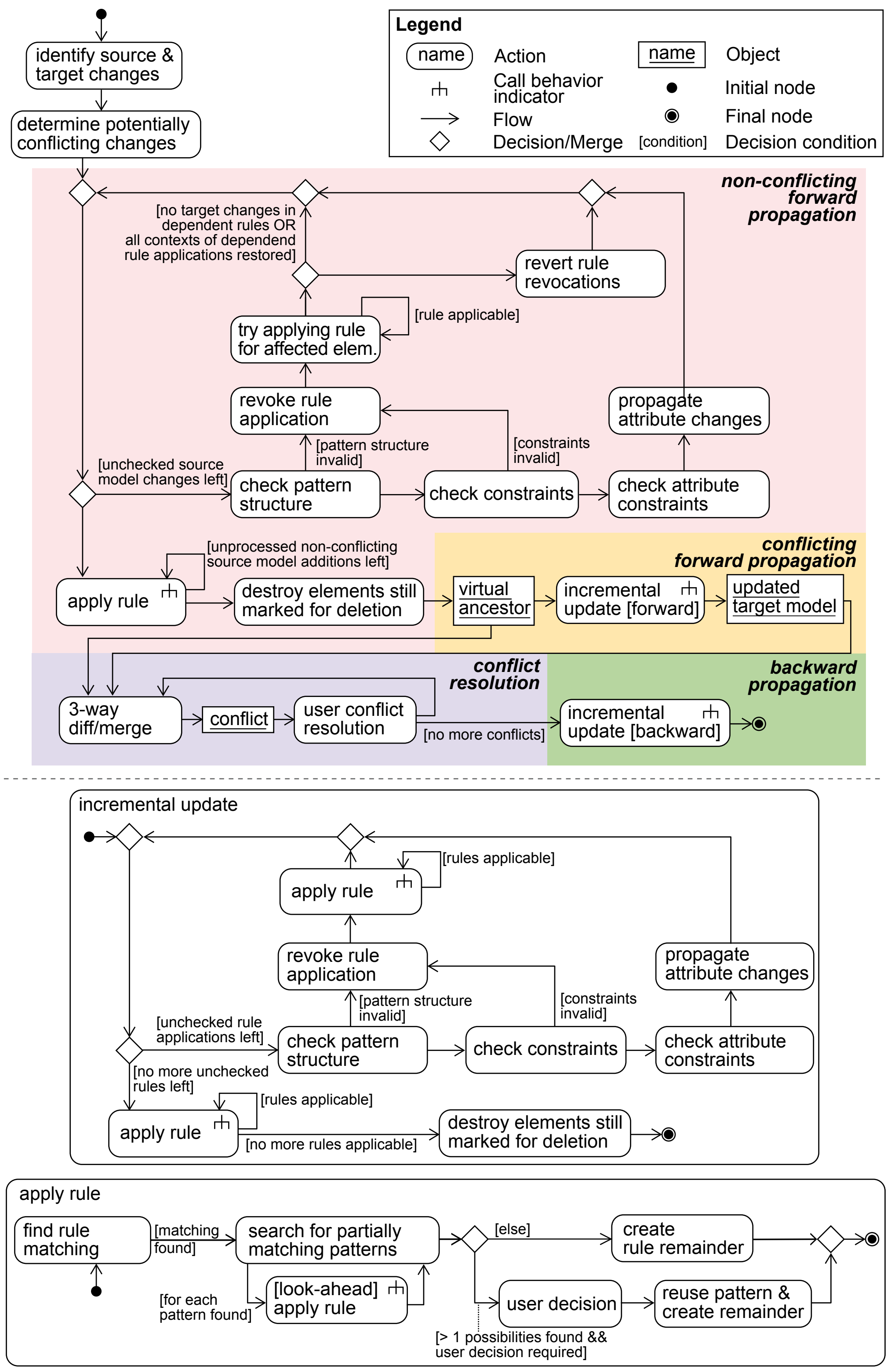

Legend
name Action
Call behavior indicator
Flow
Decision/Merge
name Object
Initial node
Final node
[condition] Decision condition
identify source & target changes
determine potentially conflicting changes
non-conflicting forward propagation
[no target changes in dependent rules OR all contexts of dependend rule applications restored]
revert rule revocations
try applying rule for affected elem.
[rule applicable]
revoke rule application
propagate attribute changes
[pattern structure invalid]
[constraints invalid]
[unchecked source model changes left]
check pattern structure
check constraints
check attribute constraints
[unprocessed non-conflicting source model additions left]
conflicting forward propagation
apply rule
destroy elements still marked for deletion
virtual ancestor
incremental update [forward]
updated target model
conflict resolution
backward propagation
3-way diff/merge
conflict
user conflict resolution
[no more conflicts]
incremental update [backward]
incremental update
apply rule
[rules applicable]
revoke rule application
propagate attribute changes
[pattern structure invalid]
[constraints invalid]
[unchecked rule applications left]
check pattern structure
check constraints
check attribute constraints
[no more unchecked rules left]
[rules applicable]
apply rule
[no more rules applicable]
destroy elements still marked for deletion
apply rule
find rule matching
[matching found]
search for partially matching patterns
[else]
create rule remainder
[for each pattern found]
[look-ahead] apply rule
user decision
reuse pattern & create remainder
[> 1 possibilities found && user decision required]

Figure 6.4: Activity diagram of the complete (bidirectional, conflict-resolving) model synchronization

subject to the transformation. The result of this phase is a virtual common ancestor that already contains non-conflicting, merged changes.

2. *Conflicting forward propagation:* For the remaining changes, we cannot be sure that they do not cause a conflict. As a second step, we propagate these remaining changes to a new version of the target model, using the virtual common ancestor as a basis for the propagation. To do so, we call the behavior incremental update.
3. *Conflict resolution:* Third, we run an external 3-way diff/merge tool. For each conflict that cannot be solved automatically, we let the user decide how to deal with the conflict. In our implementation, we use EMF Compare [Ecl13] as model diff/merge tool. It takes three model files as input: the virtual common ancestor (result of the first phase), the user-edited target model (omitted in Fig. 6.4), and the target model updated with the user edits from the source model (result of the second phase). We call EMF Compare in order to automatically merge the changes. In case that there are conflicts that cannot be solved automatically by EMF Compare, we open the EMF Compare user interface so that the user is able to resolve the remaining conflicts manually.
4. *Backward propagation:* Once all conflicts have been resolved, we run a backwards incremental transformation. As all potential conflicts have been resolved previously, no conflict resolution is required in this step. However, it may still be necessary to involve the user in cases where we cannot automatically decide between different reuse possibilities.

Next, we explain implementation details of different activities.

When using TGGs for model-to-model transformations, we try to find a way to derive the existing source model using the TGG rules. We create the correspondence and target model parts according to this derivation. TGGs are constructive graph grammars, i.e., they only produce new elements. Thus, when searching for a derivation, we use the existing source model elements to guide the search. We emulate applying a rule by finding the source produced parts of that rule in the source model, and remembering which source model elements have been (virtually) created by that rule. The different TGG implementations chose varying ways to implement this "bookkeeping".

Some TGG transformation approaches (like Wagner et al. [Wag09, KW07], MDELab/MoTE[2] [GH09]) do not explicitly store that a produced model element has been covered by a rule. Instead, they require that every produced node must be referenced by a correspondence link. When a rule is applied, this rule application creates one or more correspondence nodes that link to the model elements that have been already translated. The advantage of such an approach is that it does not require additional space to store the "is already translated" property of objects. A major drawback is that we cannot reference links. Consequently, such an approach does not allow rules that contain only edges in the produced pattern: Such a rule would be applicable infinitively often, because it does not "consume" any source model elements.

[2] `www.mdelab.de/mote`

The TGG Interpreter and eMoflon[3] [KLKS10, LAVS12a] explicitly store which elements have been translated. While eMoflon simply stores a "is already translated" flag for all source model elements (including links), the TGG Interpreter explicitly store so-called *bindings*. A binding contains which model element was matched by which TGG node/edge. While the drawback is the increased required storage space, its advantage is that we know which rule was applied to which model elements. When checking if a rule still holds during incremental update, we do not have to perform a pattern matching again to find the applied rule. Instead, we just have to check whether all rule elements still exist and all constraints hold. The time complexity of this check, therefore, is linear to the size of the input model, important especially when incrementally updating huge models.

It is also conceivable to use deltas[4] as input. Using these deltas, we know which parts of the models to check for consistency, because only rule applications at changed parts could have become invalid. This eliminates the need for checking all rule applications. However, not all design tools support deltas. Furthermore, deltas can only be generated easily during single-user editing sessions. Maintaining correct deltas when multiple users edit a model requires further effort and is typically not supported by existing collaboration/revision control tools. Thus, we decided not to implement this in our TGG Interpreter, yet.

6.1.2 Static Analyses

We implemented two static analyses on TGG rulesets. The first improves the evaluation of constraints, the latter yields information on the relationship between different rules. First, we analyze OCL constraints for the information they use, in order to find the best point in time for their evaluation during rule matching. Second, we use a *critical pair analysis* (CPA) to identify potentially conflicting rules, i.e., rules whose application prevents the application of other rules. A dependency analysis shows the preconditions of a rule, i.e., which rule(s) can form the context of this rule.

6.1.2.1 OCL Constraints

A node's name can be used as a OCL variable for the object that is matched to the node. This allows accessing the object's properties and references. In this way, the transformation engineer can specify application conditions and/or invariant constraints on the actual matched objects.

A naive solution for constraint processing is to perform a complete pattern matching, which only evaluates the object structure in the host models, and then to evaluate the constraints afterwards. If the constraint evaluates to `false` eventually, the pattern matcher has to try finding another matching. The later the decision that lead to the non-fitting matching, the better, because this

[3]`www.emoflon.org`

[4]A delta describes a single change in a model, i.e., at which positions and how a model has changed. A set of deltas describes all changes between two model versions, comparable to a *diff* in a text file.

reduces the necessary time for finding the new matching. However, we do not know at which point to take another matching decision.

The opposite way is to evaluate the constraints *every time* we enlarge the matching (by a new object-node match). This obviously can make a constraint non-processable, because not all variables (= nodes) have a value (= object), yet. Thus, evaluating an unprocessable constraint wastes time.

As solution, we analyze all constraints to find out which nodes they reference. We then evaluate a constraint not before all required nodes have been matched. This eliminates the time wasted on futile evaluation attempts, while noticing a matching decision that is invalid due to the constraint as early as possible.

The information from the constraint analysis is also used in computing rule dependencies.

6.1.2.2 Rule Dependencies

Rule conflicts can be identified using *critical pair analysis* (CPA) [HEGO10, HEOG10]. CPA has first been introduced for term rewriting, and later was adapted for graph transformations. The underlying idea of CPA is to find a minimal example that potentially forms a conflict [MTR05, LEO08].

Because TGGs only contain non-deleting graph transformation rules, CPA becomes simpler than in the general case: We only have to consider a) overlaps in the produced part of two rules, and b) rules producing elements that contradict a NAC of a rule.

Considering overlaps in the produced parts, not every overlap constitutes a conflict: The context of the overlapping rules can make their applicability disjoint. However, we require information from the metamodel to decide this. Figure 6.5 shows an example of such an assumed critical pair. There is a node of type SystemElement in both TGG rules. Thus, an object of this type could potentially be translated by both rules. However, if we also consider the metamodel, we see that the class SystemElement has the same opposite role parent for both produced references packagedElement and containedElement. An object of type SystemElement can either be packaged in a Package or contained in another SystemElement. Thus, these rules are not in conflict.

Considering rule applications that invalidate the NAC of another rule, such an analysis is more difficult. The problem here is that we use OCL to express NACs. OCL is a powerful language allowing highly complex conditions. It is, in general, undecidable. Especially problematic is that OCL constraints can navigate out of the scope of a TGG rule; for instance, an OCL constraint can refer to elements "far away" from the current rule context. At the drawback of not identifying some potential conflicts, we restrict the analysis of OCL constraints to a few frequent cases (e.g., a forbidden link) and to the scope of the TGG rule. However, we allow manual changes to the identified rule conflicts by the transformation engineer. In this way, the engineer can manually specify potentially conflicting rules (or remove a falsely identified conflict when a constraint resolves a rule conflict).

We compute the preconditions for a rule similarly. For a rule to be applied, its context must be bound, i.e., another rule application must have previously

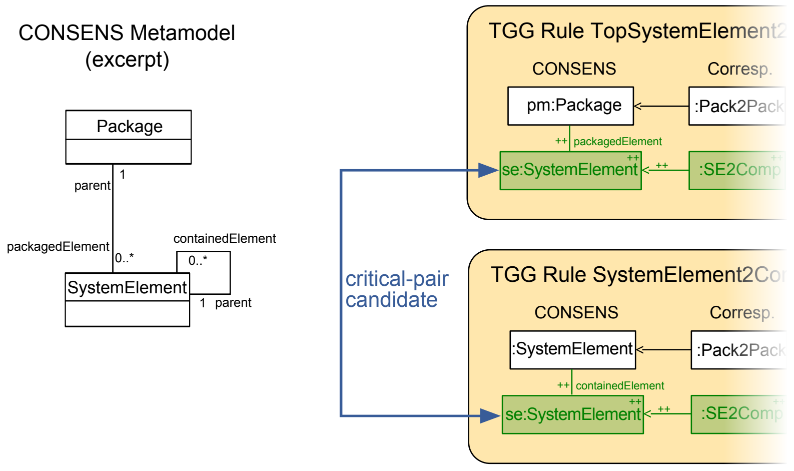
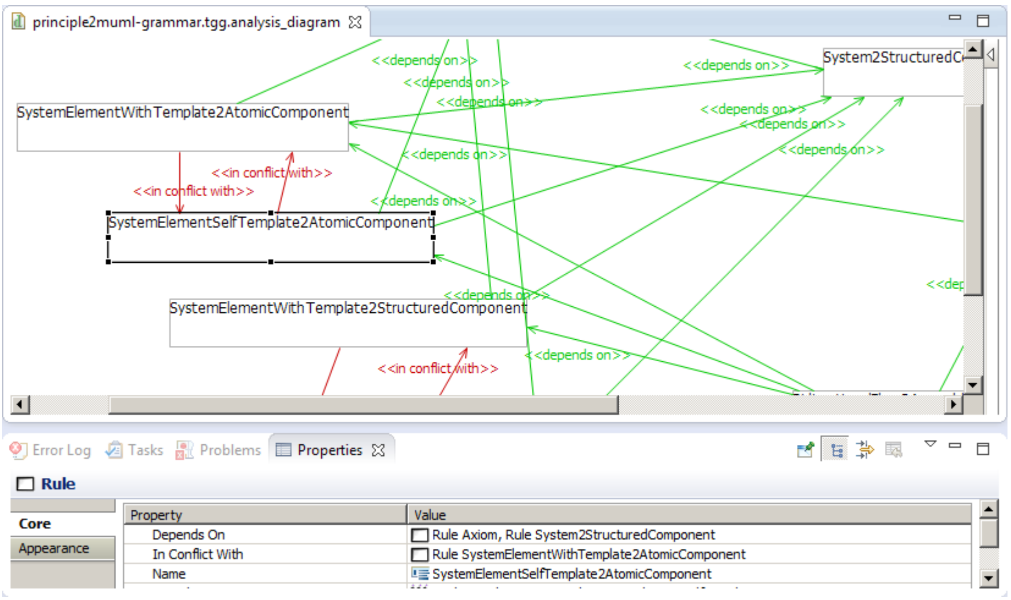


Figure 6.5: Critical pair analysis: Example for a critical pair that does not constitute a rule conflict

produced objects that are now to be matched by the context.

Figure 6.6 show a screenshot of the rule dependency visualization. You can also see “false positive” conflicts (due to OCL constraint that have not been considered in the analysis) between the rules that translate different of system elements.


principle2muml-grammar.tgg.analysis_diagram
SystemElementWithTemplate2AtomicComponent
SystemElementSelfTemplate2AtomicComponent
SystemElementWithTemplate2StructuredComponent
System2StructuredC
<<depends on>>
<<in conflict with>>
Error Log
Tasks
Problems
Properties
Rule
Core
Appearance

Property	Value
Depends On	Rule Axiom, Rule System2StructuredComponent
In Conflict With	Rule SystemElementWithTemplate2AtomicComponent
Name	SystemElementSelfTemplate2AtomicComponent

Figure 6.6: Screenshot of the rule dependency analysis

6.1.3 Correspondence View

To improve the change propagation, we have developed the *Correspondence View*, which was implemented in the course of the project group

SafeBots II [AGL+12]. It shows both models that we want to synchronize in two separate windows using their respective concrete syntax editors (cf. Fig. 6.7).

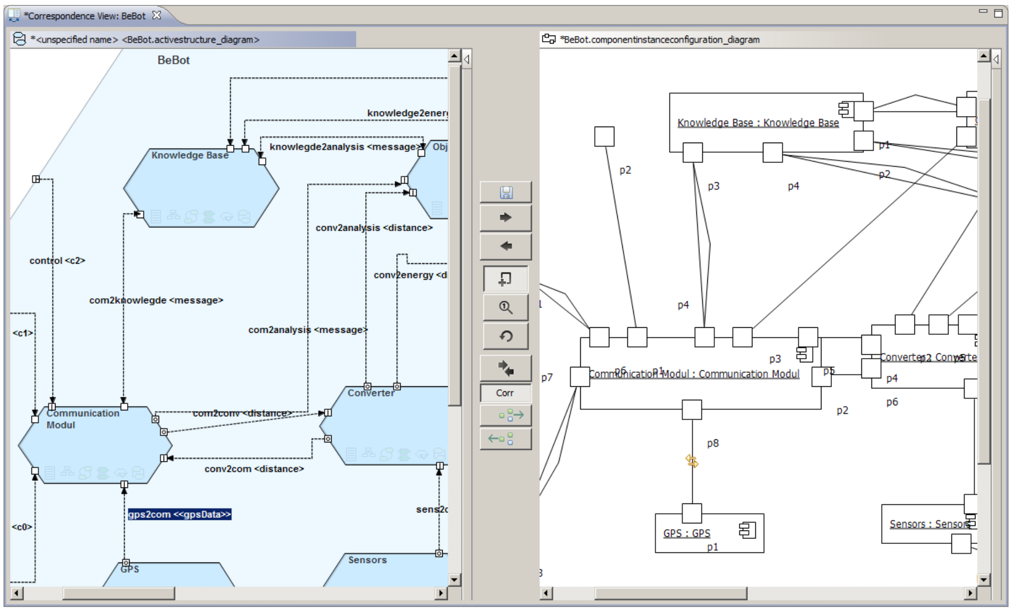


Figure 6.7: Screenshot of the correspondence view showing a CONSENS active structure and a MechatronicUML component diagram

The buttons in the center between the two editors execute different functions. The topmost button saves both models. The next two buttons trigger a transformation/incremental update in the respective direction. The mode of the transformation is set be the following three buttons: incremental update, update only the selected elements, and complete (re)transformation.

The next button shows overlay highlights on the elements in the editors that indicate which elements have been added or modified in the last transformation run. The “Corr” button enables an overlay that shows the corresponding element(s) in the other model for the currently selected element(s). In Fig. 6.1, the information flow gps2com between the GPS and the Communication Module system elements is selected, and you see the corresponding connector highlighted with yellow arrows in the component diagram on the right.

The lower two buttons trigger a layout propagation. When initially generating a discipline-specific model, it is reasonable to layout it similar to the discipline-spanning system model. This makes it easier for engineers that work with different, but related models to communicate, because they identify corresponding elements more quickly.

The layout algorithm is based on the concept of von Pilgrim [vP07]. It consists of three phases:

1. *Clustering*: For each node in the source diagram, we identify its source element. Using information from the correspondence graph, we can identify the corresponding elements in the target model. There may be several elements in the target model corresponding to a single element in the source

model (e.g., this often happens in transformations from an abstract to a more concrete model). For each node, we create a *cluster*. This cluster contains all corresponding target elements. We place each cluster at a location equal to position of the source node.
2. *Arranging and scaling of internal nodes:* If a cluster contains more than one target model elements, we have to internally arrange the cluster elements. As we do not have layout information from the source diagram, we simply arrange them equidistantly on a circle. Next, we scale the circle such that no cluster element overlaps another.
3. *Diagram scaling:* Finally, we scale the complete diagram such that no overlaps occur between nodes.

Lines (connections) between nodes are not explicitly considered; they occur as direct connections between the respective nodes in the target diagram. This can lead to lines that unnecessarily cross each other. As a possible extension, we could include an automatic layout algorithm for the lines, or we could also propagate the layout information of lines from the source diagram in cases where manually defined bend points exist.

6.1.4 Rule Backtracking/Look-Ahead

There may be more than one rule that is applicable to translate a certain source model element. This is called a *rule conflict*[5]. Each time such a rule conflict occurs, the TGG INTERPRETER has to decide which rule to apply. Other approaches like MDELab/MoTE deal with this problem by explicitly forbidding TGGs with rule conflicts [GHL14], so there will never be such decisions. However, we would like to support also non-functional rulesets.

Depending on the rule that is applied the final transformation result varies, because the rules produce different correspondence/target model elements. However, situations exist where the final result does not depend on the applied rule. Assume that rules A and B are in conflict. Applying rule A makes rule C applicable, and applying rule B makes D applicable. If the application of A, C leads to the same transformation result as B, D, we say that the rules are *confluent*.

The critical pair analysis (CPA, cf. Sect. 6.1.2) yields information on potentially conflicting rules. Using this information, the TGG INTERPRETER can distinguish between a) rules that do not have an alternative, and b) potentially conflicting rules. The latter requires a heuristics-based or user decision to select the rule to be applied.

When executing a transformation, it is reasonable to first only try applying rules that do not have conflicting rules. For potentially conflicting rules, we check whether any of the alternative, conflicting rules is also applicable for parts of or the complete identified matching. The CPA identifies which nodes/edges in the two conflicting rules constitute a critical pair. For each critical pair, we rerun the pattern matching, starting with the critical pair's other node/edge. If

[5]The same holds for reusable nodes and edges: Such a reusable node or edge can be interpreted as context or produced node. In other words, each interpretation of a reusable node/edge creates a new rule, and all of these rules may be in conflict with each other.

the matching is successful, this is in fact a conflict: Both rules can be applied, and both rules use at least one identical object or link. In this case, the transformation is likely to be non-functional, and we have to decide which rule to apply.

The TGG Interpreter computes the result for *all* conflicting rules, i.e., we create a branching point in the rule application list, and for each conflicting rule, we create a separate branch. Furthermore, the TGG Interpreter immediately proceeds and applies further rules for all of the branches. An adjustable parameter *look-ahead* defines how far it should advance, i.e. how many rules should be applied.

This approach may sort out confluent situations or incomplete transformation results. Figure 6.8 shows an exemplary rule application search tree. Rules 1, 2, and 3 are applicable at the branching point. When performing a look-ahead, applying rule 2 or 3 leads to the same state after a consecutive application of rule 4 or 5 (confluency). Applying rule 1 leads to an incomplete transformation result, i.e., not all source model elements can be translated. In this case, we can avoid asking the user with a look-ahead of 2.

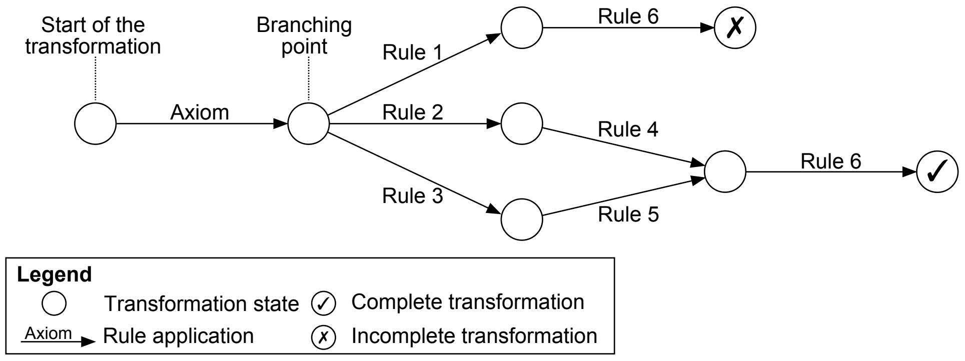


Figure 6.8: Exemplary rule application tree

Technically, we implement this using the EMF Model Transactions framework. A *transaction* encapsulates each rule application (the creation of nodes and edges, plus the binding). A transaction can also easily be undone/redone. When the TGG Interpreter finished computing one branch, we undo the rule applications of this branch to get back to the branching point. From there on, we can compute the next branch. However, allowing backtracking is computationally complex. Furthermore, altering the models using the Model Transactions framework introduces further overhead. Therefore, rule backtracking/look-ahead is not active by default.

6.1.5 Abstraction and Concretion Relations

As described in Sect. 4.2, we want to define a functional transformation definition I, and then combine it with a set of refinement rules $op_1, op_2, ...$ to automatically generate the actual consistency relation R.

The initial transformation function I and the consistency relation R is defined using TGGs. For the definition of the refinement rule, we chose Story Diagrams, a graphical in-place model transformation language [Zün05, vDHP+12]. One reason for us to choose Story Diagram is that their syntax is very similar to the compact syntax of TGG rules: Context nodes and edges are in black and white, and produced parts have a green outline. In contrast to TGGs, Story Diagrams also allow the deletion of elements by *destroy* nodes and edges, denoted in red. OCL is also used as constraint language in Story Diagrams. This simplifies the conversion of story diagram constraints to TGG constraints, which is necessary for generating the consistency relation R. Nevertheless, other model transformation languages like, for instance, Henshin[6] are also suitable to define refinement rules.

Figure 6.9 shows the technical concept of our solution. We assume that the target language B is the more concrete language, so the refinement rules are defined for that language. The story diagrams define how a concrete instance of the target model can be changed without consequences for the source model. The story diagrams are typed by the target metamodel ("typed by" relation in Fig. 6.9).

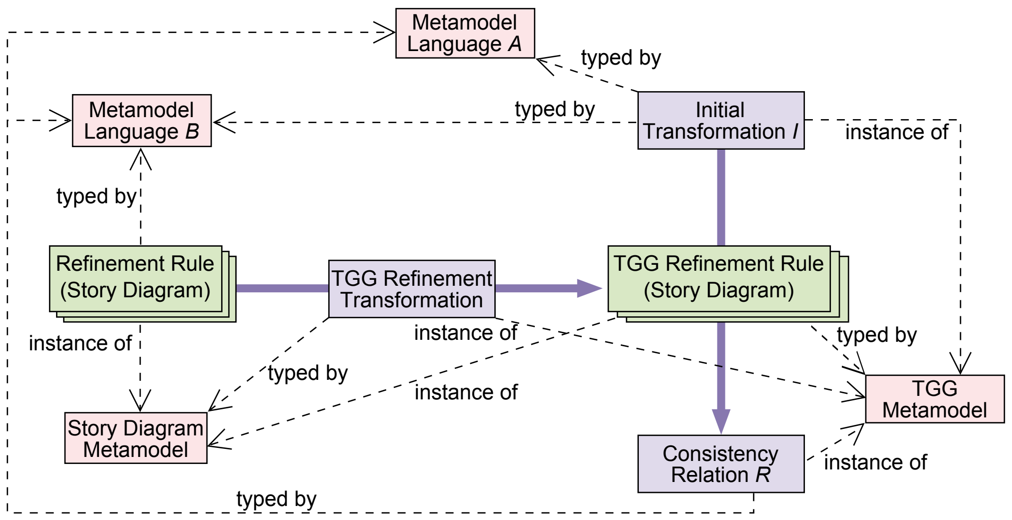


Figure 6.9: Technical concept for generating the consistency relation R from the initial transformation I and the refinement rules

To include the refinement information, we want to apply these rules to our TGG ruleset. To do so, we try to apply the refinement rules to the produced target pattern of all TGG rules in the initial transformation ruleset I. The result will be a TGG rule that produces the right-hand side of the refinement rule in the target domain (which is the refined state of the target model). We do not want to modify the original rule, because it is still part of the consistency relation. Therefore, we do not alter existing TGG rules, but create a copy of the TGG rule in which the refinement rule was applied.

The story diagrams have to be converted such that they work on the TGG metamodel. This conversion is performed by a model transformation, which is,

[6]Website: `https://www.eclipse.org/henshin/`

again, implemented using TGGs (“TGG Refinement Transformation”). Generally, this transformation performs a 1-to-1 mapping for all constructs of the Story Diagram language. When dealing with types, some of the rules implement a more complex mapping. The rules of this transformation can be found in Appendix B.

The result of applying the modified story diagrams to the initial ruleset I is the consistency relation R. It contains all refinements defined by the refinement rules. A model synchronization based on this consistency relation recognizes refinements to the target model by a matching TGG rule for this refinement.

Right now, the implementation does not support cases where a refinement rule is so complex that it affects several TGG rules at once. However, this can be achieved using a trick. The types of nodes in the refinement rule (story diagram) identify potentially affected TGG rules. Using the information from the rule dependency analysis (cf. Sect. 6.1.2.2), we can create amalgamated rules that consist of a combination of single TGG rules by “gluing” them together at nodes of the same type. Then, the TGG refinement rules can be applied to these amalgamated rules as before.

6.1.6 TGG Syntax Extensions

We developed a concept for incorporating the concrete syntax of the referenced modeling languages into the TGG rule editor (see Sect. 5.3 for details). Support for concrete syntax TGG editing has been implemented in the course of the project group SafeBots II [AGL+12].

Figure 6.10 shows an example of the TGG rule editor showing the rule SystemElement2Component of the transformation from CONSENS to MechatronicUML. At the bottom, the rule designer can switch between the classic, object-diagram-style rule editor and the new editor using the concrete syntax elements from the respective modeling languages.

The *Graphical Editing Framework* (GEF) is the de-facto standard for visual editors for modeling languages within Eclipse. When using the *Eclipse Modeling Framework* (EMF), the *Graphical Modeling Framework* (GMF) provides a convenient, model-driven development approach for defining the graphical syntax of a modeling language and generating a graphical editor for it. GMF is build upon GEF, i.e., a generated GMF editor internally uses GEF for displaying the graphical syntax. The TGG Interpreter and the TGG rulesets use EMF as the underlying (meta-)modeling technology. Consequently, our TGG rule editor uses GMF/GEF.

Our implementation only supports GEF-based editors to be incorporated into the TGG rule editor. Although other frameworks or even manually implemented graphical editors could theoretically be used within our concept, it is technically difficult to do so. GEF provides a uniform way of accessing the visual notations of elements; this may not be the case for other frameworks/implementations. Furthermore, the way in which other editor frameworks draw graphical elements may be incompatible with GEF.

GEF-based editors store the visual representation with different means. For instance, there could be a class that contains the complete representation for

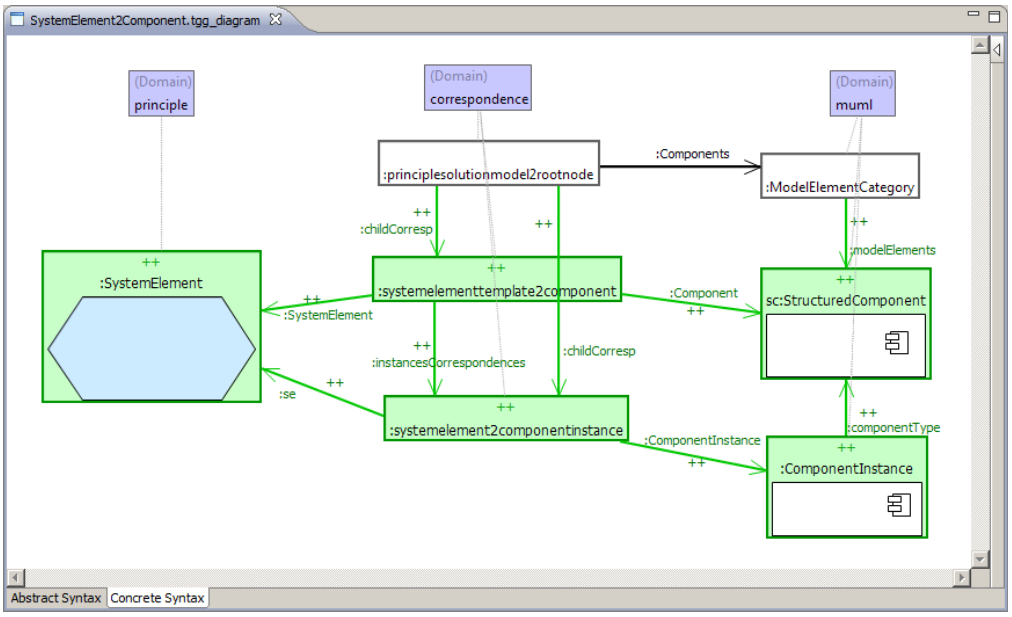


Figure 6.10: Screenshot of the TGG rule editor using the concrete syntax of the modeling languages CONSENS (left) and MechatronicUML (right) [AGL+12]

each modeling construct; it is then instantiated once for each instance of the modeling construct. The representation could also be manually constructed using anonymous classes; in such a case, there is no way of accessing this class from outside the editor. Therefore, the only uniform way of accessing the concrete syntax is to create a real instance of a class and use the `getFigure()` method of its GEF edit part.

For rendering the concrete-syntax TGG rule, we proceed as follows. For each TGG node in the TGG rule, we:

1. identify the corresponding metamodel type (class) of the respective modeling language,
2. create an instance of that class,
3. use the corresponding visual editor (more specifically: the view provider) for this modeling language to create the visual representation,
4. create an edit part for the instance using the modeling language's edit part provider,
5. append the edit part's figure (`getFigure()`) to the TGG node's GMF runtime notation node.

In this way, we keep the outer visual representation of a TGG node (the b/w or green boxes), but it now contains the concrete syntax element of the modeling language. For model elements that do not have a visual representation in its visual editor, we fall back to the standard abstract-syntax TGG notation. In Fig. 6.10, the correspondence nodes and :ModelElementCategory node are using this abstract-syntax notation.

This approach is as general as possible, as it typically does only require minimal manual implementations/modifications or no manual implementation at all. However, some GEF editors that we used were not properly implemented and/or registered according to the Eclipse plug-in mechanisms. We implemented additional searches for view providers and edit part providers to circumvent this problem.

One general, conceptual problem remains: The visual representation of an object may depend not only on its class, but also on its properties or even on other objects that it references. The object's edit part accesses this information using EMF methods when computing the visual representation. We can use the TGG rule to create objects and links between them accordingly[7]; in this way these "emulated" objects could be accessed similar to a "real" model. However, if the edit part accesses a property or object *not* part of the TGG rule's graph, errors like null pointer exceptions can occur. This will prevent the display of a visual representation.

Conceptually solving this problem is difficult. We could simply create those non-existing elements or set a default attribute value to prevent the error. However, the syntax then will probably look different from what is specified in the TGG. Therefore, we fall back to the standard (non-concrete-syntax) TGG notation in such a case.

6.1.7 Constraints and Application Conditions

Next, we describe how we implemented invariant constraints and application conditions. We explain the selected language to specify constraint. Last, we describe how we implemented that application conditions do not invalidate the equivalence between simultaneous production and transformation semantics, i.e., the correctness of the transformation.

6.1.7.1 Specification Language

In our TGG Interpreter, we implemented both application conditions as well as constraints using the Object Constraint Language (OCL) [OMG12]. OCL is a side-effect free language, i.e., it just evaluates constraints, but cannot modify objects. This is important because of the declarative specification of TGGs: a TGG only specifies relations that must hold after the transformation run, but does not define *how* this is achieved. In other words, there are no explicit operational semantics attached to a TGG.[8] It is up to the control algorithm when to evaluate or to enforce such application conditions/constraints.

Allowing side-effects on constraint evaluation would violate the declarative character of a TGG. Using a side-effect free language as constraint language guarantees that TGGs remain declarative.

[7]A value can be assigned to a property when there exists a eligible attribute value constraint.

[8]In fact, different tools use different application and control strategies to run a transformation.In general, a transformation engine has to keep track of model elements that have been already translated (so-called "bookkeeping"). The engines differ in how they search for *non-translated elements* to continue the transformation process. Whereas, for instance, eMoflon [AKRS06] uses its matching algorithm to search for such elements [KLKS10], the TGG Interpreter explicitly maintains a list of non-translated elements that may be translated next (a so-called "front") [HLG+13, UPB14].

Details of the concept of constraints and application conditions using OCL, especially how we deal with the semantics of application conditions, can be found in Sect. 5.1.

6.1.7.2 Ensuring Correctness with Application Conditions

As described in Sect. 5.1.3, we also have to emulate the simultaneous production semantics when evaluating application conditions in order to keep the transformation *correct* (cf. Sect. 2.4.2.2). In particular, the parts of the source model that are not translated yet must be ignored when evaluating an application condition, because they would not exist in the simultaneous production case. Otherwise, the semantics of a transformation would differ from the semantics of a simultaneous production, potentially resulting in an incorrect transformation. We implemented this by three helper operations in OCL, `isBound()`, `getIfBound()`, and `selectBound()`.

`isBound()` is defined for all types (`OclAny`) and returns a boolean value that indicates whether this object is already bound by the transformation, i.e., whether it was already translated by a produced node. For properties, it always returns `true`. When determining the bound state of an object, we only consider successfully applied rules, but not the rule that is currently matched. `getIfBound()` is defined for all types (`OclAny`). It returns the object if it was already bound, and `null` otherwise. For this, it uses the `isBound()` operation. We defined it directly in OCL:

```
context OclAny
    def: getIfBound() : OclAny =
        if isBound() then self else null endif
```

`selectBound()` is defined for `Collections` of objects. It filters out those elements that are not yet bound. It also uses the `isBound()` operation, and is defined in OCL:

```
context Collection:
    def: selectBound() : Boolean = select(isBound())
```

Using this helper operations, we can automatically rewrite OCL application conditions such that they do not refer to elements not yet bound. We rewrite OCL application conditions as follows. Every occurrence of “`->`” is replaced by “`->selectBound()->`”, and every “`.`” is replaced by “`.getIfBound().`”. Let us consider the example from Sect. 5.1.3: `list->sizeOf() = 0`. With this application condition, we want to ensure that the containing rule is only applied when no elements of this list have been translated, yet. It is rewritten to `list->selectBound()->sizeOf() = 0`. In this form, only bound elements are considered when computing the size of the list. Therefore, the application condition only holds when translating an element from this list for the first time (which could be any element of the list, not just the first element in an ordered list).

6.1.8 TGG Debugging

The TGG Interpreter tool suite can be applied in different scenarios. Its component/plug-in architecture allows users to individually define which parts of the tool suite are required for their purpose. Transformation engineers typically use the complete tool suite for transformation development; this includes debugging facilities. When applied only for transformation execution, no debugging plug-ins are necessary. In fact, dispatching debugging events in such a case will decrease performance.

Therefore, the TGG Interpreter tool suite is designed such that there is no dependency from the TGG Interpreter to the debugging plug-ins. Via its interfaces Execution Control Debug Events, the TGG Interpreter allows a Debug Manager to plug in. Thereby the Debug Manager receives debug events from the interpreter and can pause or restart the transformation execution.

However, besides that interfaces, there is also no explicit dependency from the Debug Manager to the TGG Interpreter.[9] Other model transformation engines that implement these interfaces can also use the Debug Manager. Another component, the Debugging Metamodel, implements concepts like breakpoints and the execution step hierarchy (cf. Sect. 5.4). The Debug Manager is also responsible for managing breakpoints. Whenever a new debug event is received from the TGG Interpreter, it checks whether this event hits a breakpoint. Debuggers, which attach to the Debug Manager, implement the user interface for the debugging, e.g., showing the current matching state using annotations in the TGG Editor or providing actions for setting and removing breakpoints.

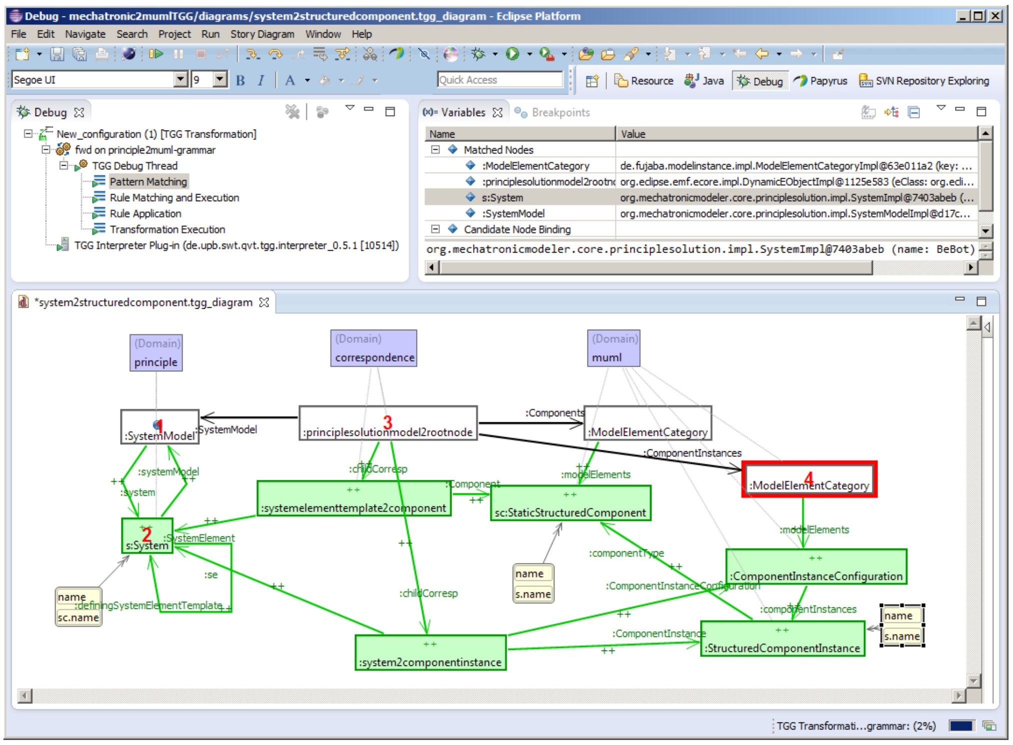
The debugging architecture generally follows the *model-view-controller paradigm*. The Debugging Metamodel component provides the model, the Debugger implements the view, and the Debug Manager serves as the controller. The concept was partially implemented by Ackermann [Ack10] in his Bachelor's thesis.

Figure 6.11 shows a screenshot of the TGG debugger. The debugger resembles the layout from traditional debuggers: The stack view is at the top left, properties of the selected stack frame (here: which nodes are matched to which objects) can be inspected at the top right. At the bottom, there is the TGG rule that is currently active. You can see a breakpoint at the upper left node :SystemModel. The red numbers in the nodes indicate the sequence of the matching process. The red-bordered node is the current position of the matching process. The default Eclipse debugging functions, e.g. pause, stepwise execution, run, can be used to control the transformation execution. We did not implement reverse debugging in the TGG Interpreter.

6.2 Model Transformations

We have developed several model transformation in the context of this thesis. Figure 6.12 shows the three essential transformations and their interplay.

[9]Technically, this is implemented using the Eclipse extension point mechanism.


Debug - mechatronic2mumlTGG/diagrams/system2structuredcomponent.tgg_diagram - Eclipse Platform
File Edit Navigate Search Project Run Story Diagram Window Help
Segoe UI
Quick Access
Resource
Java
Debug
Papyrus
SVN Repository Exploring
Debug
Variables
Breakpoints
*system2structuredcomponent.tgg_diagram
(Domain) principle
(Domain) correspondence
(Domain) muml
:SystemModel
:principlesolutionmodel2rootnode
:ModelElementCategory
s:System
:systemelementtemplate2component
sc:StaticStructuredComponent
:ModelElementCategory
:ComponentInstanceConfiguration
:system2componentinstance
:StructuredComponentInstance
TGG Transformati...grammar: (2%)

New_configuration (1) [TGG Transformation]
fwd on principle2muml-grammar
TGG Debug Thread
Pattern Matching
Rule Matching and Execution
Rule Application
Transformation Execution
TGG Interpreter Plug-in (de.upb.swt.qvt.tgg.interpreter_0.5.1 [10514])

Name	Value
Matched Nodes	
:ModelElementCategory	de.fujaba.modelinstance.impl.ModelElementCategoryImpl@63e011a2 (key: ...
:principlesolutionmodel2rootn	org.eclipse.emf.ecore.impl.DynamicEObjectImpl@1125e583 (eClass: org.ecli...
s:System	org.mechatronicmodeler.core.principlesolution.impl.SystemImpl@7403abeb (...
:SystemModel	org.mechatronicmodeler.core.principlesolution.impl.SystemModelImpl@d17c...
Candidate Node Binding	

```
org.mechatronicmodeler.core.principlesolution.impl.SystemImpl@7403abeb (name: BeBot)
```

Figure 6.11: Screenshot of the TGG Debugger during matching

The transformation from CONSENS to MechatronicUML maps a) the active structure partial model to component diagrams, and b) the Behavior–States partial model to Real-Time Statecharts. The ruleset is shown partially in Appendix A.

We developed a transformation from MechatronicUML to MATLAB/Simulink/Stateflow, which retains the semantics of MechatronicUML in Simulink simulations [HRB$^+$14, HRS13]. A particular challenge when developing this transformation was that block diagrams of Simulink are completely static, i.e., Simulink does allow neither instantiation nor destruction of blocks nor switching of signal flows. However, modern mechatronic systems, especially self-* systems, heavily rely on reconfiguration, and CONSENS and MechatronicUML thus allow dynamic models. We emulate the reconfiguration in MATLAB/Simulink/Stateflow by using disengageable blocks ("enabled subsystems"), *communication switches*, and *packet-based communication* (similar to Ethernet). Especially we deal with reconfiguration by switching on/off the enabled subsystems and/or changing the target address of the packets. For details on the concepts of this transformation we refer to Heinzemann et al. [HRS13]. We have published all technical details on this transformation (including the ruleset) as a technical report [HRB$^+$14]. To transform CONSENS models to initial MATLAB/Simulink/Stateflow, we reuse concepts from this transformation. This has been described in Sect. 3.2.3.

Finally, the generation of the consistency relation R also uses a TGG transformation. See Sect. 6.1.5 for details and Appendix B for the ruleset.

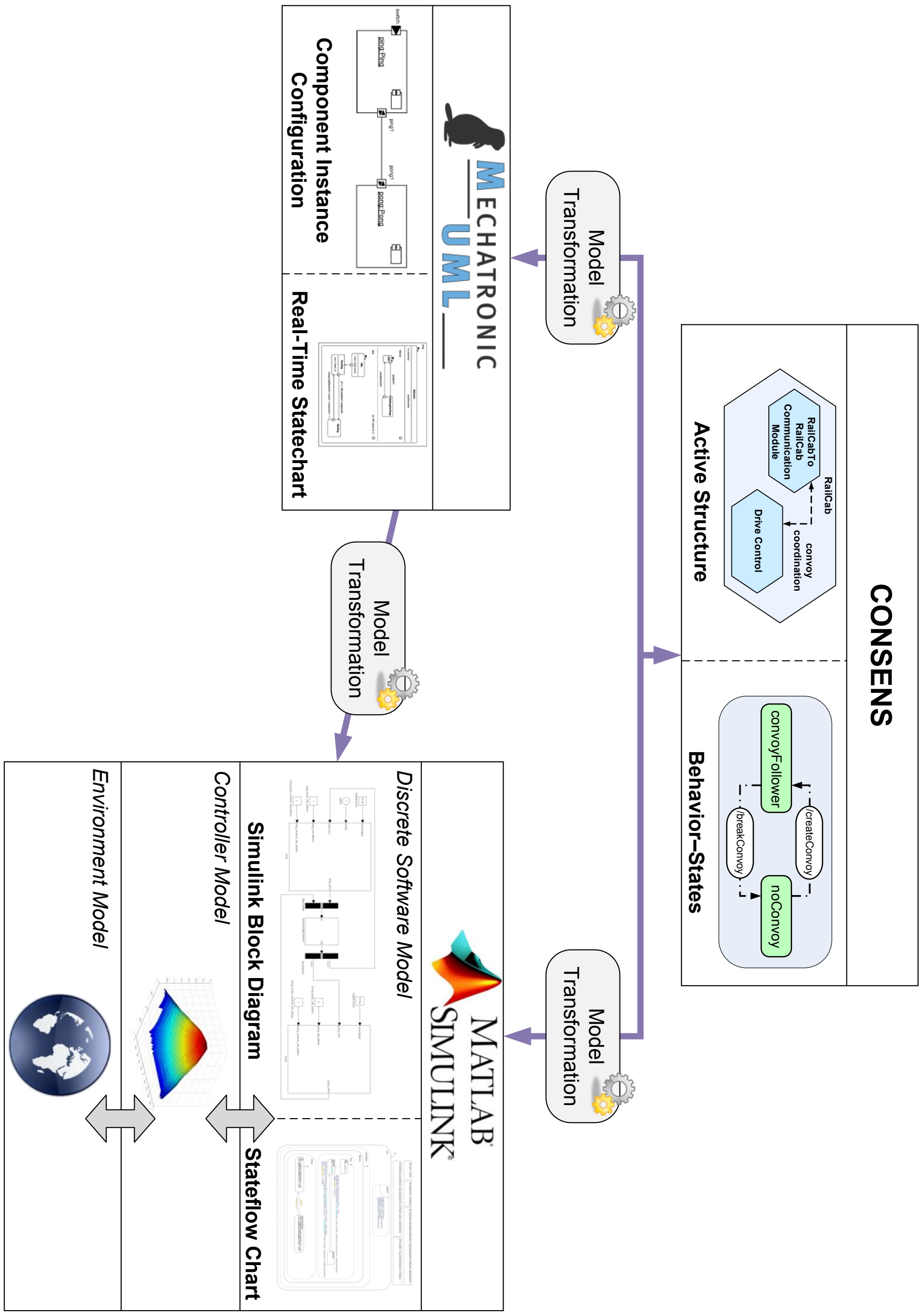


Figure 6.12: Overview of the implemented transformations

6.3 Evaluation

We developed several powerful extensions to model transformation/synchronization and TGG concepts within this thesis. We evaluated the techniques using a range of examples.

6.3.1 Incremental Updates with Element Reuse

With our novel incremental update algorithm (cf. Sect. 4.1), we intend to address the issue of information loss during update operations. Our algorithm does not focus on performance improvements. Some operations of our solution turn out to be relatively time-consuming. In this section, we evaluate the performance of our algorithm. By comparing it to a traditional incremental update algorithm, we estimate the performance impact of our novel features. Results indicate that in many practical situations, the performance drop will be less than 30 %, which does not severely impair the applicability of our approach.

Finding a valid matching (i.e., a monomorphism) is in $O(n^l)$ time for fixed patterns with l vertices and n host graph vertices [Epp95]. A typical algorithm is a depth-first search that starts using a given initial node-to-object match and tries to find a matching for the remainder of the graph. Whenever it runs into a dead-end (i.e., the matching is incomplete and no further nodes/objects can be matched), the algorithm has to backtrack.

A traditional TGG graph matching algorithm like the one implemented in the approach of Giese and Wagner [GW09] only has to compute one matching for the context and source produced pattern of a rule. That means that it does not have to calculate all matching possibilities by building up a complete matching tree. When it is guided by good heuristics, it may even find a matching without or with only few backtracking steps. This heavily reduces the runtime in practice.

Besides the regular rule matching (of the context and source produced patter), our novel algorithm also has to *build a complete matching tree* for the set of elements marked for deletion. In this way, our algorithm identifies all possible ways of reusing elements. Building a complete search tree is exponential in the number of candidate objects ("regular" objects and objects marked for deletion). However, we only search for a matching within the set of elements marked for deletion. Thus, our algorithm will exhibit a worst-case runtime of $O(n^l + {n_d}^l)$ for the rule matching, where n_d is the number of elements marked for deletion.

On the other hand, our algorithm may avoid several unnecessary rule revocations and subsequent re-applications: By reusing elements marked for deletion, we may be able to reestablish the context of dependent rule applications and, therefore, avoid their revocation. This could reduce the overall runtime of the incremental update run.

To estimate the performance impact of the features of our improved algorithm, we implemented both our algorithm and the one by Giese and Wagner [GW09] in our TGG Interpreter. We used two different system models (M_1 and M_2) that are transformed to and synchronized with a software model using the CONSENS-to-MechatronicUML transformation (see Appendix A). We applied an initial transformation, so the CONSENS and MechatronicUML models were consistent after that transformation. We then measured the average runtime (on a 2 GHz Intel Core2 Duo, Windows 7 x64, Eclipse 3.7) over three incremental update runs after applying editing operations op_1 or op_2 to the CONSENS model.

M_1 is a simplified active structure of the RailCab and is shown in Fig. 6.13. It contains a total of 80 model elements and is rather flat (three levels of system elements). M_2, a technical example, is an extended active structure and contains 150 model elements. Figure 6.14 shows M_2. The main difference to M_1 is that it has additional nested system elements, resulting in a deeper hierarchy (six levels).

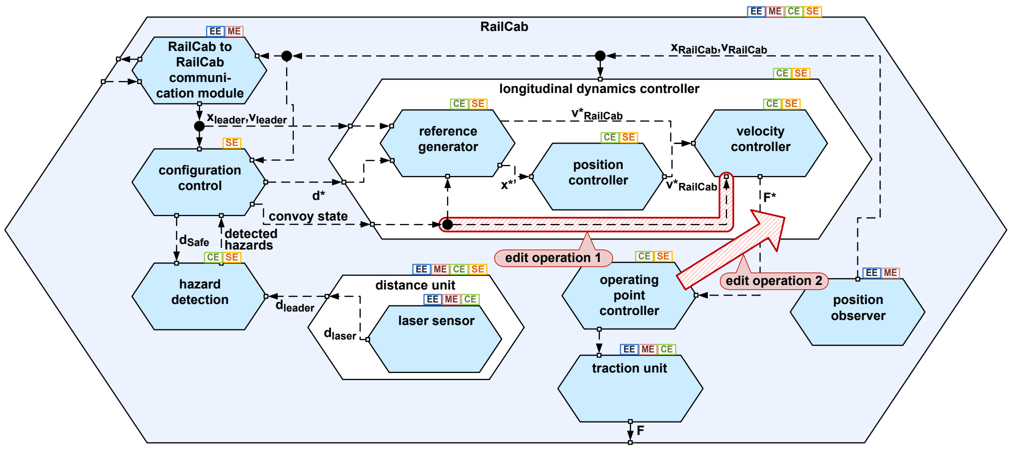


Figure 6.13: Performance evaluation scenario 1: Typical editing operations in a system model during the Design and Development phase

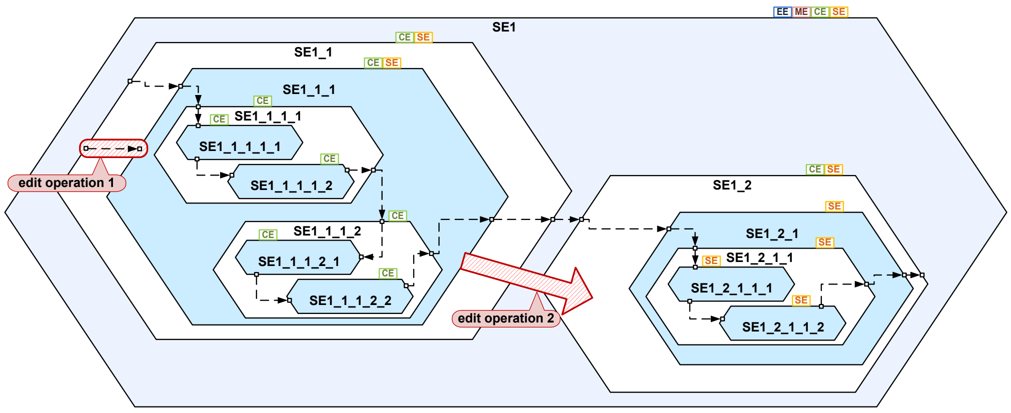


Figure 6.14: Performance evaluation scenario 2: Artificial example with larger system element hierarchy

Applying op_1 means removing a port from a system element of the second level as described in the example above. op_2 is a more complex editing operation

	Standard Incr. Updates	Improved Incr. Updates	Relative Difference
$op_1(M_1)$	756 ms	867 ms	+ 14.7 %
$op_2(M_1)$	794 ms	1022 ms	+ 28.7 %
$op_1(M_2)$	1320 ms	1201 ms	- 9.0 %
$op_2(M_2)$	1357 ms	1563 ms	+ 15.2 %

Table 6.1: Runtime performance for propagating the editing operations shown in Fig. 6.13 and 6.14

that consists of four move and delete operations on sub-system elements of the second level. E.g., the configuration control is moved into the longitudinal dynamics controller.

Table 6.1 shows the results of the performance evaluation. The runtime of the new algorithm on model M_1 after editing operations op_1 resp. op_2 is increased by 15 % resp. 28 %. For op_2 on model M_2 there also is a performance drop of 15 %, but for op_1 on M_2 our algorithm performs 9 % better than the old one.

This performance improvement for the case $op_1(M_2)$ is due to the deeper hierarchy of system elements. Removing the information flow causes the system element SE1_1_1 to be mapped to a controller component instead of a hybrid component. Revoking the rule application leads to a revocation of all dependent rule applications (for all the contained system elements SE1_1_1_1, SE1_1_1_2, ...). Our improved algorithm first checks for alternative rules before revoking dependent rule applications (cf. Sect. 4.1.2).

In summary, our algorithm works best (in terms of performance) when there are only few altered elements, because then the number of candidate objects for the partially reusable pattern search is low. There could even be performance improvements when a large amount of revocations of dependent rules is prevented. Overall, the prevention of information loss comes with a performance decrease in most editing cases. However, in typical editing cases we examined, the maximum performance drop was only about 30 % in comparison with the old algorithm.[10]

In our examples, we observed that good partial matchings were often found early in the partially reusable pattern search. Thus, the metrics could be used to determine the quality of a partial matching already during the search. When a good-quality matching is found, we could even decide to terminate the search, possibly long before the complete matching tree is build up. Then we may miss the intended way of reusing the elements, but we believe that there are many examples where the metrics could determine the "best" matching early, improving the overall performance significantly. However, we did not elaborate further on this topic.

[10]Note that runtime performance was not the primary goal when designing this algorithm. Thus, it is likely that performance improvements are possible by optimizing the algorithm.

6.3.2 Bidirectional Synchronization

In Sect. 4.3.4, we presented a simultaneous, bidirectional synchronization method that tries to partially synchronize the models by propagating non-conflicting changes immediately. Using this method, the models that have to be compared and merged by the model-merging tool differ less. The rationale behind this idea is that the less differences between the models, the better the merging results will be. To evaluate the usefulness of this approach, we compare it with the simple simultaneous, bidirectional synchronization that only builds upon complete incremental updates (cf. Sect. 4.3.3).

We use EMF Compare as external model-merging tool. The results of our tests indicate that the merging of EMF Compare can be reliable for smaller models of *some* metamodels. However, up to the current version of EMF Compare (3.0.2), even minimal example models of certain metamodels can cause "phantom" differences, although the compared files are bitwise identical.[11] This casts a poor light on the merging reliability of EMF Compare in general. Thus, the evaluation results in this section have to be regarded with suspicion. A further evaluation is advised as soon as these problems have been fixed or other reliable model comparison tools emerge.

In our experiments, EMF Compare was able to merge all changes without errors when it works with models that use universally unique identifiers (UUIDs)[12] for all objects. Hence, in cases where both models of a transformation use UUIDs, it is not necessary to use the improved bidirectional synchronization. However, not all modeling languages use unique IDs. In most cases, the modeling language is not under control of the transformation developer. Especially when integrating third-party tools, we cannot simply add unique identifiers to a modeling language just to make the job of our synchronization easier.

As the modeling languages in our scenario all have UUIDs, we use another example to test the merging precision: the TGG benchmark of Hildebrandt et al. [HLG+13] and Leblebici et al. [LAS+14]. The (meta)models in that example are not affected by the "phantom-differences" bug. We (automatically) create source models of different sizes, transform them to target models, and then apply random changes to both source and target models. Next, we run both the simple synchronization and the advanced synchronization (that first propagates non-conflicting changes) and compare their merging precision. It is important to note that none of the applied changes is in conflict with another; thus, both synchronizations should be able to automatically merge them.

[11]For instance, this is the case for the ClassesToTables example of our TGG Interpreter tool suite.

[12]EMF has different ways of referencing an object. First, a UUID of an object never changes and allows referencing it over its whole lifetime; if UUIDs are available, these are the best way to reference objects. Second, an attribute of a class can be defined as identifier, e.g. the name of an object. Third, an object can be referenced by the position in the container of its parent. This will create URIs similar to XPath [W3C01], for example, `//@modelElements.42/@componentInstances.5`, which references the 6th component instance withing the 43th model element. The latter is the default way in EMF.

Using the second or third way, changes to the identifying attribute or in the order of the list will cause a URI to point to nothing or to a wrong object.

For models with a large number of elements that differ significantly, the merging degrades. For the simple synchronization, we encountered imprecise merging results starting with models of size > 5000 and number of changed elements > 500 (in both model versions). Using the advanced synchronization, some, but not all of the wrong matchings can be avoided.

In summary, the advanced synchronization is primarily useful for large models that do not use UUIDs and have a significant amount of changes. However, the tool in use, EMF COMPARE, suffers from some serious flaws that makes it difficult to be applied for certain models. This reduces the validity of this evaluation.

6.3.3 TGG Debugging

To evaluate the implemented debugging approach presented in Sect. 5.4, we investigate which kinds of bugs we are able to find with the approach and how easy this is. Thus, we need a classification of bugs that occur in TGG model transformations. We discuss the capabilities of the debugging concept using this classification of bugs.

KUSEL ET AL. [KSWR09] present such a classification for QVT. On a top level, they distinguish between *intra-relational* (concerning only one relation) and *inter-relational* (concerning more than one relation) pitfalls. In the following, we adapt this classification for TGGs. We discuss whether the presented transformation debugging concept helps the transformation engineer to detect and locate bugs. Assume we are transforming or updating in forward direction.

Intra-Rule Bugs Intra-rule bugs are bugs that are only due to a single rule. They primarily affect the target model elements that are created in this rule, although such a bug may also affect further rule applications.

- **Too weak matching pattern**: The context and the source produced part of the rule cover more situations than intended.
  Result: The rule is also applied in cases where it is not designed for.
  Detection: This type of bug can be found using a breakpoint on the rule-application step of a certain rule. Whenever this rule is about to be applied, the transformation engineer is able to inspect the current matching. Ideally, we would like to "step back in time", reversing the transformation execution, such that we can find out where exactly the error occurred during the previous pattern-matching step. However, this is not supported by existing transformation engines. Therefore, we have to debug in two steps. In this second step, the transformation engineer can use a pattern matching breakpoint on an object that is wrongly matched to find out whether the pattern matching accepts this certain situation.
- **Too restrictive matching pattern:** The context and the source produced part of the rule cover less situations than intended.
  Result: The rule is not applied in all intended cases; thus, some elements are not translated.
  Detection: Given the not translated elements, the engineer can use a pat-

tern matching breakpoint using these elements (together with the respective rule) as a breakpoint condition; whenever these objects are considered by the pattern matching, he or she can inspect why the pattern does not accept them using stepwise execution.

- **Wrong application conditions or constraints:** The application conditions/constraints contains erroneous calculations.
  Result: The rule is applied in cases where it is not designed for, or it is not applied for all intended cases.
  Detection: Wrong application-condition/constraint calculations can be identified using breakpoints on the constraint-checking execution step with the application condition/constraint as breakpoint condition.
- **Wrong source-target connection:** The mapping between the source and the target model elements is ambiguous, i.e., there is more than one way of matching the context pattern.
  Result: Created model elements are linked to wrong parent elements, or attribute values are calculated using wrong elements' attributes.
  Detection: The engineer could use a breakpoint on the successful application of the rule with the corresponding source model element as breakpoint condition. The he or she can inspect the matching to find out the origin of the bug.
- **Wrong attribute constraints:** The attribute constraints contain erroneous calculations.
  Result: Attribute values of created model elements contain wrong values.
  Detection: The transformation engineer can use a breakpoint on the erroneous attribute constraint in the enforce-execution step to inspect the calculation of the result.

Inter-Rule Bugs Inter-rule bugs are bugs that are due to the interplay between two or more rules.[13]

- **Uncovered source model parts:** No rule covers a certain part of the source model.
  Result: Transformation result is incomplete.
  Detection: If no rule matches certain elements of the source model, this can be debugged using a breakpoint on the pattern matching of (one of) these source model elements within the rule(s) that should translate them. Then the engineer stepwise continues the pattern matching process to find the bug. If such a breakpoint is never hit, that means that there is a problem due to rule dependencies, which can be found with the help of the dependency view.
- **Conflicting rules:** More than one rule covers the same source model part.
  Result: Non-deterministic or unexpected transformation result.
  Detection: If more than one rule covers the same source model part, this

[13] A special case is a rule that produces the precondition for itself again, e.g., when translating hierarchical structures. We also consider bugs that are caused by the interplay of different rule applications of the same rule as inter-rule bugs.

does not necessarily lead to non-determinism when only using a single transformation engine that deterministically decides in such situations. Therefore, such bugs may be hard to find, as they manifest themselves rarely. Nevertheless, the rule dependency view can give insights in potential rule conflicts.

- **Missing precondition:** The rule's context pattern requires a condition that is never created by other rules.
  Result: The rule is never applied, and some elements are not translated.
  Detection: If a rule's context pattern requires a condition that is never be created by other rules, the rule dependency view does not show any other rules that this rule depends on.

The bug type that occurred most frequently in our TGG practice, "too restrictive context pattern", can be investigated easily using our debugging concept. For other bug types, it may be not that easy. Most important limitation here is the lack of reverse debugging facilities[14]: Often we would like to define a breakpoint that hits in a situation that will be definitely wrong; then we want to investigate how this situation could have evolved.

General reverse debugging facilities require substantial changes to the transformation engine. However, if we limit reverse debugging only to the pattern-matching execution step, a "virtual" reverse debugging could be implemented simply by recording all events that occurred during the pattern matching. To reverse the transformation, we could stepwise undo these recorded steps virtually, without actually modifying the engine's or the models' state. This is possible because no model modifications (node/edge creations/deletions, property assignments) occur during this step. A drawback is that no modifications (like changing a node's matching, or the value of an object's property) would be possible during this "recorded playback".

In our experience, our debugging approach helps identifying and fixing bugs in a transformation more quickly. However, its usefulness and potential improvements have to be investigated further, e.g., with a user study that also involves inexperienced TGG users.

[14]Reverse debugging is also missing in most debuggers for general purpose programming languages.

CHAPTER 7

Conclusion and Future Research

This thesis presented a comprehensive method for ensuring consistency across different models used in model-based systems engineering. We put particular emphasis on advanced mechatronic systems that are developed in a highly interdisciplinary manner. Our techniques help preventing time-consuming and cost-intensive iterations in the development process caused by inconsistencies between the development artifacts of different disciplines. We evaluated the method using examples from the development of the RailCab transportation system.

7.1 Summary

The development of complex mechatronic systems requires collaboration of different disciplines, especially mechanical engineering, electrical engineering, control engineering, and software engineering. Between those disciplines, there are several overlaps and interfaces. For instance, the mechanical construction of a vehicle determines its total weight, and this weight is important for the control strategies responsible for steering and braking. Interdependencies, and thus complexity, also stem from the increasingly advanced software contained in today's technical systems. For instance, software engineers implement the communication between different systems, but this communication heavily influences the control strategies.

To tackle the complexity of modern technical systems, the development is usually model-based. Various models represent different aspects of the system under development: From an abstract, interdisciplinary system model to detailed discipline-specific models, we find models of diverse abstraction levels in the development process. These models are subject to changes during the whole development process by teams of several engineers.

The interdisciplinary dependencies manifest themselves as dependencies between those models. Consequently, we have to ensure that whenever changes occur in one model, the consistency of all other models will be checked. For interdisciplinary relevant changes, we have updated all affected models.

Model transformation and synchronization techniques are a promising approach to propagate changes between different, but interrelated models. However, existing model transformation techniques have several drawbacks when applied in such a scenario:

- Models only partially overlap in their information. The transformation only considers the overlapping model parts. Propagating changes may cause a *loss of information* in those model parts that are not subject to the transformation.
- We find typically *horizontal transformations* that map between models of different abstraction levels. However, existing transformation techniques are primarily designed for vertical transformations (i.e., mapping between models of a similar level of abstraction).
- Models will not be synchronizing immediately after changes. Instead, the synchronization is deferred. A special systems engineer later explicitly executes it. As a result, interdisciplinary editing conflicts may arise where the different models contradict each other – a situation a traditional model transformation solution cannot resolve.
- The model transformation specification itself is hard to develop, understand, and maintain.

Some model transformation techniques introduce a large amount of user interaction during the transformation/synchronization to mitigate these issues. However, developing and executing a model transformation requires extensive training. In practice, there are only few engineers capable of performing this manual task.

In this thesis, we have developed an integrated model synchronization method that addresses these issues. It can be customized with respect to the scenario and – most importantly – in its amount of automation. Depending on the level of system and transformation engineers available, it may be run fully automatic or with more or less user interaction. Using metrics and heuristics, we encode expert knowledge and guidelines that have proven successful previously.

In particular, we developed a novel incremental update algorithm that prevents information loss by reusing elements. It is the basis for a bidirectional, conflict-resolving synchronization. Our approach also supports vertical transformations by defining refinements for discipline-specific modeling languages. To ease the definition of transformations, we presented an improved syntax, enhanced semantic features and debugging facilities for Triple Graph Grammars [Sch95], a formal model transformation technique.

By reducing manual interventions, improved automation, and easing the transformation definition, we argued that our method can significantly improve the interdisciplinary, model-based development of complex mechatronic systems, as well as other model-based approaches like software development with MDA. It helps avoiding time-consuming and cost-intensive iterations in the development process and leads to less errors and flaws due to interdisciplinary inconsistencies in the final product.

7.2 Future Research

The concepts presented in this thesis give rise to new research questions. There are also reasonable extensions, e.g., with respect to performance, user interaction, or improved analyses.

Most solutions presented in this thesis are not strictly TGG-specific. In particular, most existing model-to-model transformation techniques suffer from similar drawbacks. It remains to be investigated how the presented solutions can be ported to other transformation techniques.

Concerning the definition of refinement rules, the discipline's engineers typically encounter examples for new refinements when modifying their discipline-specific models. With this example in mind, they define the corresponding refinement rule(s) manually using Story Diagrams. With the rising importance of model transformation, techniques emerged that use examples to generate transformation rules [KLR$^+$12]. Using these techniques, we could (at least semi-automatically) create the corresponding refinement rules using one or more examples of valid refinements.

Concepts in this field of "model transformation by example" (e.g., [Var06, KLR$^+$12] can also be used to ease the definition of model-to-model transformations. However, research suggests that defining a transformation remains an "iterative and interactive" process [Var06]. Model-transformation-by-example techniques will therefore also profit from powerful debugging concepts and better-understandable representations. Combining these approaches seems worth further research.

Incrementally updating currently is technically separated from model merging and the resolution of editing conflicts. In a case where we have different ways of propagating a change to a target model, it is also reasonable to analyze whether these alternatives could lead to editing conflicts. In general, we should choose the way that produces the least conflicts.

With increasing reliability and usability of model-comparison-and-merging solutions, the importance of information-preserving incremental updates may, in turn, decrease. As of today, model merging tools are still mendable, but they underwent large improvements over the last years. For instance, EMF Compare provides sophisticated visualization means, even for graphical modeling languages. Using model merging in distributed development processes should be investigated further, from a technical as well as an empirical viewpoint.

Some transformation languages/tools are better suited for certain scenarios than others. Up to now, few works focus on how to select an appropriate transformation technology. Lehrig [Leh12] presents a first framework for assessing the quality of transformations. He first classifies a given transformation scenario. Using this classification, he presents a first, simple decision tree that guides an engineer in selecting a good transformation technology. This work may serve as a starting point for further empirical research.

In general, the empirical validation of the approaches presented in this thesis can be improved. Up to now, the techniques have only been tested by students and academic researchers who all have had significant experience with model-based development. Thus, the applicability in industry-relevant processes has yet to be investigated. This holds both for the transformation-use aspect (i.e., which education is required to efficiently transform and incrementally update models) and the transformation-engineering aspect (i.e., how easy it is to develop and maintain a model-to-model transformation specification).

Concerning the latter aspect, we presented some improvements for understandability and maintainability in this thesis (concrete syntax in TGG rules, TGG debugging means). However, TGGs (and model transformation in general) remain a complex technology that requires a large amount of training and experience. This is at least partially due to a lack of abstraction: TGG rules are precise and formal specifications that define model mappings on the low level of metamodel concepts. However, engineers typically work with more abstract concepts that may not be directly reflected in the TGG rules. On the other hand, formal precision is of high importance when developing safety-critical systems. Therefore, we have to find a reasonable compromise between preciseness and understandability.

In software and system development, engineers typically start the development on an abstract level (requirements, general functions) and get more and more concrete in the course of the development (active structure, then discipline-specific models). We should aim for a similar approach when developing model transformations: first define the general requirements and basic inter-model dependencies, then implement the details of the actual transformation. One promising idea is the creation of an abstract *dependency description language*. Engineers from different disciplines should be able to specify all kinds of (inter- and intra-model) dependencies using the concepts of their own domains with the help of that language. These dependencies could be used to generate initial model transformation specifications (in different model transformation languages), which could be refined by specialized transformation engineers in the following.

When pairwise synchronizing three or more models, the mappings between these models must not be in conflict with each other. Otherwise a transformation from model A over model B to model C could lead to different results than a direct transformation from A to C. In such a scenario, we have to ensure the *consistency of consistency* by analyzing whether the different mappings are compatible with each other. We could also use an abstract dependency description language to help ensuring the consistency of consistency in cases where the transformations are implemented with different model transformations approaches/languages.

APPENDIX A

Transformation from CONSENS to MechatronicUML

This TGG ruleset implements the transformation from a system model (specified using the CONSENS language) to a MechatronicUML software model. The transformation has been developed in the course of the project group SafeBots II. The principles of this transformation are described in Sect. 3.2.2. It makes use of the concept of rule inheritance we described earlier [GR12]. Figure A.1 shows the inheritance relations between the different TGG rules of this ruleset.

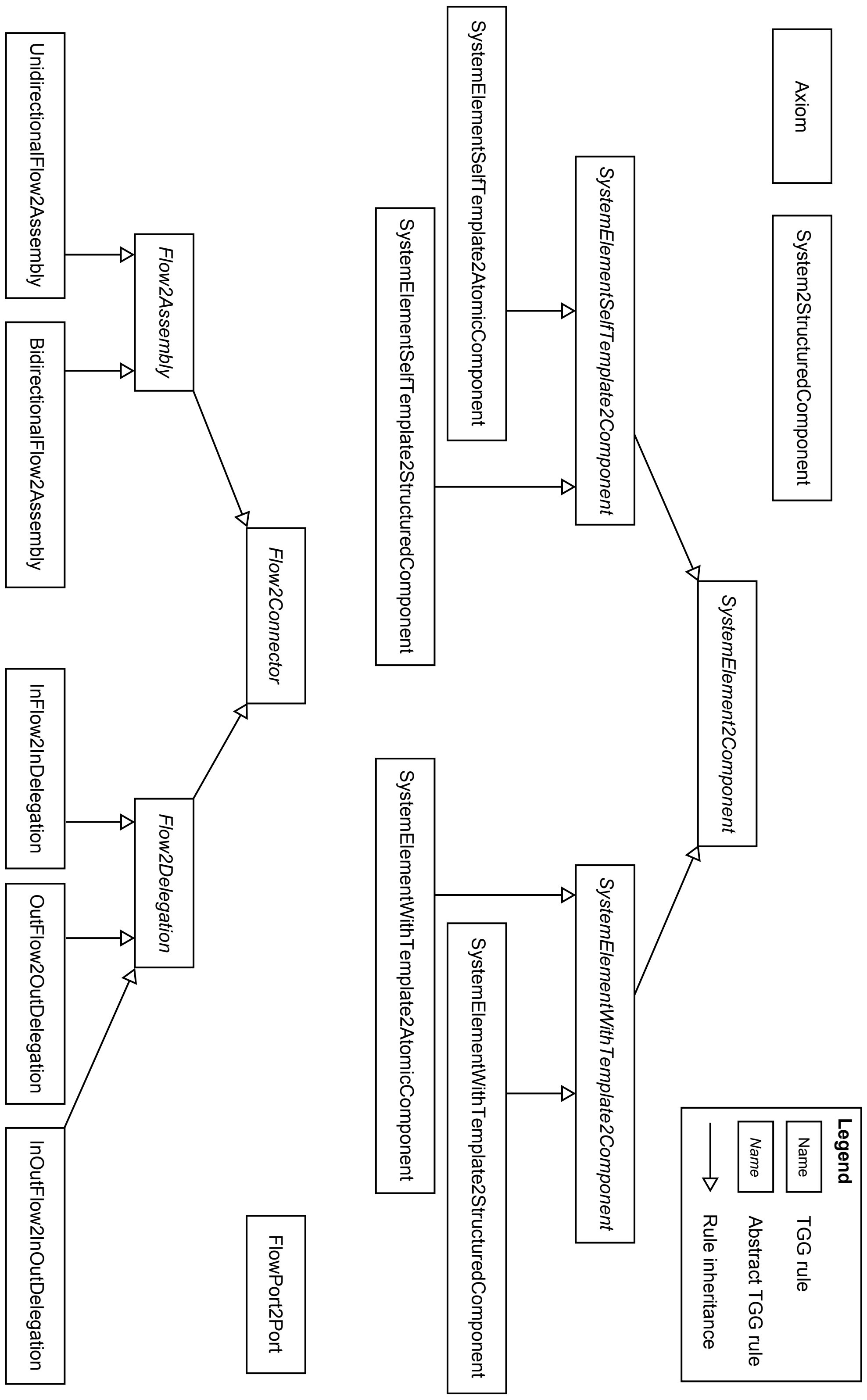

Axiom
System2StructuredComponent
SystemElementSelfTemplate2AtomicComponent
SystemElementSelfTemplate2Component
SystemElementSelfTemplate2StructuredComponent
SystemElement2Component
SystemElementWithTemplate2AtomicComponent
SystemElementWithTemplate2Component
SystemElementWithTemplate2StructuredComponent
UnidirectionalFlow2Assembly
BidirectionalFlow2Assembly
Flow2Assembly
Flow2Connector
InFlow2InDelegation
OutFlow2OutDelegation
InOutFlow2InOutDelegation
Flow2Delegation
FlowPort2Port
Legend
Name
TGG rule
Name
Abstract TGG rule
Rule inheritance

Figure A.1: Inheritance relations between the different TGG rules

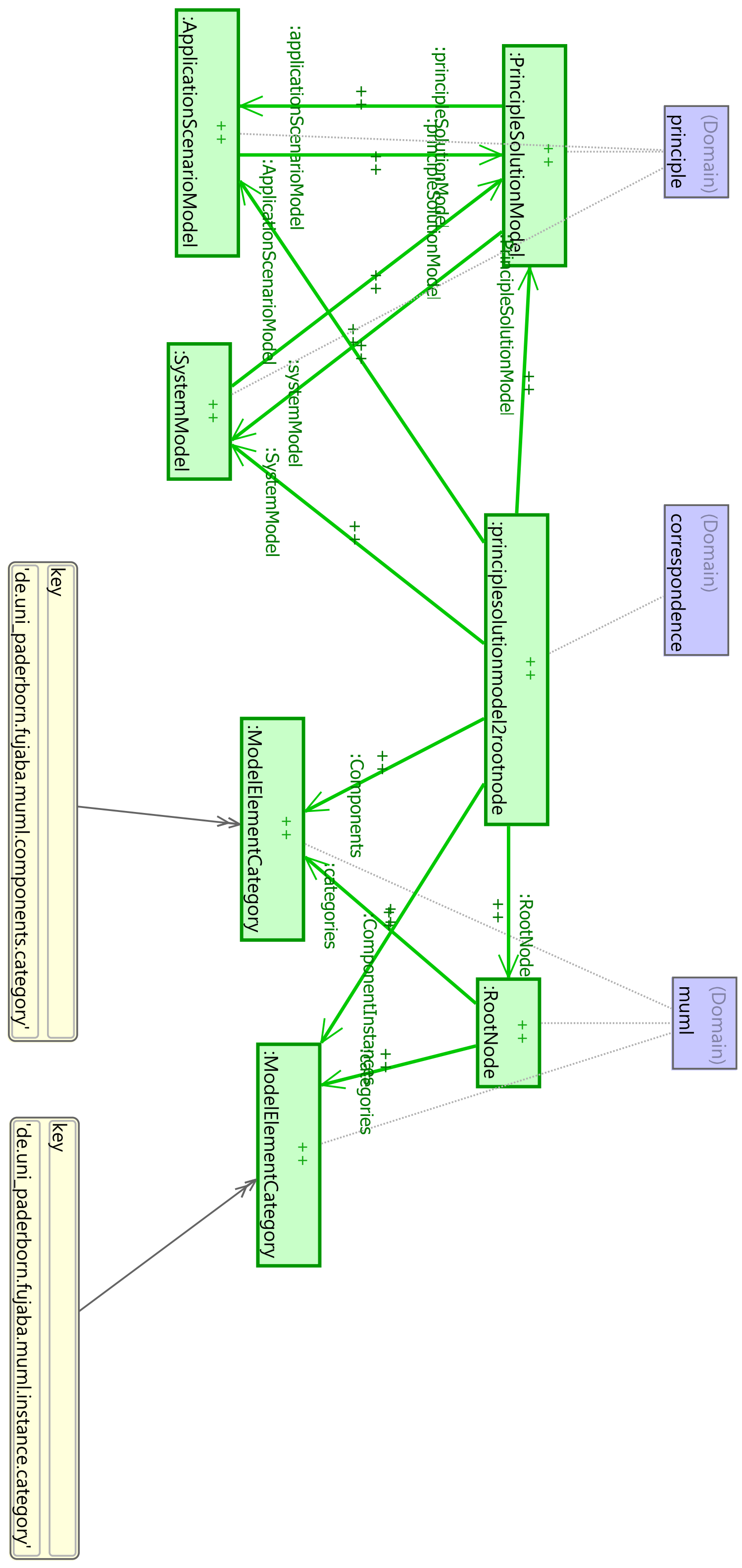


Figure A.2: TGG Axiom CONSENS2MUML

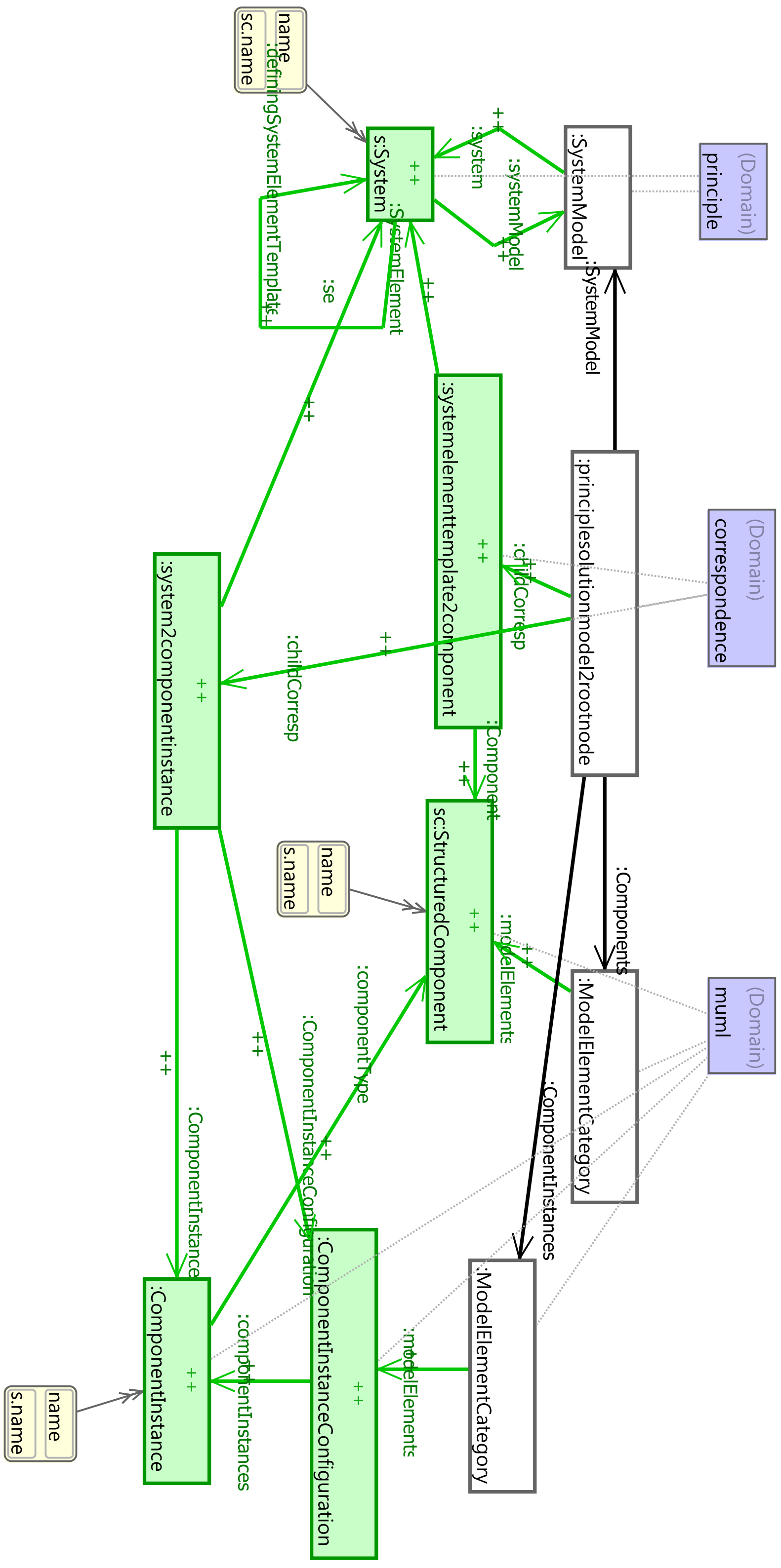


Figure A.3: TGG Rule System2StructuredComponent

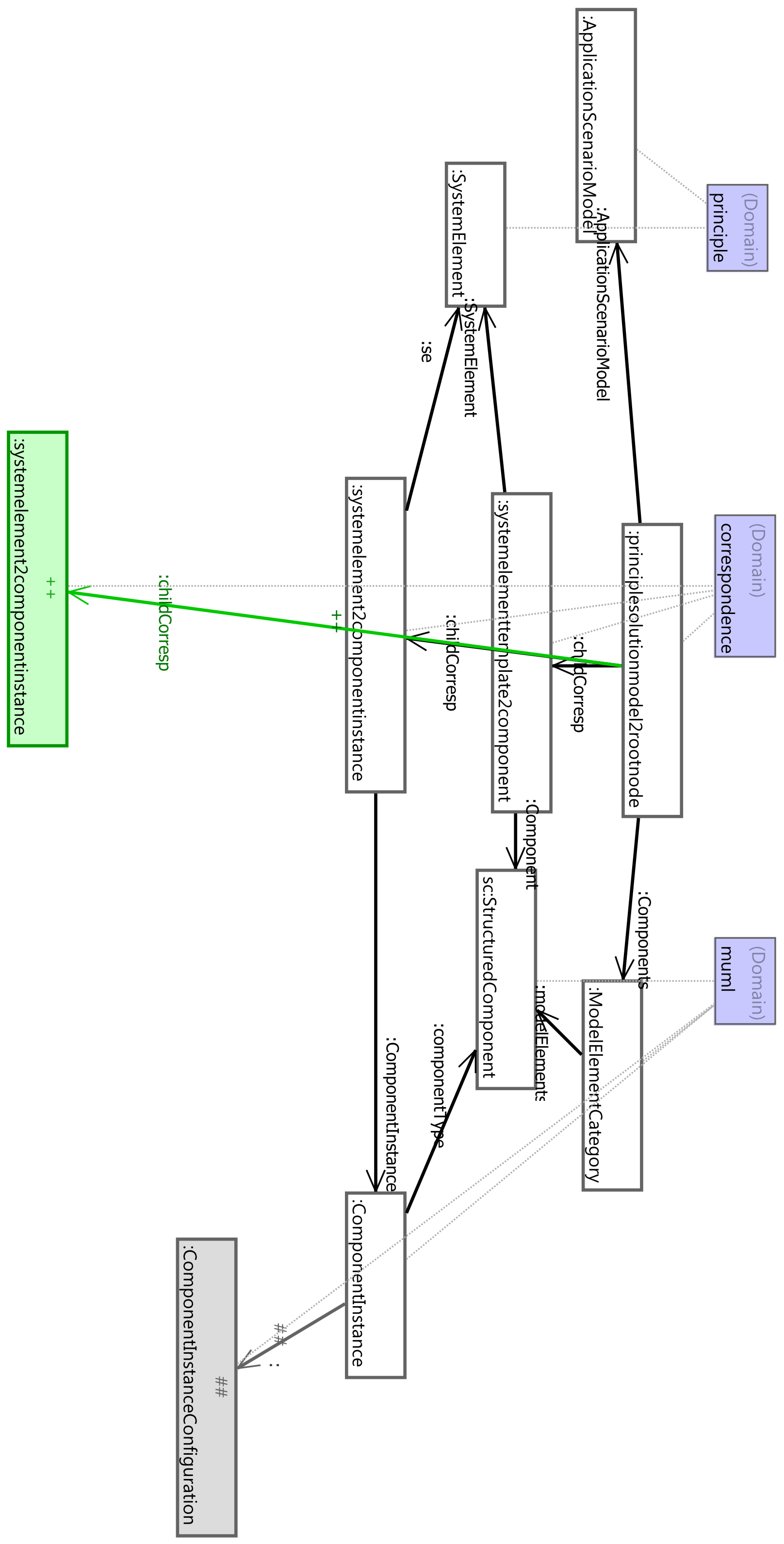


Figure A.4: TGG Rule SystemElement2Component

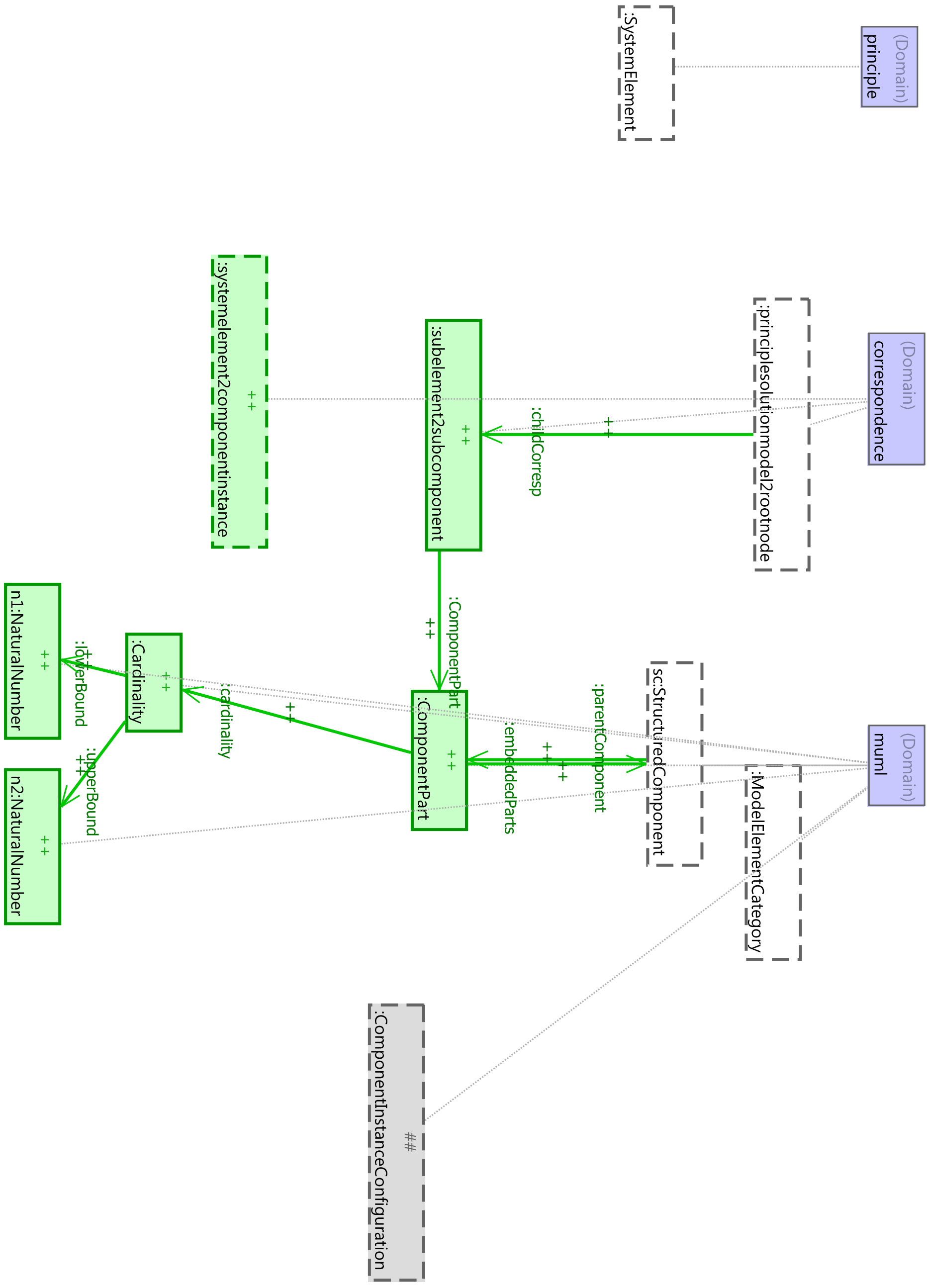


Figure A.5: TGG Rule SystemElementNoTemplate2Component

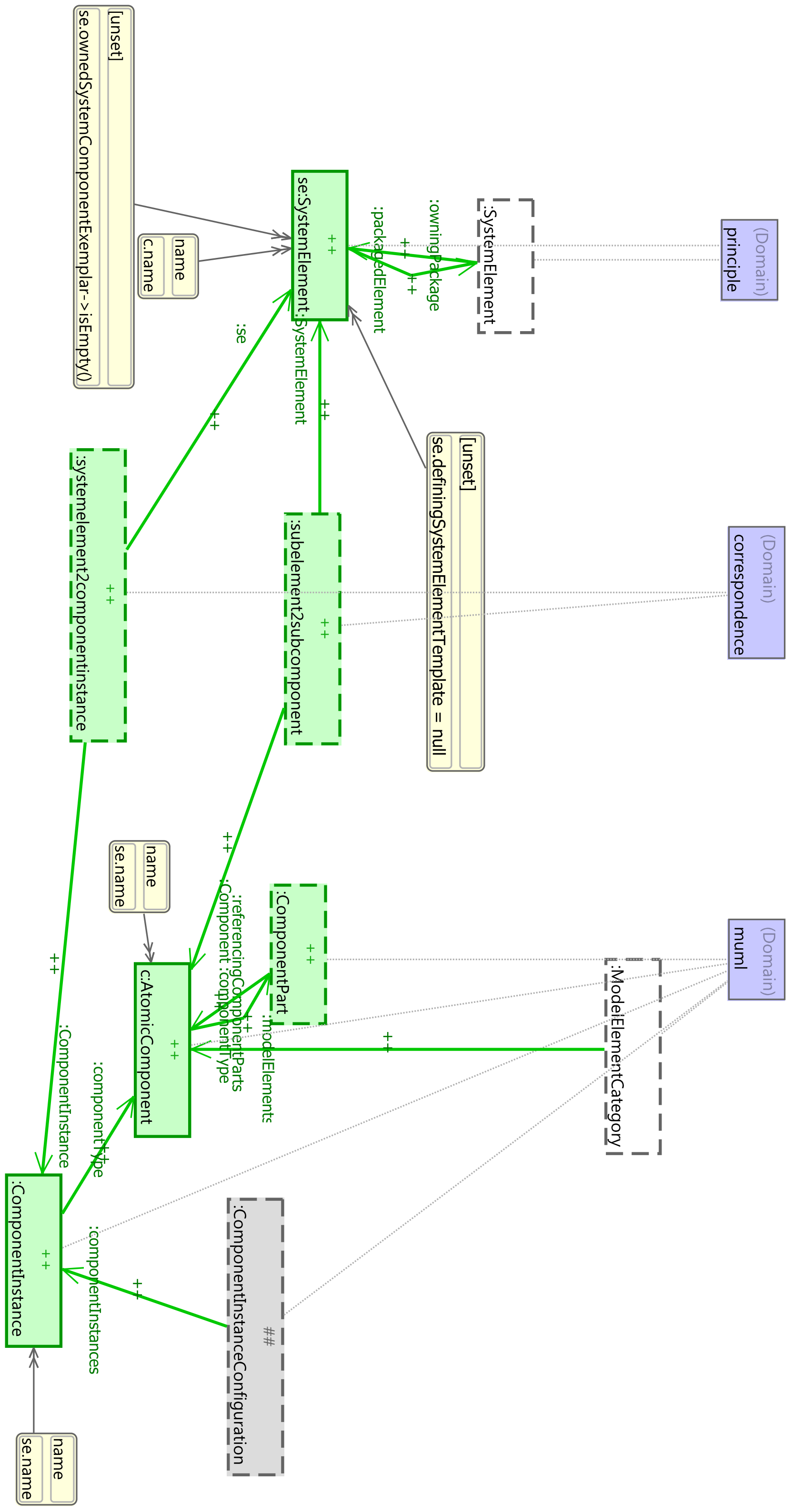


Figure A.6: TGG Rule SystemElementNoTemplate2AtomicComponent

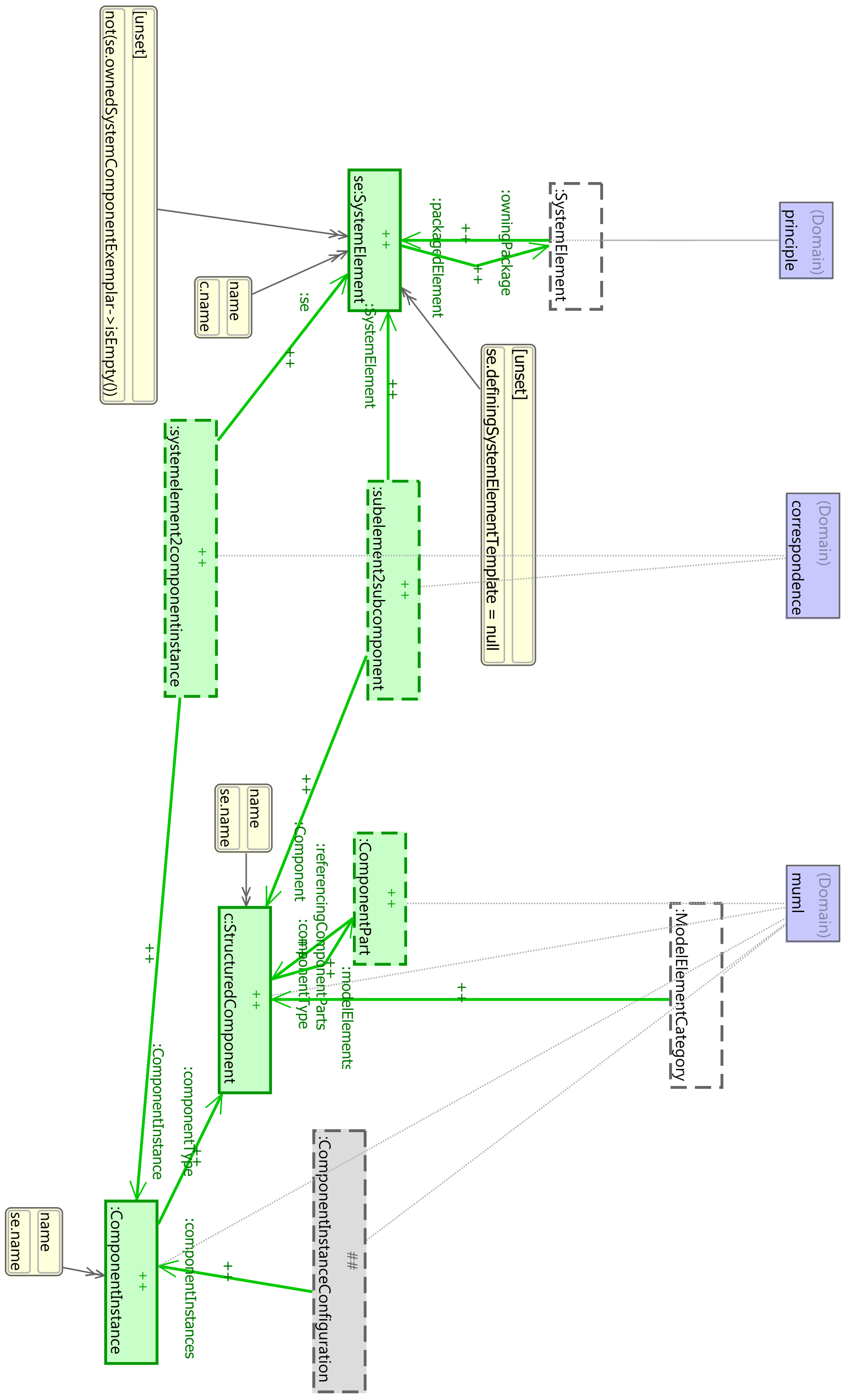


Figure A.7: TGG Rule SystemElementNoTemplate2StructuredComponent

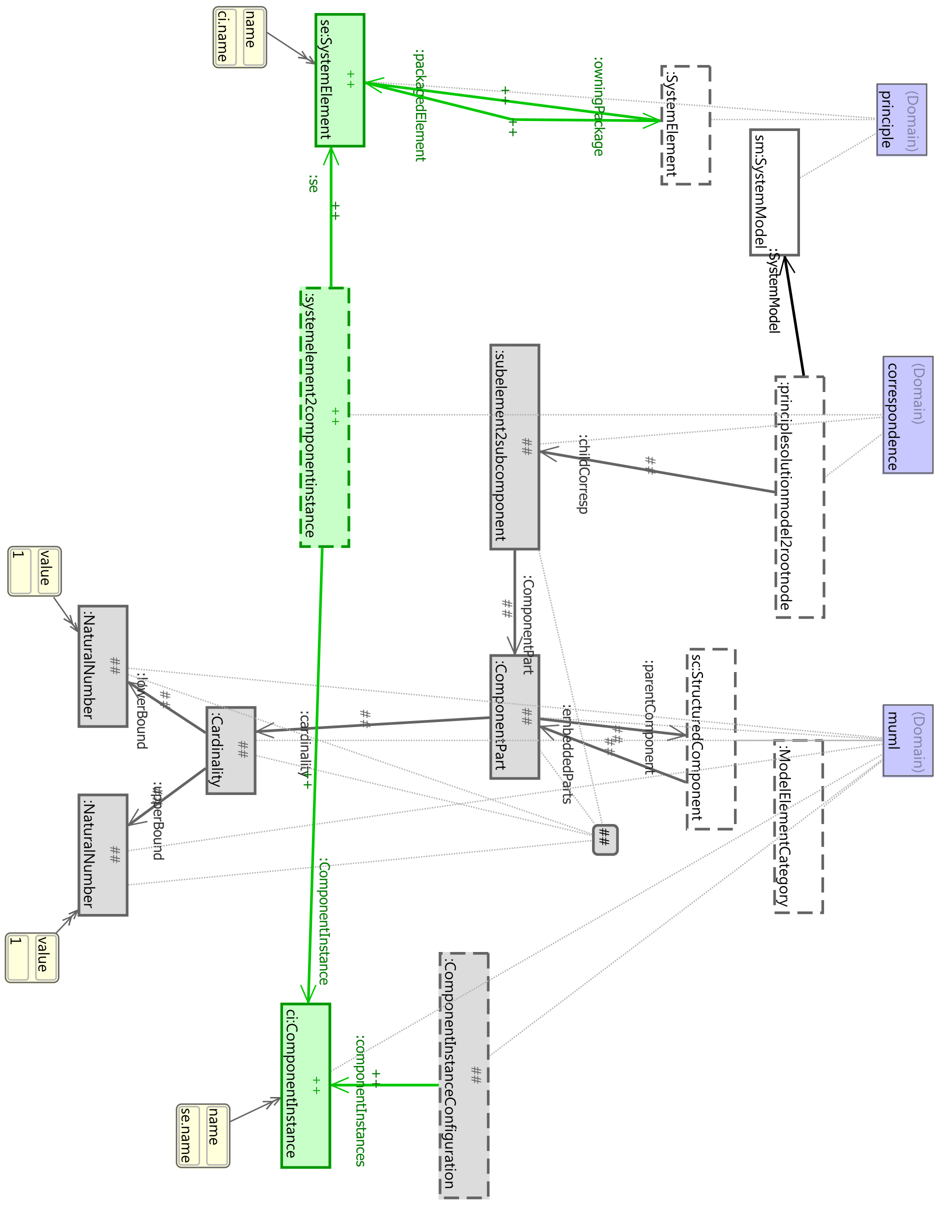


Figure A.8: TGG Rule SystemElementWithTemplate2Component

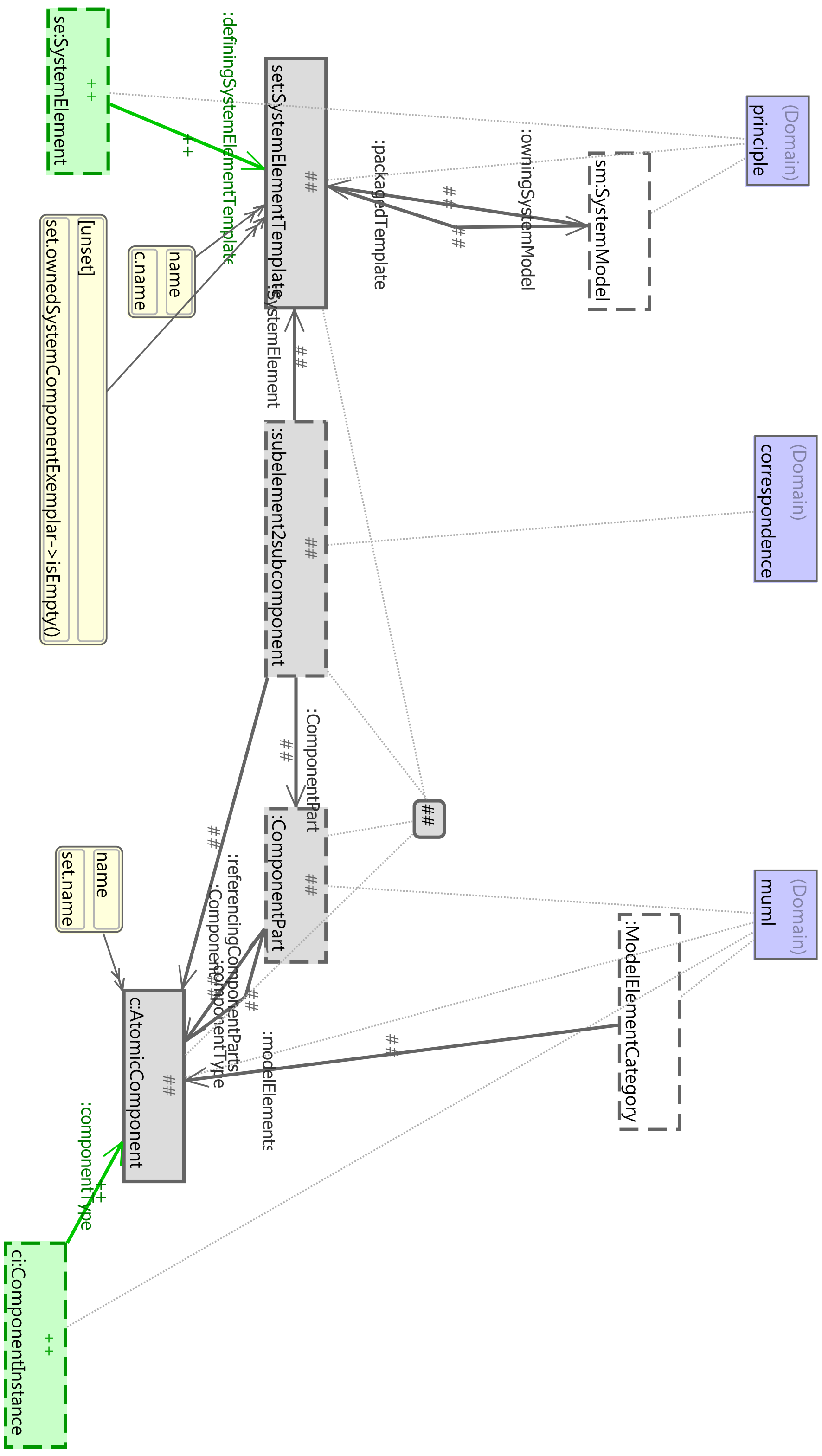


Figure A.9: TGG Rule SystemElementWithTemplate2AtomicComponent

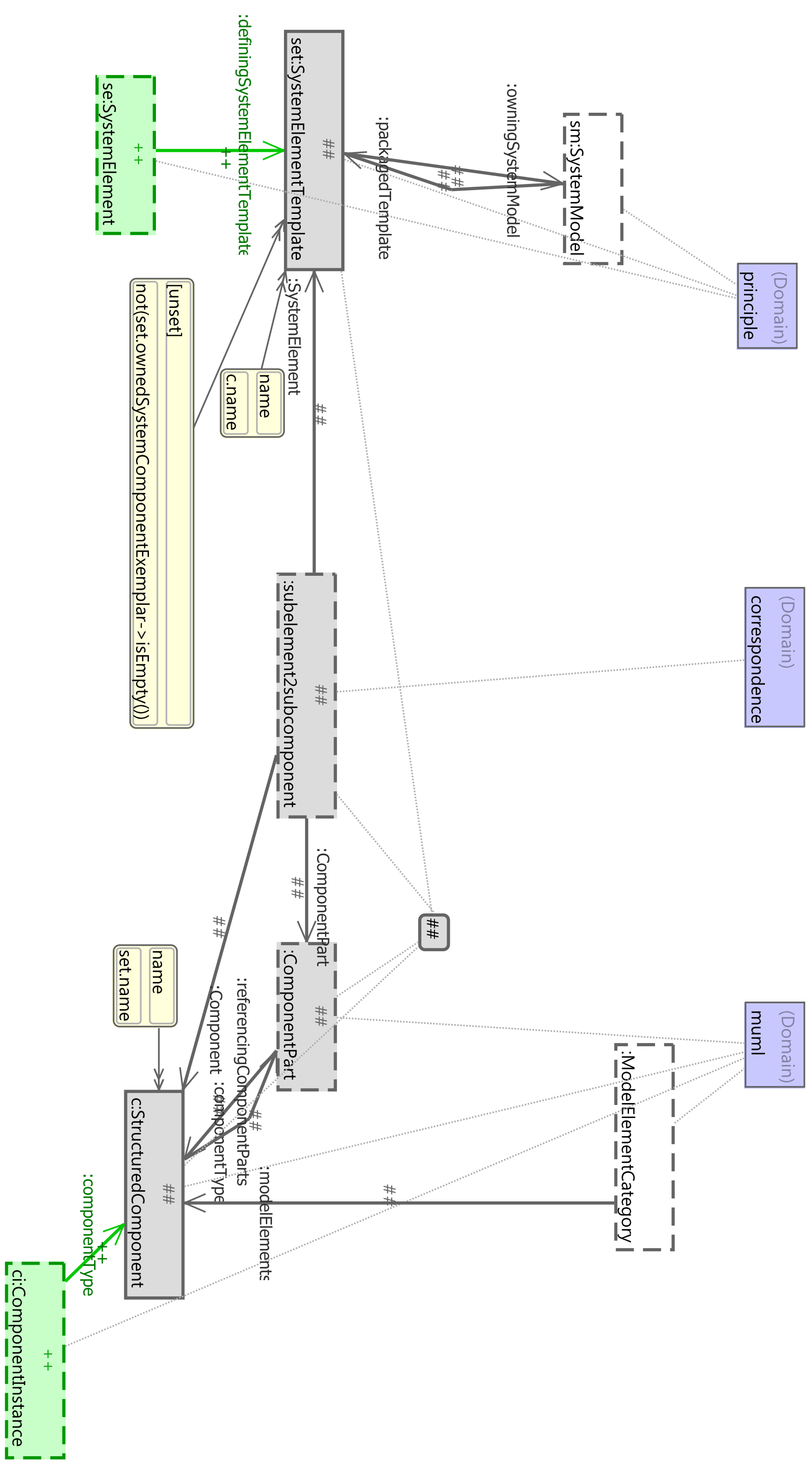


Figure A.10: TGG Rule SystemElementWithTemplate2StructuredComponent

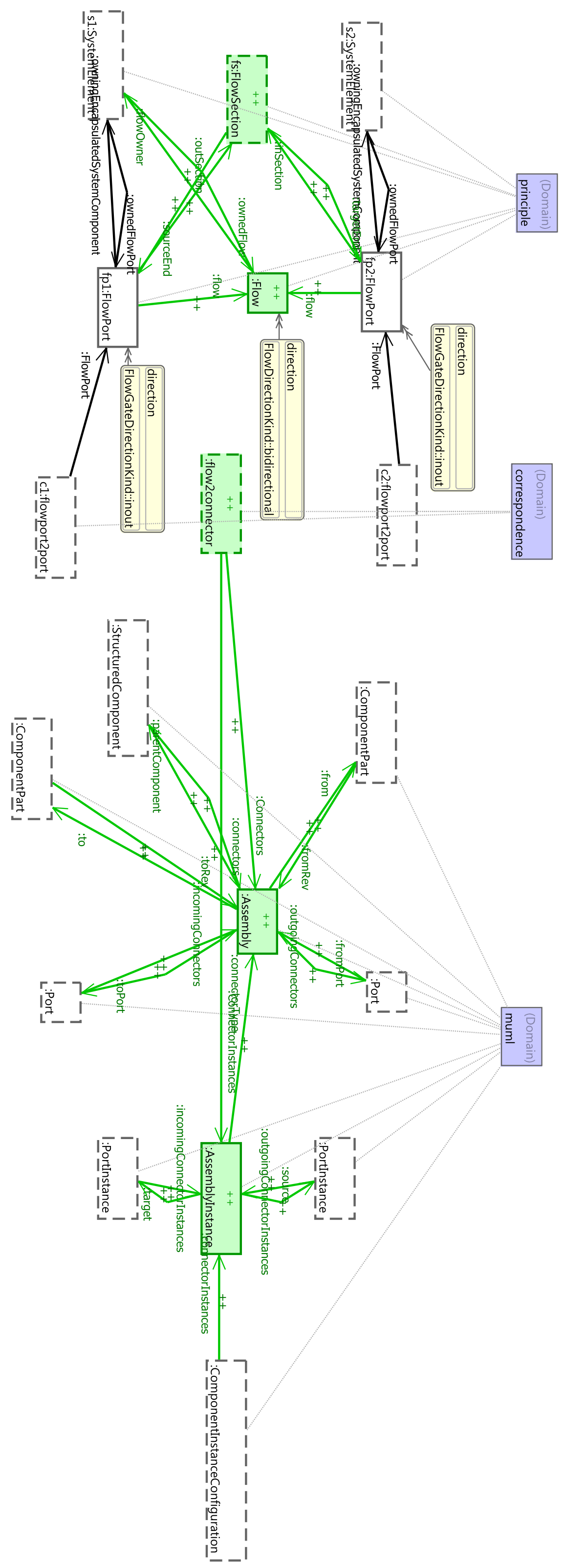


Figure A.11: TGG Rule BidirectionalFlow2Assemblies

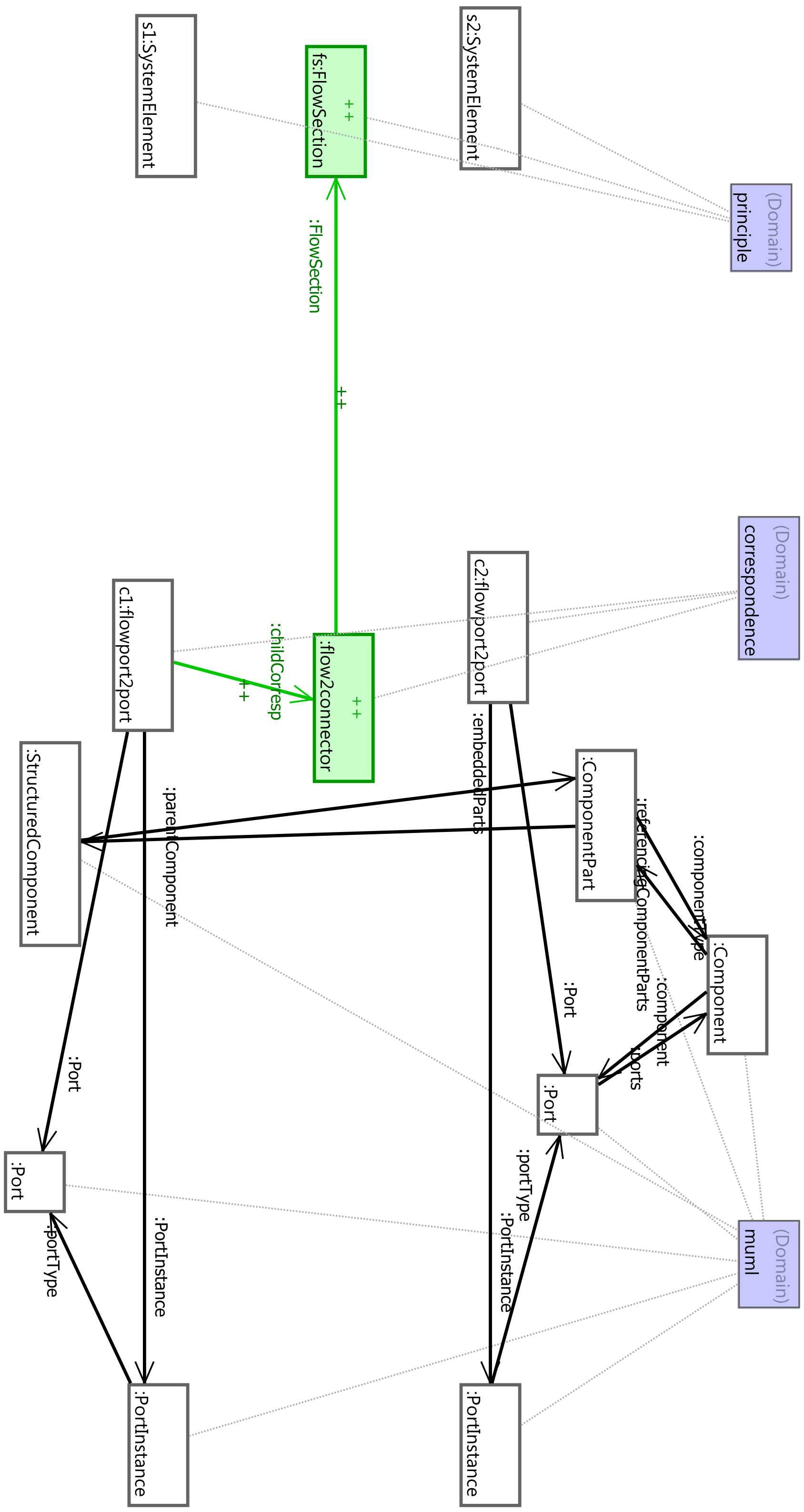


Figure A.12: TGG Rule Flow2Connector

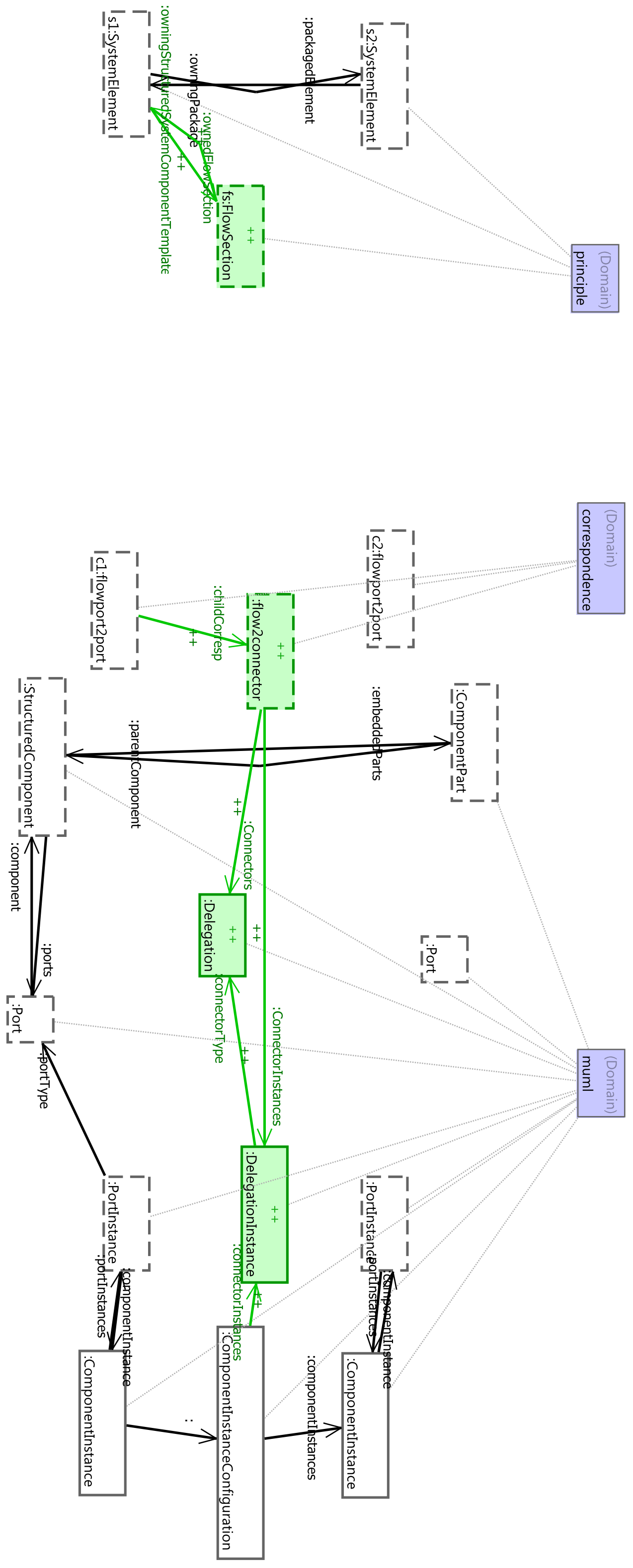


Figure A.13: TGG Rule Flow2Delegation

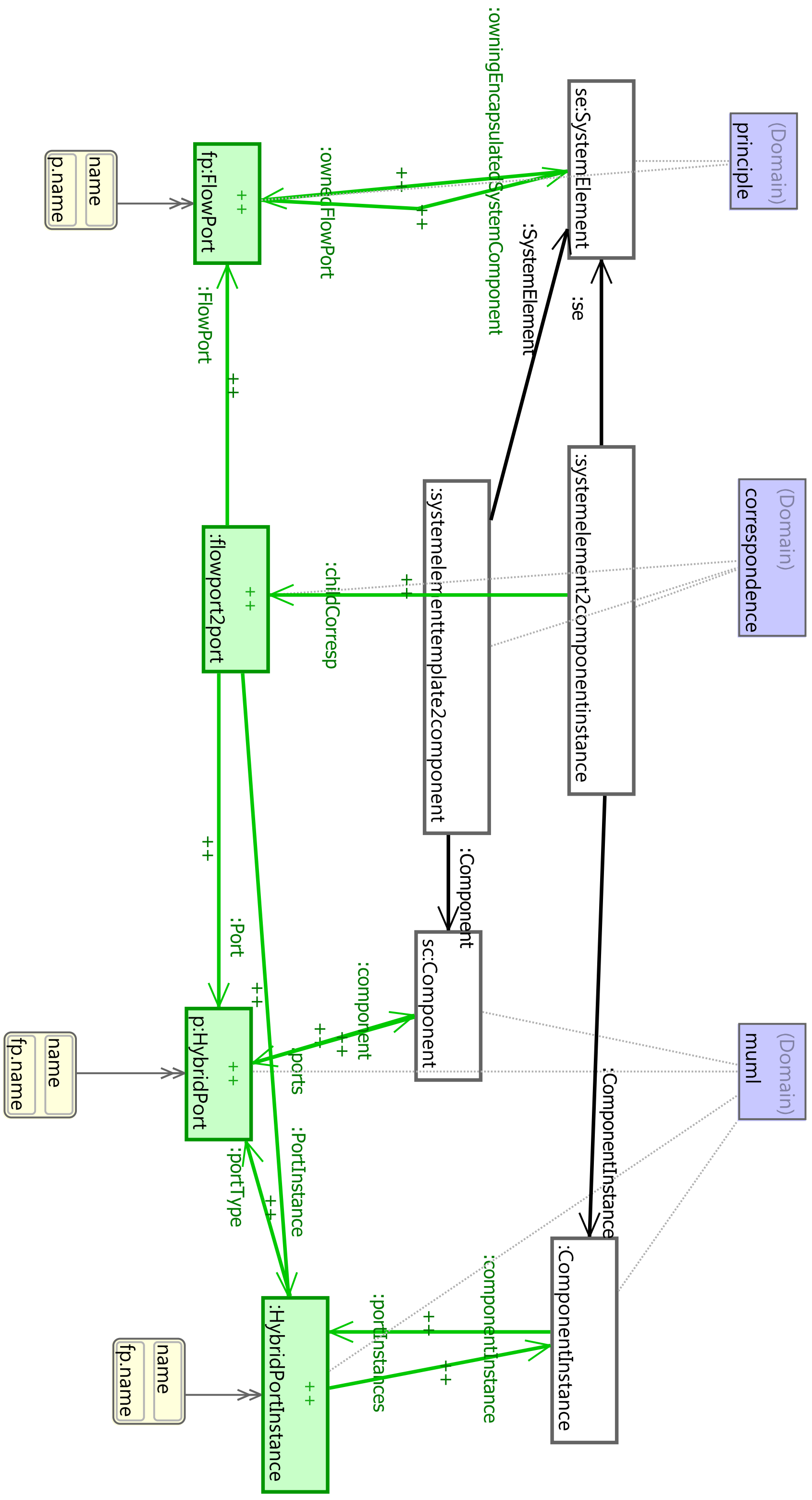


Figure A.14: TGG Rule FlowPort2Port

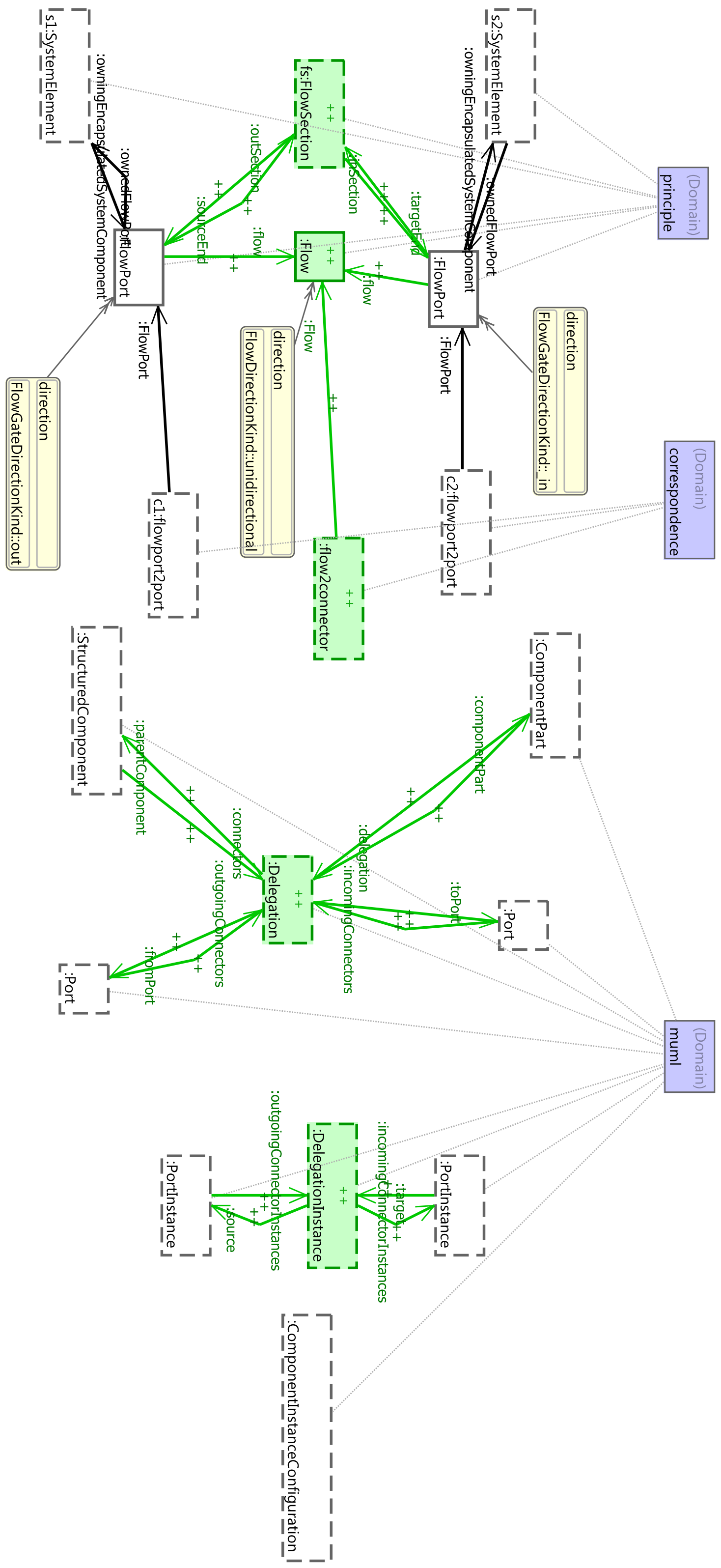


Figure A.15: TGG Rule InFlow2InDelegation

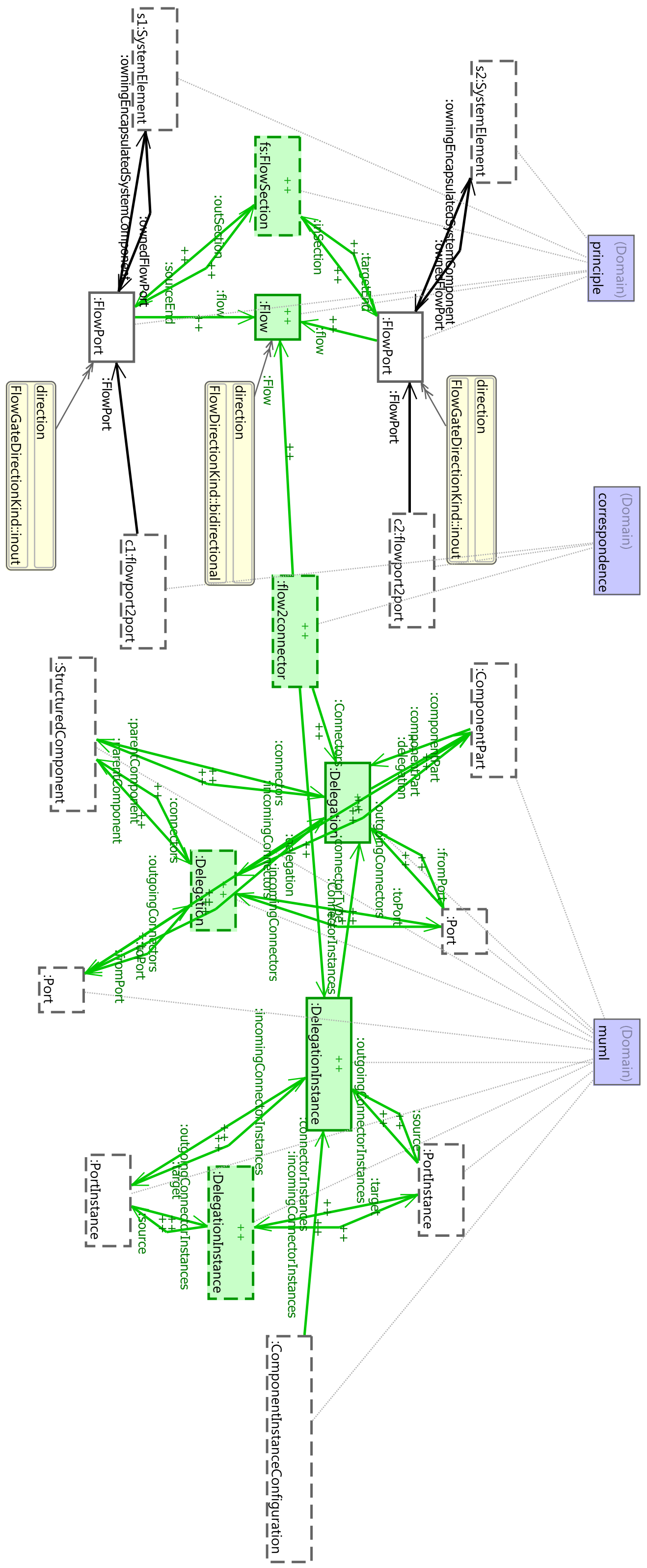


Figure A.16: TGG Rule InOutFlow2InOutDelegation

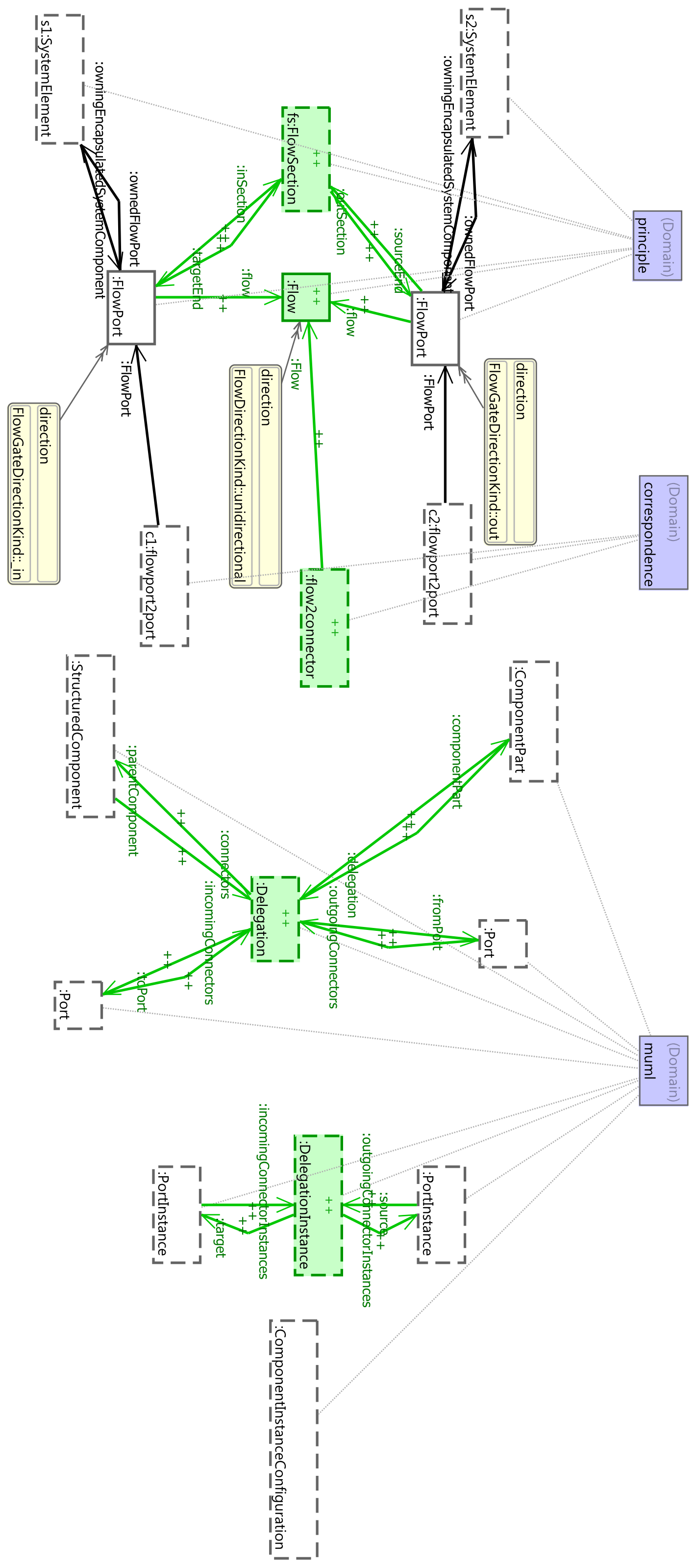


Figure A.17: TGG Rule OutFlow2OutDelegation

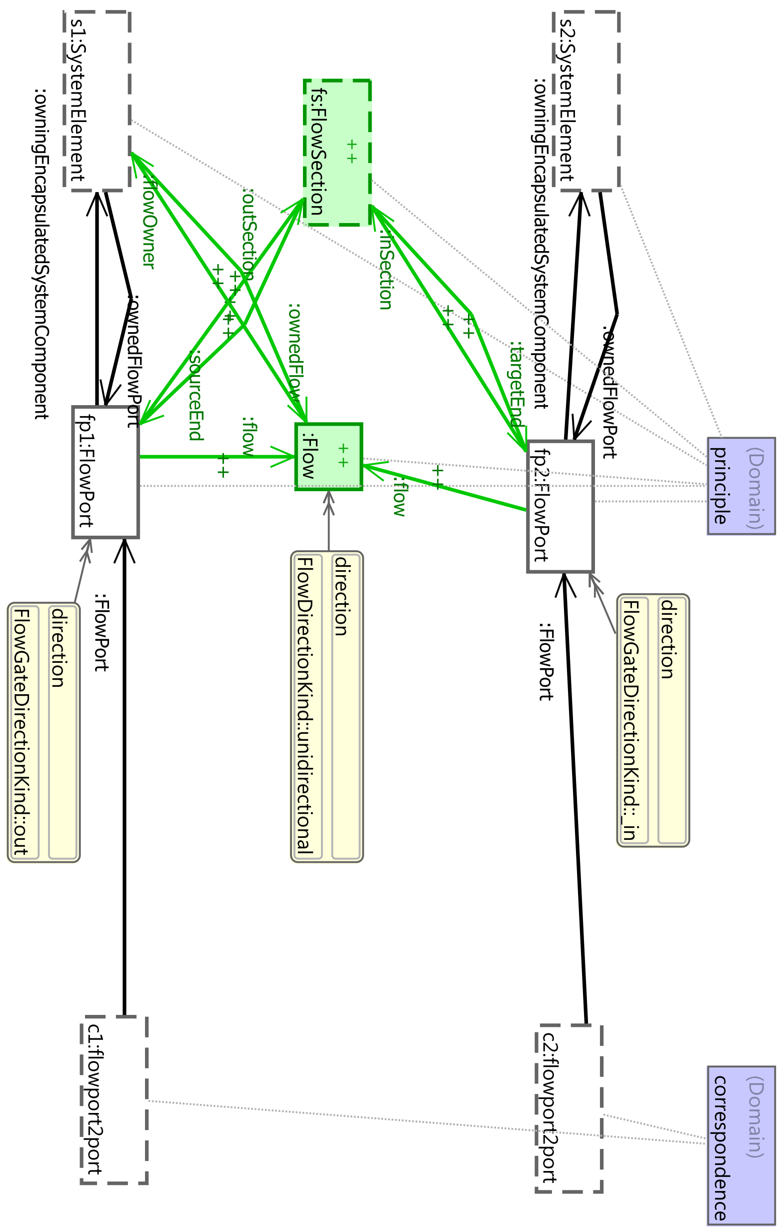


Figure A.18: TGG Rule UnidirectionalFlow2Assembly

APPENDIX B

Transformation from Refinement Rules to TGG Refinements

This TGG ruleset implements the transformation from refinement rules (specified using Story Diagrams) to Story Diagrams that modify an existing TGG ruleset in order to reflect these refinement rules. The input is a set of refinement rules typed by a language metamodel (typically the metamodel of the target language). The output is story diagram rules typed by the TGG metamodel. These rules are then applied to an existing TGG ruleset.

For details on the approach, see Sect. 4.2. Details on the implementation can be found in Sect. 6.1.5.

In principle, these rules perform a 1-to-1 mapping for all constructs of the Story Diagram language. As the TGG ruleset is typed over the target (and source) language, and we want the Story Diagrams to modify this ruleset, we have to adjust the story diagram such that they operate on the metamodel of the TGG. To do so, some of the rules (those that deal with types) implement a more complex mapping; these rules are the rules in Fig. B.7, B.8, B.9, B.10, B.11, B.12, B.13, B.14, and B.15.[1]

This ruleset makes use of rule inheritance: Rules Variable2Variable-contextNode (Fig. B.9), Variable2Variable-producedNode-create (Fig. B.10), and Variable2Variable-producedNode-destroy (Fig. B.11) all inherit from the (abstract) rule Variable2Variable (Fig. B.8).

[1]As this transformation cannot be run backwards, some of these rules lack bidirectionality.

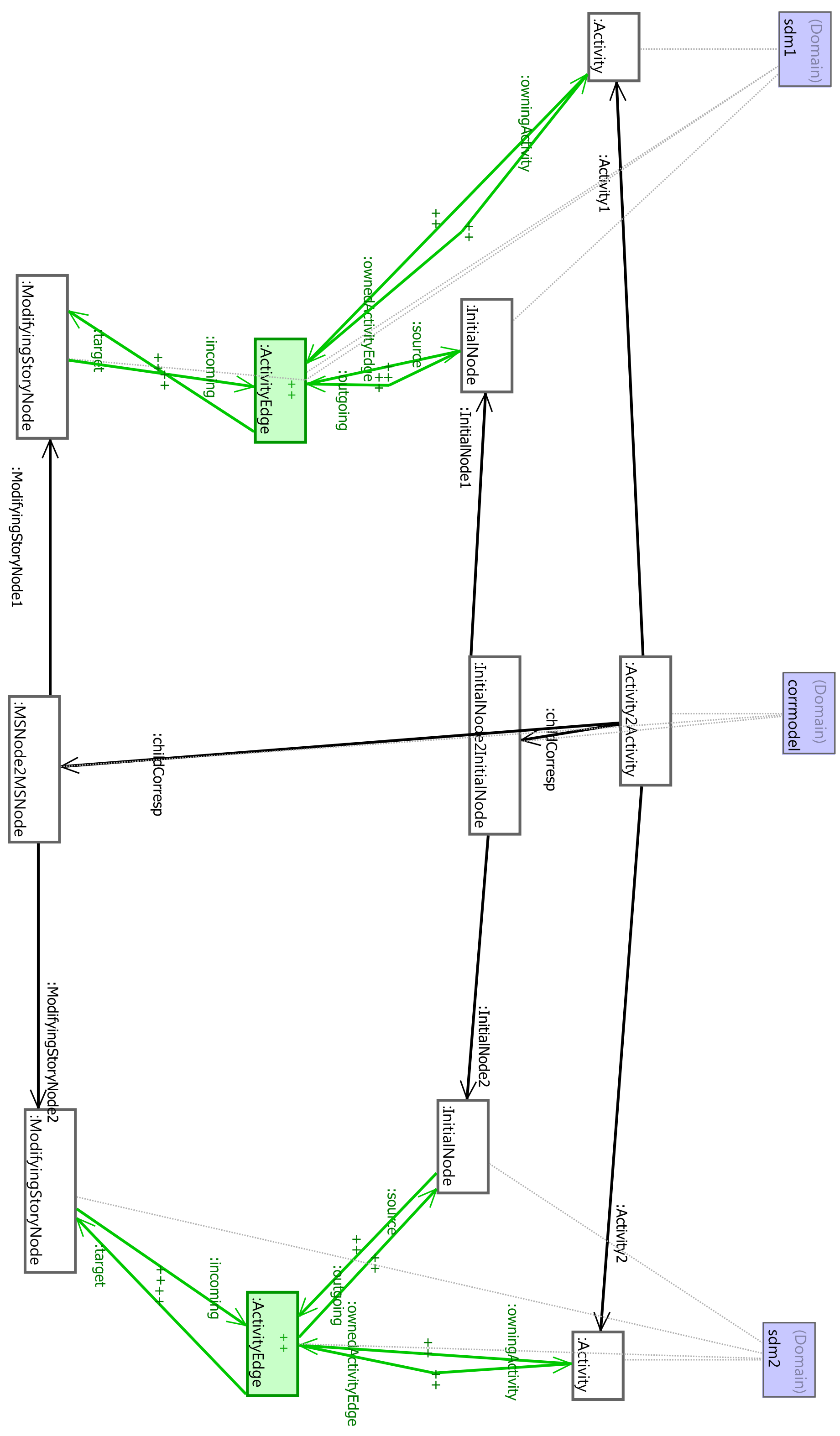


Figure B.1: TGG Rule ActivityEdge2ActivityEdge-1

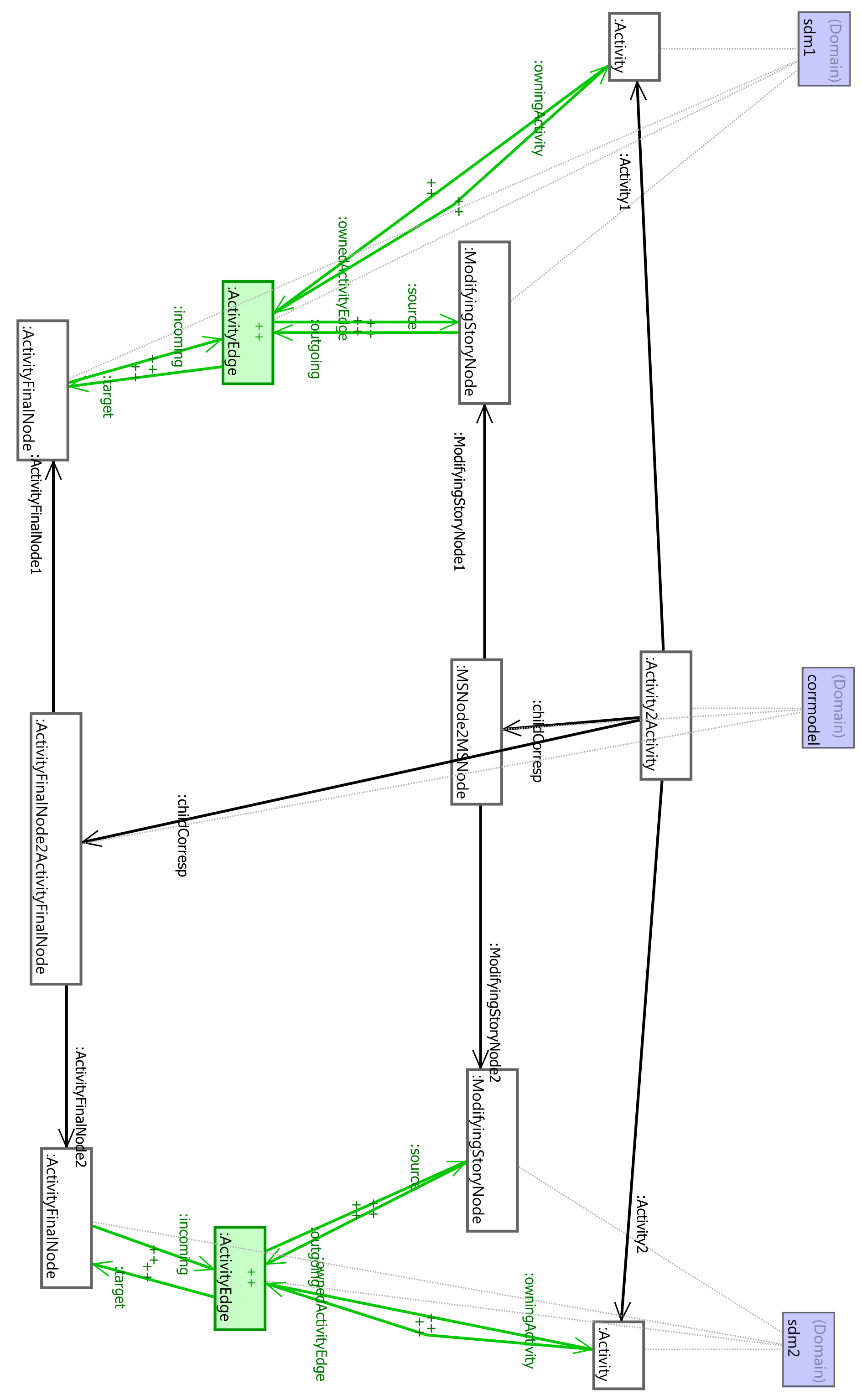


Figure B.2: TGG Rule ActivityEdge2ActivityEdge-2

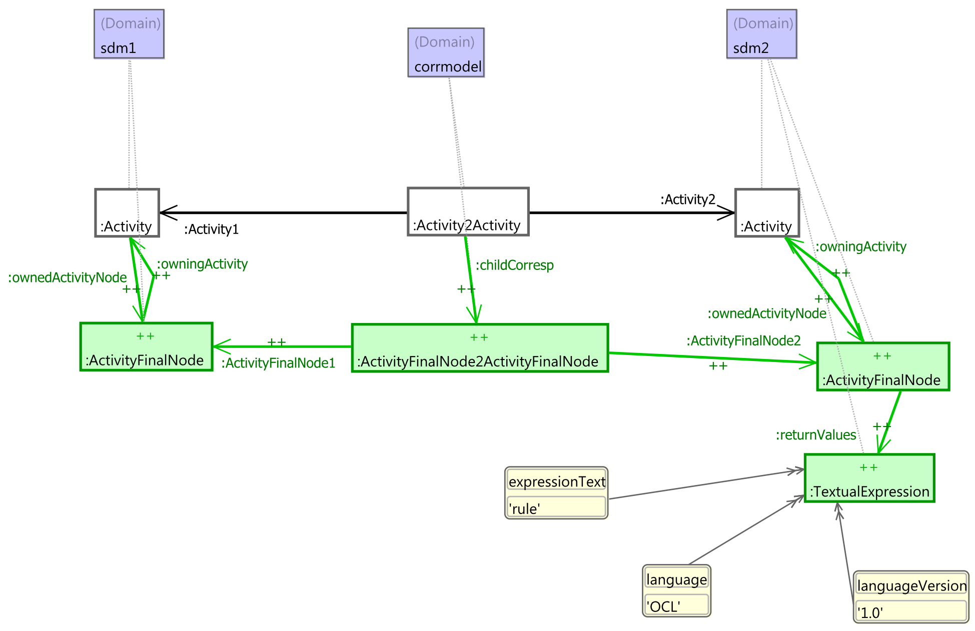


Figure B.3: TGG Rule FinalNode2FinalNode

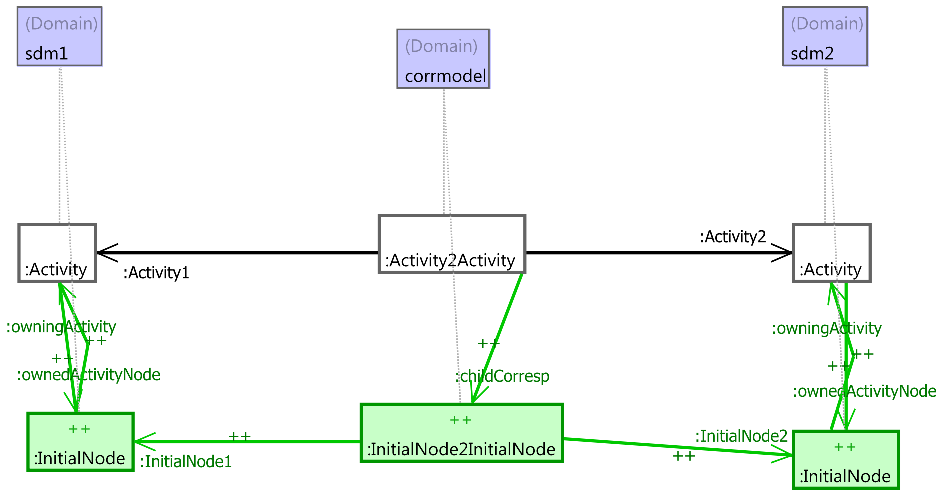


Figure B.4: TGG Rule InitialNode2InitialNode

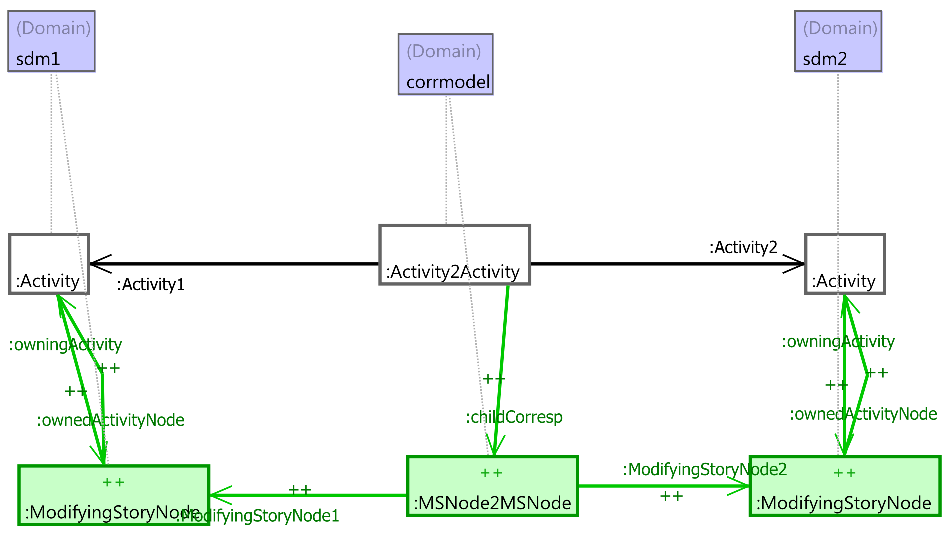


Figure B.5: TGG Rule MSNode2MSNode

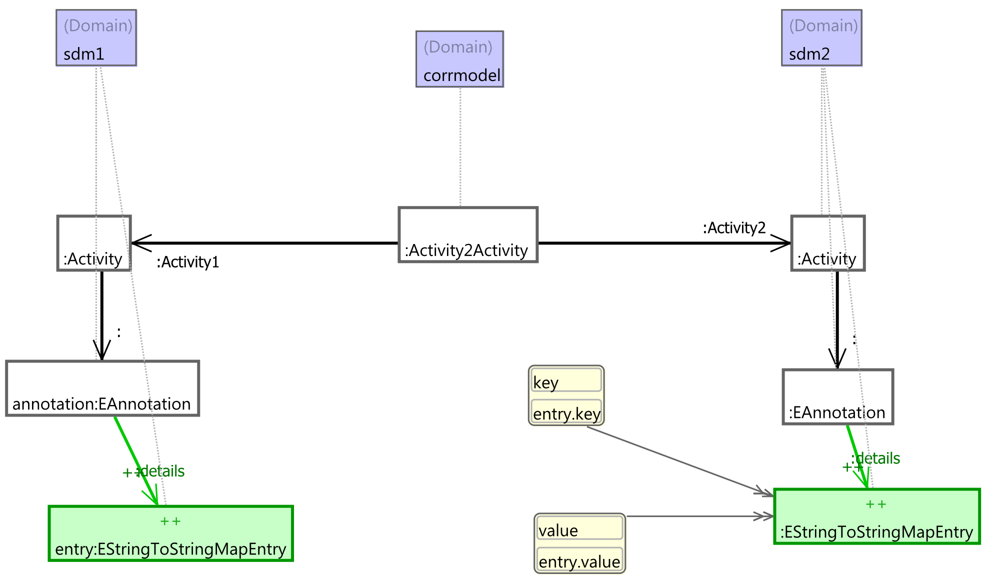


Figure B.6: TGG Rule Type2Type

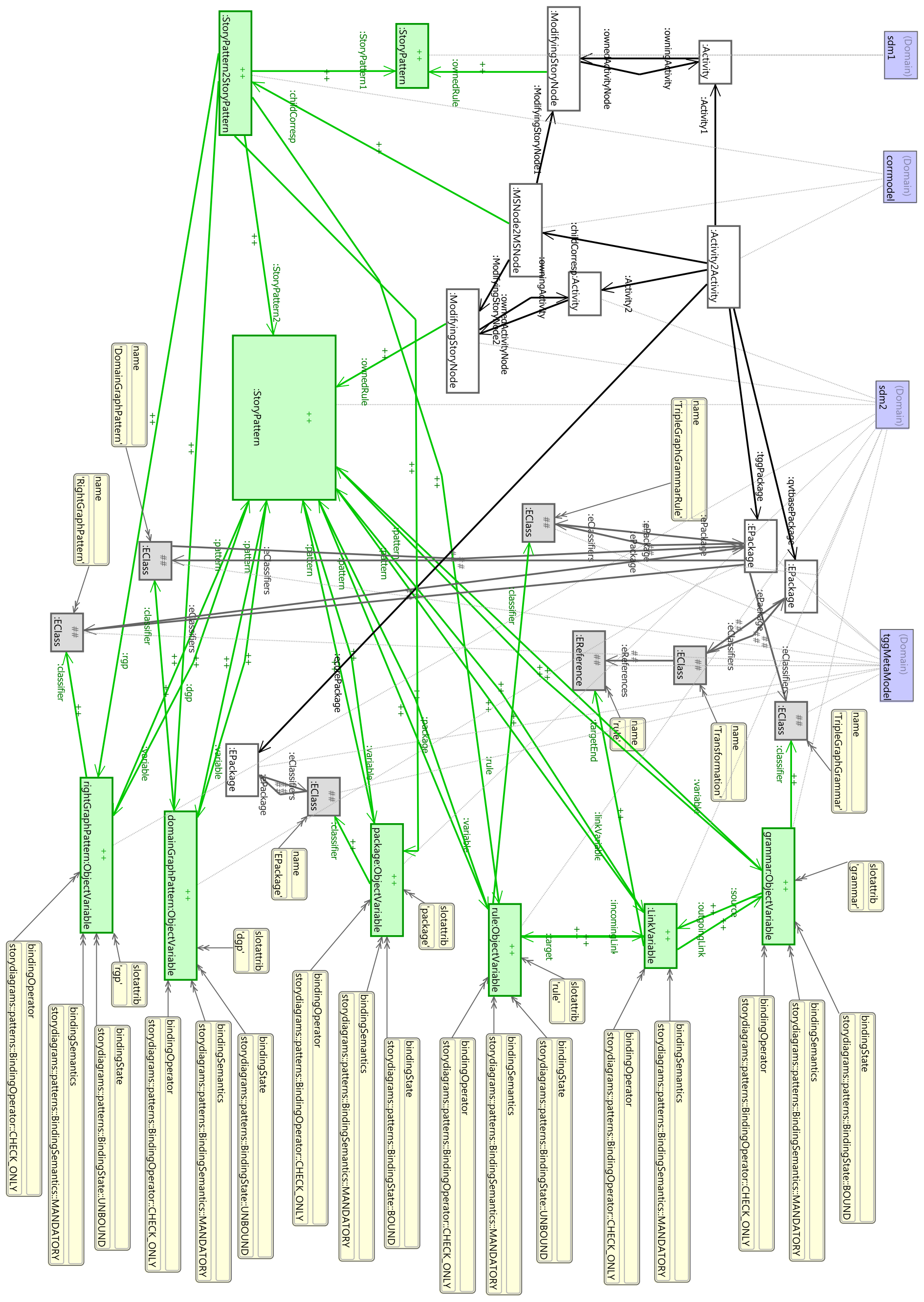


Figure B.7: TGG Rule StoryPattern2StoryPattern

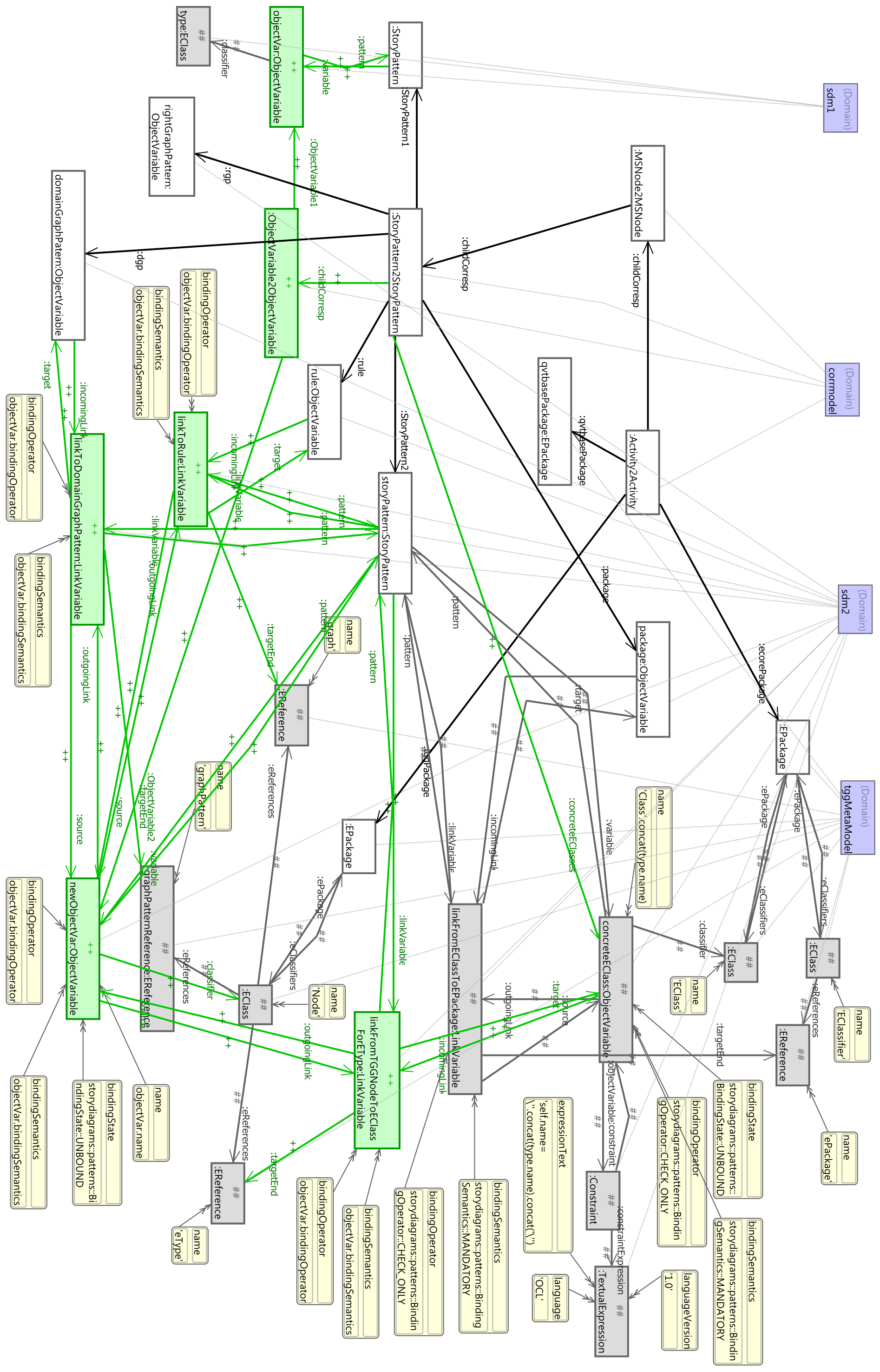

(Domain) sdm1
(Domain) corrmodel
(Domain) sdm2
(Domain) tggMetaModel
typeEClass
objectVar:ObjectVariable
:StoryPattern
rightGraphPattern: ObjectVariable
:MSNode2MSNode
domainGraphPattern:ObjectVariable
:ObjectVariable2ObjectVariable
:StoryPattern2StoryPattern
rule:ObjectVariable
qvtbasePackage:EPackage
:Activity2Activity
linkToRuleLinkVariable
linkToDomainGraphPattern:LinkVariable
storyPattern:StoryPattern
package:ObjectVariable
:EReference
:EPackage
graphPatternReference:EReference
newObjectVar:ObjectVariable
:EClass
linkFromEClassToEPackage:LinkVariable
concreteEClass:ObjectVariable
linkFromTGGNodeToEClass ForEType:LinkVariable
:Constraint
:TextualExpression
bindingOperator
objectVar.bindingOperator
bindingSemantics
objectVar.bindingSemantics
name
'Class'.concat(type.name)
'graphPattern'
'Node'
'EClass'
'EClassifier'
'ePackage'
'eType'
'OCL'
'1.0'
expressionText
'self.name=
\''.concat(type.name).concat('\'')'
bindingState
storydiagrams::patterns::BindingState::UNBOUND
storydiagrams::patterns::BindingOperator::CHECK_ONLY
storydiagrams::patterns::BindingSemantics::MANDATORY

Figure B.8: TGG Rule Variable2Variable

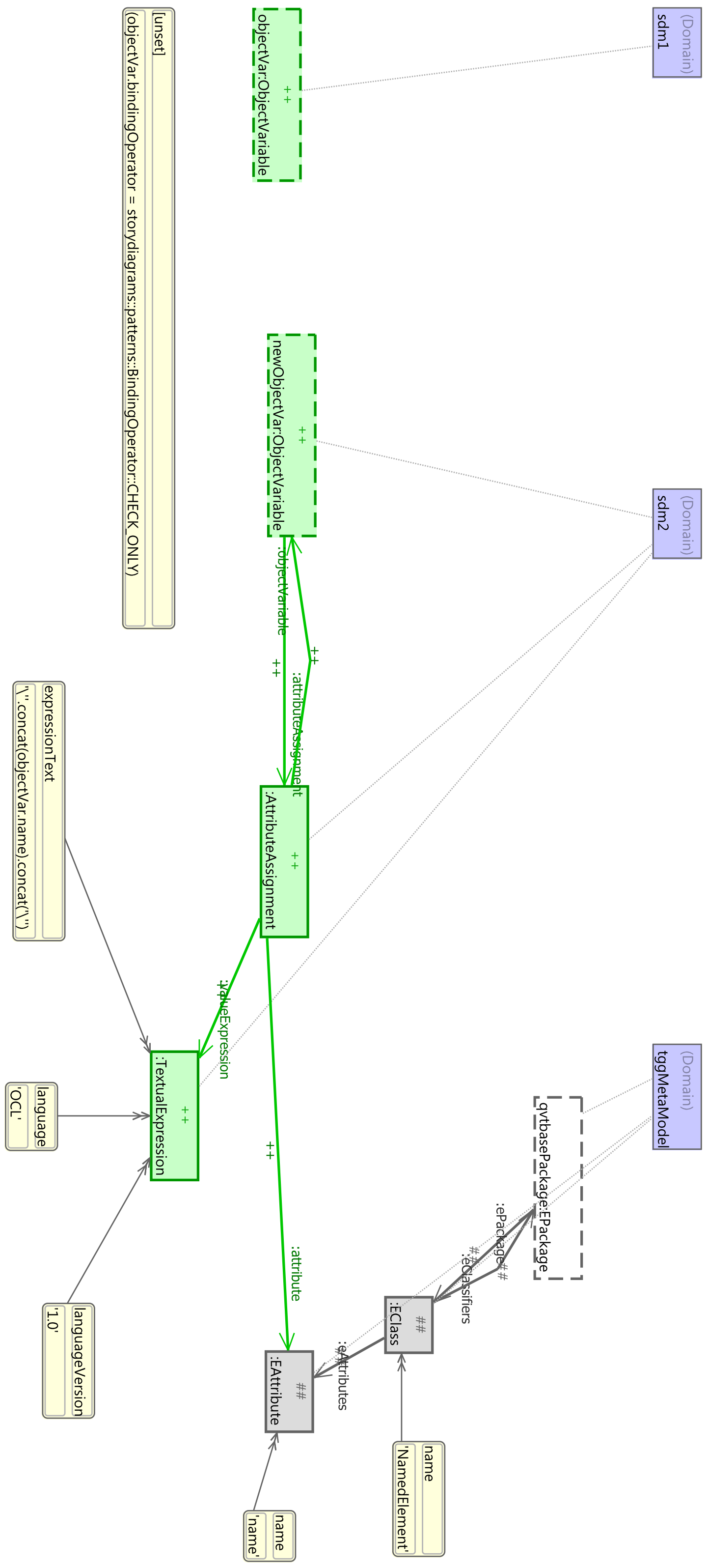


Figure B.9: TGG Rule Variable2Variable-contextNode

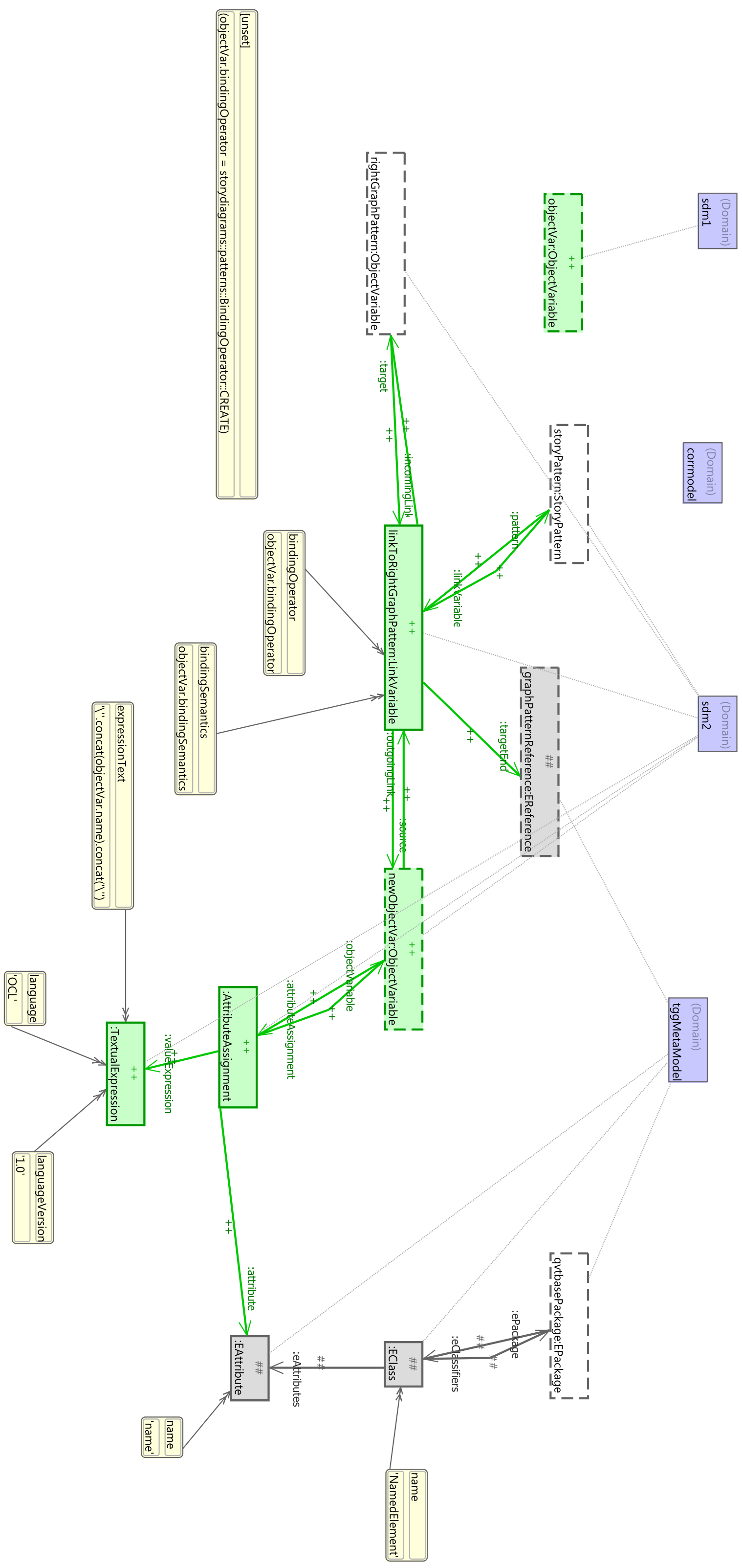


Figure B.10: TGG Rule Variable2Variable-producedNode-create

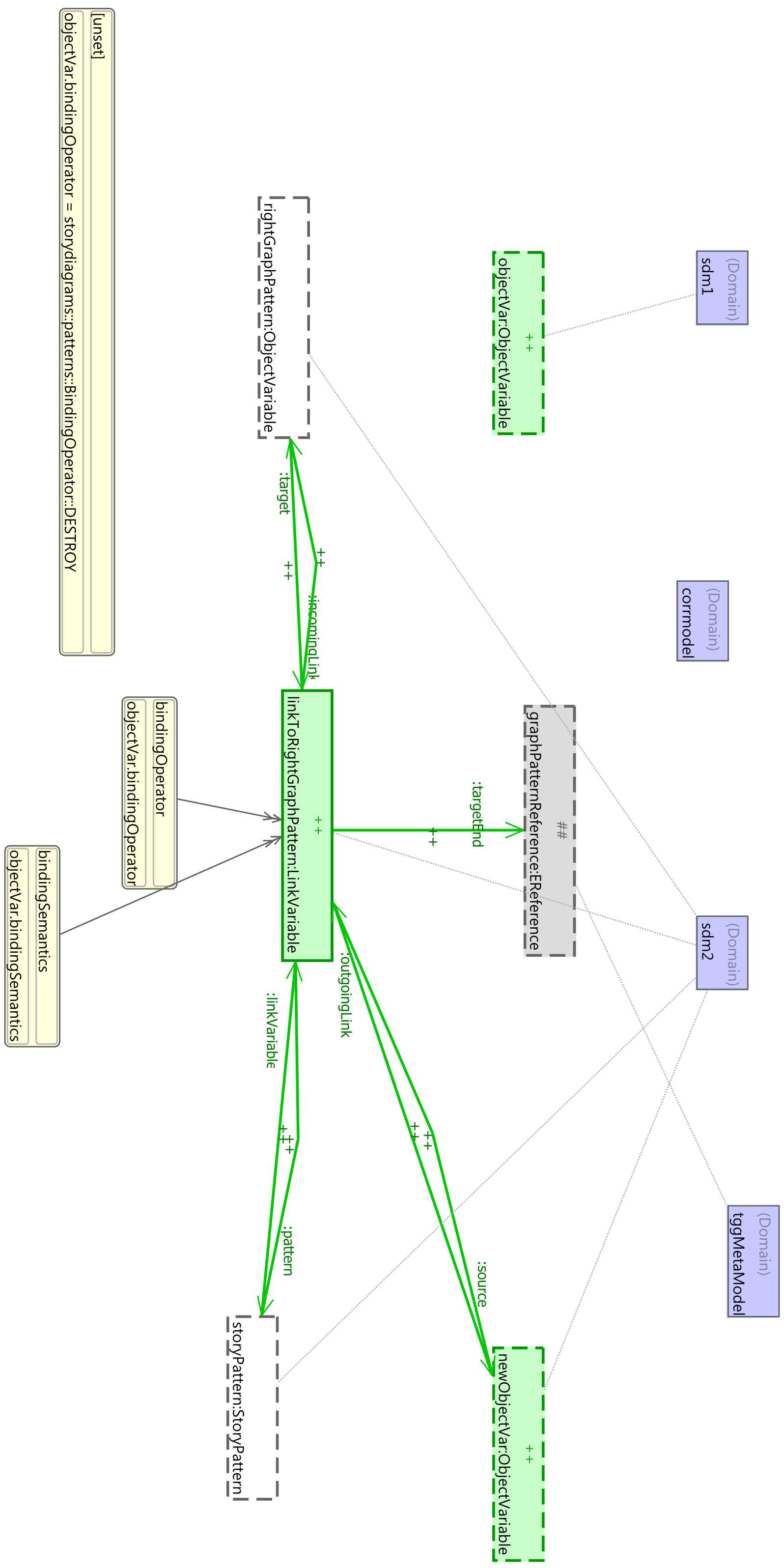

[unset]
objectVar.bindingOperator = storydiagrams::patterns::BindingOperator::DESTROY
rightGraphPattern:ObjectVariable
objectVar:ObjectVariable
++
(Domain)
sdm1
:target
++
++
:incomingLink
(Domain)
corrmodel
bindingOperator
objectVar.bindingOperator
linkToRightGraphPattern:LinkVariable
++
:targetEnd
++
graphPatternReference:EReference
##
bindingSemantics
objectVar.bindingSemantics
(Domain)
sdm2
:linkVariable
:outgoingLink
++
++
++
++
:pattern
:source
(Domain)
tggMetaModel
storyPattern:StoryPattern
newObjectVar:ObjectVariable
++

Figure B.11: TGG Rule Variable2Variable-producedNode-destroy

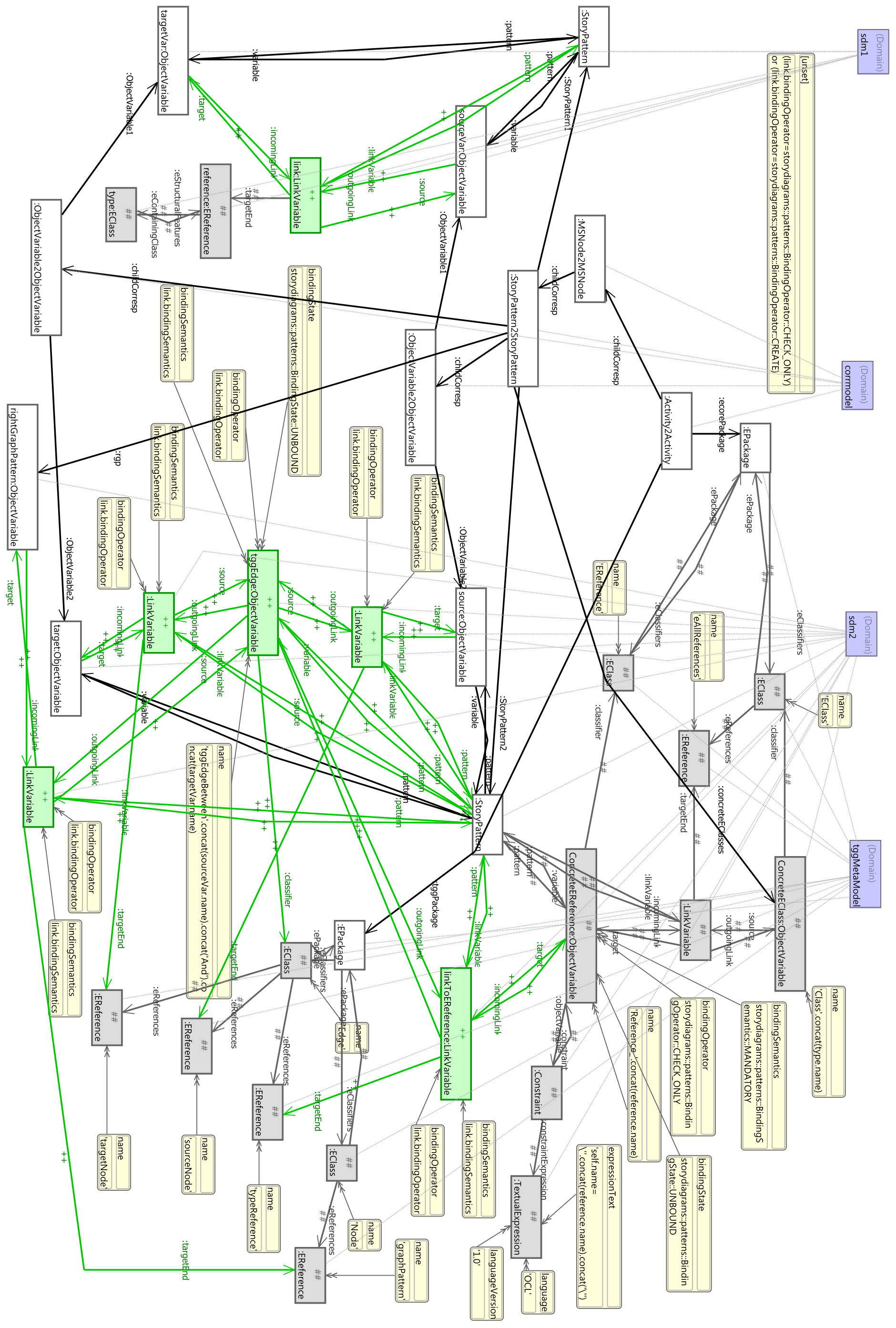

(Domain) sdm1
(Domain) corrmodel
(Domain) sdm2
(Domain) tggMetaModel
:StoryPattern
targetVar:ObjectVariable
sourceVar:ObjectVariable
linkLinkVariable
reference:EReference
typeEClass
:MSNode2MSNode
:StoryPattern2StoryPattern
:ObjectVariable2ObjectVariable
:Activity2Activity
:EPackage
:EClass
:EReference
rightGraphPattern:ObjectVariable
tggEdge:ObjectVariable
target:ObjectVariable
source:ObjectVariable
:LinkVariable
linkToReference:LinkVariable
ConcreteEReference:ObjectVariable
ConcreteEClass:ObjectVariable
:Constraint
:TextualExpression
bindingState
bindingOperator
bindingSemantics
link.bindingOperator
link.bindingSemantics
storydiagrams::patterns::BindingState::UNBOUND
storydiagrams::patterns::BindingOperator::CHECK_ONLY
storydiagrams::patterns::BindingSemantics::MANDATORY
[unset]
(link.bindingOperator=storydiagrams::patterns::BindingOperator::CHECK_ONLY)
or (link.bindingOperator=storydiagrams::patterns::BindingOperator::CREATE)
'tggEdgeBetween'.concat(sourceVar.name).concat('And').concat(targetVar.name)
'Reference'.concat(reference.name)
'Class'.concat(type.name)
'self.name='.concat(reference.name).concat('')
'OCL'
'1.0'
'sourceNode'
'targetNode'
'typeReference'
'graphPattern'
'Node'
'Edge'
'EClass'
'eAllReferences'
'EReference'

Figure B.12: TGG Rule LinkVariable2Edge-checkonly-create

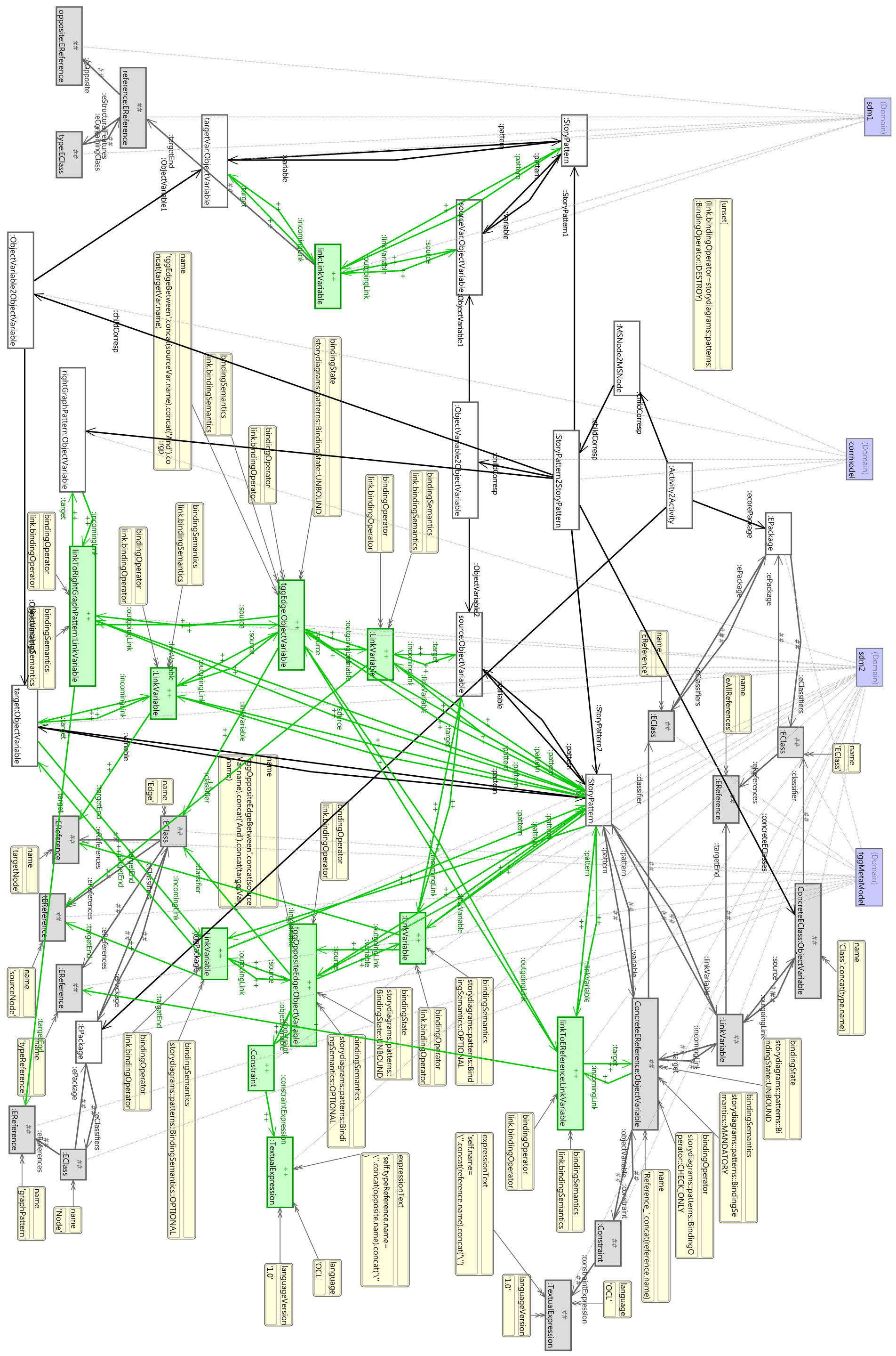


Figure B.13: TGG Rule LinkVariable2Edge-destroy

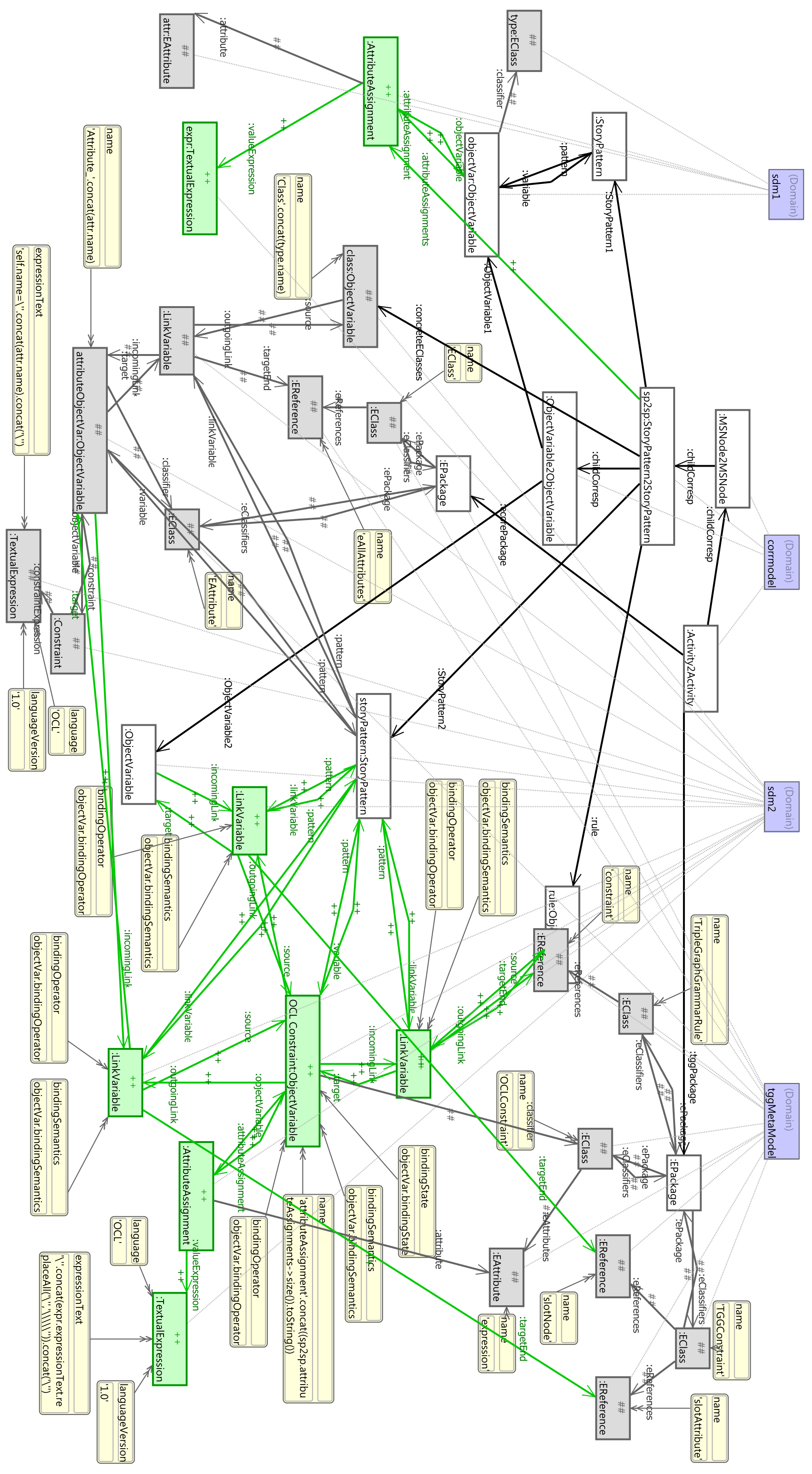


Figure B.14: TGG Rule AttributeAssignment2OCLConstraint

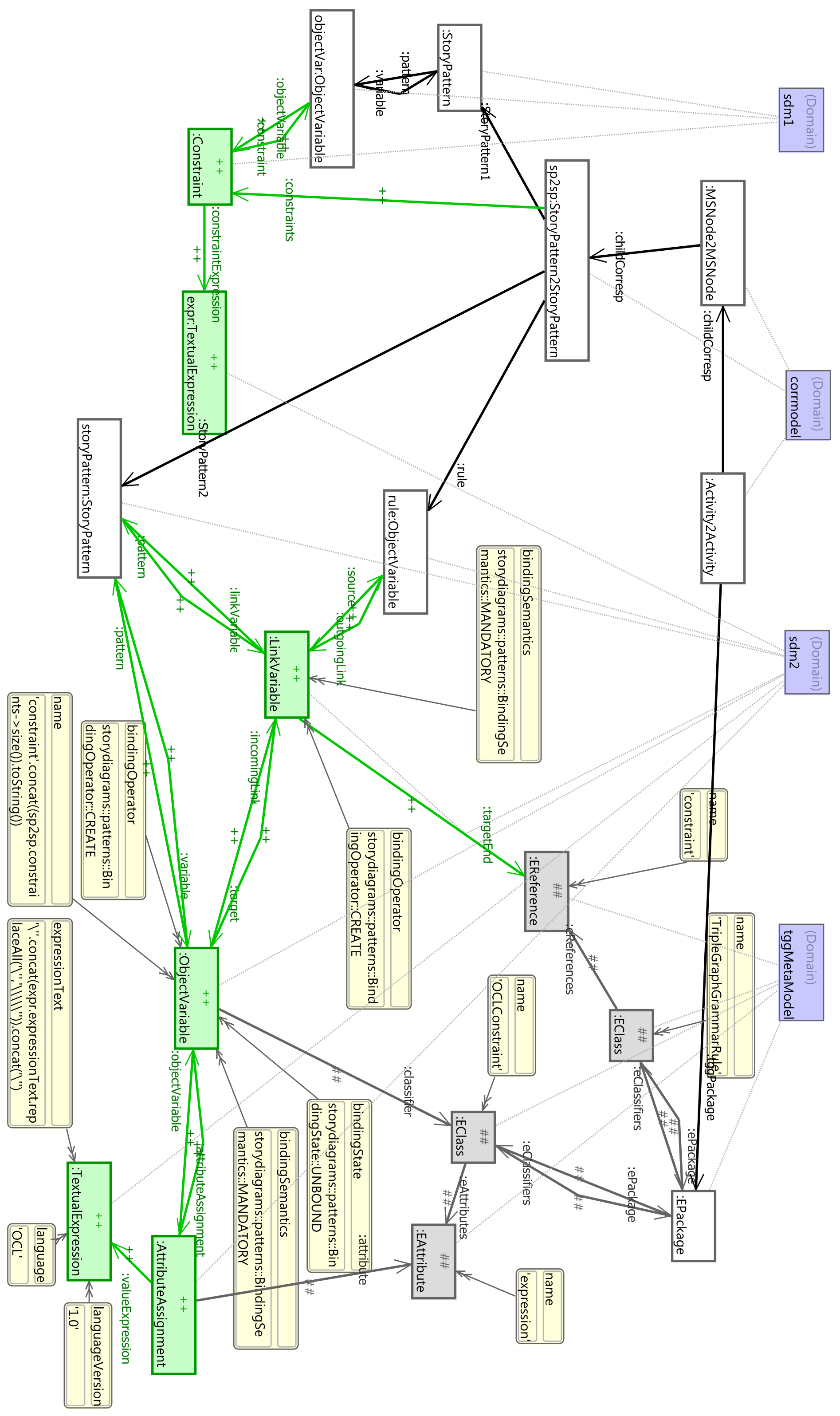


Figure B.15: TGG Rule Constraint2Constraint

Bibliography

[ABD^{+}14] H. Anacker, C. Brenner, R. Dorociak, R. Dumitrescu, J. Gausemeier, P. Iwanek, W. Schäfer, and M. Vaßholz. Methods for the domain-spanning conceptual design. In J. Gausemeier, F. J. Rammig, and W. Schäfer (editors), *Design Methodology for Intelligent Technical Systems*, Lecture Notes in Mechanical Engineering, pp. 117–182. Springer Berlin Heidelberg, 2014.

[Ack10] P. Ackermann. *Debugging von Modelltransformationen*. Bachelor thesis, University of Paderborn, June 2010.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADF^{+}14] H. Anacker, M. Dellnitz, K. Flaßkamp, S. Groesbrink, P. Hartmann, C. Heinzemann, C. Horenkamp, B. Kleinjohann, L. Kleinjohann, S. Korf, M. Krüger, W. Müller, S. Ober-Blöbaum, S. Oberthür, M. Porrmann, C. Priesterjahn, R. Radkowski, C. Rasche, J. Rieke, M. Ringkamp, K. Stahl, D. Steenken, J. Stöcklein, R. Timmermann, A. Trächtler, K. Witting, T. Xie, and S. Ziegert. Methods for the design and development. In J. Gausemeier, F. J. Rammig, and W. Schäfer (editors), *Design Methodology for Intelligent Technical Systems*, Lecture Notes in Mechanical Engineering, pp. 183–350. Springer Berlin Heidelberg, 2014.

[ADG^{+}09] P. Adelt, J. Donoth, J. Geisler, S. Henkler, S. Kahl, B. Klöpper, E. Münch, S. Oberthür, C. Paiz, H. Podlogar, M. Porrmann, R. Radkowski, C. Romaus, A. Schmidt, B. Schulz, H. Voecking, U. Witkowski, and K. Witting. *Selbstoptimierende Systeme des Maschinenbaus – Definitionen, Anwendungen, Konzepte*. 234,. HNI Verlagsschriftenreihe, Paderborn, 2009.

[AGL^{+}12] A. Anis, S. Goschin, S. Lehrig, C. Stritzke, and T. Zolynski. PG SafeBots II – Developer documentation, April 2012.

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In A. Rensink and J. Warmer (editors), *Model Driven Architecture – Foundations and Applications: Second European Conference*, vol. 4066 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, Heidelberg, 2006.

[AP96] M. Acar and R. Parkin. Engineering education for mechatronics. *Industrial Electronics, IEEE Transactions on*, 43(1):106–112, feb 1996.

[AP11] K. Altmanninger and A. Pierantonio. A categorization for conflicts in model versioning. *Elektrotechnik und Informationstechnik*, 128(11-12):421–426, 2011.

[AS08] C. Amelunxen and A. Schürr. Formalising model transformation rules for UML/MOF 2. *Software, IET*, 2(3):204–222, 2008.

[ASK10] K. Altmanninger, W. Schwinger, and G. Kotsis. Semantics for accurate conflict detection in smover: Specification, detection and presentation by example. *International Journal of Enterprise Information Systems (IJEIS)*, 6(1):68–84, 2010.

[ASW09] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems (IJWIS)*, 5(3):271–304, 2009.

[AVS12] A. Anjorin, G. Varró, and A. Schürr. Complex attribute manipulation in TGGs with constraint-based programming techniques. In *Proceedings of the First International Workshop on Bidirectional Transformations (BX 2012)*. 2012.

[Bal91] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, pp. 158–165. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991.

[BBB+12] S. Becker, C. Brenner, C. Brink, S. Dziwok, R. Löffler, C. Heinzemann, U. Pohlmann, W. Schäfer, J. Suck, and O. Sudmann. The MechatronicUML design method – process, syntax, and semantics. Tech. Rep. tr-ri-12-326, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Aug. 2012.

[BCE+06] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pp. 5–12. ACM, New York, NY, USA, 2006.

[Ben05] K. Bender (editor). *Embedded Systems – qualitätsorientierte Entwicklung*. Springer, Berlin, Heidelberg, 2005.

[BGO06] S. Burmester, H. Giese, and O. Oberschelp. Hybrid UML components for the design of complex self-optimizing mechatronic systems. In J. Braz, H. Araújo, A. Vieira, and B. Encarnacao (editors), *Informatics in Control, Automation and Robotics I*. Springer, Mar. 2006.

[BKPS07] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb 2007.

[BLS+12] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. The past, present, and future of model versioning. *Emerging Technologies for the Evolution and Maintenance of Software Models, IGI Global*, pp. 410–443, 2012.

[BSWK12] P. Brosch, M. Seidl, M. Wimmer, and G. Kappel. Conflict visualization for evolving UML models. *Journal of Object Technology*, 11(3):2:1–30, Oct. 2012.

[BRD06] Bundesrepublik Deutschland. V-Modell XT 1.4, 2006.

[BW07] T. Baar and J. Whittle. On the usage of concrete syntax in model transformation rules. In I. Virbitskaite and A. Voronkov (editors), *Perspectives of Systems Informatics*, vol. 4378 of *Lecture Notes in Computer Science*, pp. 84–97. Springer Berlin Heidelberg, 2007.

[bx12] *First International Workshop on Bidirectional Transformations (BX 2012). Proceedings*, vol. 49 of *Electronic Communications of the EASST*, 2012.

[bx13] *Second International Workshop on Bidirectional Transformations (BX 2013). Proceedings*, 2013.

[BY03] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg (editors), *Lectures on Concurrency and Petri Nets*, vol. 3098 of *Lecture Notes in Computer Science*, pp. 87–124. Springer, 2003.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches, 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621 –645, 2006.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pp. 151–158. ACM, New York, NY, USA, 1971.

[CREP10] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Jtl: a bidirectional and change propagating transformation language. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE 2010)*. Eindhoven, The Netherlands, 2010.

[DK13] K. Duddy and G. Kappel (editors). *Theory and Practice of Model Transformations – 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, vol. 7909 of *Lecture Notes in Computer Science*. Springer, 2013.

[DMSS10] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha. Debugging model-transformation failures using dynamic tainting. In T. D'Hondt (editor), *Object-Oriented Programming (ECOOP 2010)*, vol. 6183 of *Lecture Notes in Computer Science*, pp. 26–51. Springer Berlin Heidelberg, 2010.

[EC01] S. Easterbrook and M. Chechik. 2nd International Workshop on Living with Inconsistency (IWLWI01). *SIGSOFT Software Engineering Notes*, 26(6):76–78, Nov. 2001.

[EEKR12] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (editors). *Graph Transformations – 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, vol. 7562 of *Lecture Notes in Computer Science*. Springer, 2012.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[Eis97] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, Apr. 1997.

[ELHN⁺10] A. Egyed, R. E. Lopez-Herrejon, B. Nuseibeh, G. Botterweck, M. Chechik, and Z. Hu (editors). *3rd Workshop on Living with Inconsistencies in Software Development (LWI 2010). Proceedings*. 2010.

[Epp95] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pp. 632–640. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995.

[EPT04] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg (editors), *Graph Transformations*, vol. 3256 of *Lecture Notes in Computer Science*, pp. 161–177. Springer Berlin Heidelberg, 2004.

[ERD⁺97] J. Engelfriet, G. Rozenberg, F. Drewes, H.-J. Kreowski, A. Habel, A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe, M. Korff, L. Ribeiro, A. Wagner, A. Corradini, B. Courcelle, A. Ehrenfeucht, T. Harju, and A. Schürr. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, vol. 1. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[FGYO95] S. M. Fohn, A. Greef, R. E. Young, and P. O'Grady. Concurrent engineering. In H. H. Adelsberger, J. Lažanský, and V. Mařík (editors), *Information Management in Computer Integrated Manufacturing*, vol. 973 of *Lecture Notes in Computer Science*, pp. 493–505. Springer Berlin Heidelberg, 1995.

[FM92] K. Forsberg and H. Mooz. The relationship of systems engineering to the project cycle. *Engineering Management Journal*, 4:36–43, 1992.

[Fow03] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[Fra06] U. Frank. *Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme*. Ph.D. thesis, University of Paderborn, 2006.

[GCW+13] J. Gausemeier, A. M. Czaja, O. Wiederkehr, R. Dumitrescu, C. Tschirner, and D. Steffen. Studie: Systems Engineering in der industriellen Praxis. In J. Gausemeier, R. Dumitrescu, F. Rammig, W. Schäfer, and A. Trächtler (editors), *9. Paderborner Workshop Entwurf mechatronischer Systeme*, vol. 310 of *HNI-Verlagsschriftenreihe*, pp. 431–445. Heinz Nixdorf Institut, Paderborn, 2013.

[Gei08] L. Geiger. Model level debugging with Fujaba. In *Proceedings of the 6th International Fujaba Days*, pp. 23–28. Dresden, Germany, 2008.

[GFDK09] J. Gausemeier, U. Frank, J. Donoth, and S. Kahl. Specification technique for the description of self-optimizing mechatronic systems. *Research in Engineering Design*, 20:201–223, 2009.

[GH09] H. Giese and S. Hildebrandt. Efficient model synchronization of large-scale models. Tech. Rep. 28, Hasso Plattner Institute at the University of Potsdam, 2009.

[GHL14] H. Giese, S. Hildebrandt, and L. Lambers. Bridging the gap between formal semantics and implementation of triple graph grammars. *Software & Systems Modeling*, 13(1):273–299, 2014.

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GK10] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with triple graph grammars. *Software and Systems Modeling*, 9:21–46, 2010. 10.1007/s10270-009-0121-8.

[GKLE11] C. Gerth, J. Küster, M. Luckey, and G. Engels. Detection and resolution of conflicting change operations in version management of process models. *Software and Systems Modeling*, pp. 1–19, December 2011.

[GKP+14] J. Gausemeier, S. Korf, M. Porrmann, K. Stahl, O. Sudmann, and M. Vaßholz. Development of self-optimizing systems. In J. Gausemeier, F. J. Rammig, and W. Schäfer (editors), *Design Methodology for Intelligent Technical Systems*, Lecture Notes in Mechanical Engineering, pp. 65–115. Springer Berlin Heidelberg, 2014.

[GKRT08] J. Greenyer, E. Kindler, J. Rieke, and O. Travkin. TGGs for transforming UML to CSP: Contribution to the AGTIVE 2007 graph transformation tools contest. Tech. Rep. tr-ri-08-287, Software Engineering Group, Department of Computer Science, University of Paderborn, 2008.

[GL12] H. Giese and L. Lambers. Towards automatic verification of behavior preservation for model transformation via invariant checking. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (editors), *Graph Transformations*, vol. 7562 of *Lecture Notes in Computer Science*, pp. 249–263. Springer Berlin Heidelberg, 2012.

[GLO09] E. Guerra, J. Lara, and F. Orejas. Pattern-based model-to-model transformation: Handling attribute conditions. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pp. 83–99. Springer-Verlag, Berlin, Heidelberg, 2009.

[GMP09] R. Grønmo and B. Møller-Pedersen. Concrete syntax-based graph transformation. Research Report 389 389, Department of Informatics, University of Oslo, Norway, Oslo, Norway, 2009.

[GMT99] M. Goedicke, T. Meyer, and G. Taentzer. Viewpoint-oriented software development by distributed graph transformation: towards a basis for living with inconsistencies. In *Requirements Engineering, 1999. Proceedings. IEEE International Symposium on*, pp. 92–99. 1999.

[Gos10] S. Goschin. *Modellsynchronisation bei simultanen Änderungen.* Bachelor thesis, University of Paderborn, October 2010.

[GPR11] J. Greenyer, S. Pook, and J. Rieke. Preventing information loss in incremental model synchronization by reusing elements. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011).* 2011.

[Grø09] R. Grønmo. *Using Concrete Syntax in Graph-based Model Transformations.* Ph.D. thesis, Faculty of Mathematics and Natural Sciences at the University of Oslo, 2009.

[GR12] J. Greenyer and J. Rieke. Applying advanced TGG concepts for a complex transformation of sequence diagram specifications to timed game automata. In A. Schürr, D. Varró, and G. Varró (editors), *Applications of Graph Transformations with Industrial Relevance*, vol. 7233 of *Lecture Notes in Computer Science*, pp. 222–237. Springer Berlin Heidelberg, 2012.

[Gre06] J. Greenyer. *A Study of Model Transformation Technologies: Reconciling TGGs with QVT.* Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, 2006.

[Gri03] K. Grimm. Software technology in an automotive company: major challenges. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pp. 498–503. IEEE Computer Society, Washington, DC, USA, 2003.

[GRR13] P. V. Gorp, T. Ritter, and L. M. Rose (editors). *Modelling Foundations and Applications – 9th European Conference, ECMFA 2013, Montpellier, France, July 1-5, 2013. Proceedings*, vol. 7949 of *Lecture Notes in Computer Science.* Springer, 2013.

[GRS14a] J. Gausemeier, F. J. Rammig, and W. Schäfer (editors). *Design Methodology for Intelligent Technical Systems – Develop Intelligent Technical Systems of the Future*. No. XVIII in Lecture Notes in Mechanical Engineering. Springer Berlin Heidelberg, 2014.

[GRS14b] J. Gausemeier, F.-J. Rammig, and W. Schäfer (editors). *Design Methodology for Intelligent Technical Systems – Develop Intelligent Technical Systems of the Future*. Lecture Notes in Mechanical Engineering. Springer-Verlag, 2014.

[GRSS14] J. Gausemeier, F. J. Rammig, W. Schäfer, and W. Sextro (editors). *Dependability of Self-Optimizing Mechatronic Systems*. No. XVI in Lecture Notes in Mechanical Engineering. Springer Berlin Heidelberg, 2014.

[GSG+09] J. Gausemeier, W. Schäfer, J. Greenyer, S. Kahl, S. Pook, and J. Rieke. Management of cross-domain model consistency during the development of advanced mechatronic systems. In M. N. Bergendahl, M. Grimheden, and L. Leifer (editors), *Proceedings of the 17th International Conference on Engineering Design (ICED'09)*, vol. 6. 2009.

[GST14] J. Gausemeier, W. Schäfer, and A. Trächtler (editors). *Semantische Technologien im Entwurf mechatronischer Systeme – Effektiver Austausch von Lösungswissen in Branchenwertschöpfungsketten*. Carl Hanser Verlag, München, 2014.

[GU08] T. Goldschmidt and A. Uhl. Retainment rules for model transformations. In *1st International Workshop on Model Co-Evolution and Consistency Management*. 2008.

[GU11] T. Goldschmidt and A. Uhl. A formal framework for retainment patterns for trace-based model transformations. *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, 0:91–99, 2011.

[GW06] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio (editors), *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, vol. 4199 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, Genova, Italy, 2006.

[GW09] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8:21–43, 2009.

[HB13] C. Heinzemann and S. Becker. Executing reconfigurations in hierarchical component architectures. In *Proceedings of the 16th international ACM Sigsoft symposium on Component based software engineering*, CBSE '13. ACM, New York, NY, USA, Jun. 2013.

[HEGO10] F. Hermann, H. Ehrig, U. Golas, and F. Orejas. Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In J. Bézivin, R. Soley, and A. Vallecillo (editors), *Proceedings of the International Workshop on Model Driven Interoperability (MDI'10)*, MDI '10, pp. 22–31. ACM, New York, NY, USA, 2010.

[HEO$^+$13] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, S. Gottmann, and T. Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software & Systems Modeling*, pp. 1–29, 2013.

[HEOG10] F. Hermann, H. Ehrig, F. Orejas, and U. Golas. Formal analysis of functional behaviour of model transformations based on triple graph grammars. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr (editors), *Proceedings of the International Conference on Graph Transformation (ICGT '10)*, vol. 6372 of *LNCS*, pp. 155–170. SPRINGER, 2010.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Special issue of Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[HKK04] B. Hardung, T. Kölzow, and A. Krüger. Reuse of software in distributed embedded automotive systems. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, pp. 203–210. ACM, New York, NY, USA, 2004.

[HKR$^+$10] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Showing full semantics preservation in model transformation – a comparison of techniques. In S. M. D. Méry (editor), *IFM 2010*, vol. 6396, pp. 183–198. Springer Verlag Berlin-Heidelberg, 2010.

[HLG$^+$12] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick, and I. Richter. Automatic conformance testing of optimized triple graph grammar implementations. In A. Schürr, D. Varró, and G. Varró (editors), *Applications of Graph Transformations with Industrial Relevance*, vol. 7233 of *Lecture Notes in Computer Science*, pp. 238–253. Springer Berlin Heidelberg, 2012.

[HLG$^+$13] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin, and A. Schürr. A survey of triple graph grammar tools. *EC-EASST*, Post-Proceedings of the Second International Workshop on Bidirectional Transformations (BX 2013), 2013.

[HLR06] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. *Model Driven Engineering Languages and Systems*, 2006.

[HLR07] M. Hibberd, M. Lawley, and K. Raymond. Forensic debugging of model transformations. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, pp. 589–604. Nashville, USA, 2007.

[HN98] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(4):335–367, Oct. 1998.

[HPR⁺12] C. Heinzemann, U. Pohlmann, J. Rieke, W. Schäfer, O. Sudmann, and M. Tichy. Generating Simulink and Stateflow models from software specifications. In *Proceedings of the 12h International Design Conference DESIGN 2012*. 2012.

[HRB⁺14] C. Heinzemann, J. Rieke, J. Bröggelwirth, A. Pines, and A. Volk. Translating MechatronicUML models to MATLAB/Simulink and Stateflow. Tech. Rep. tr-ri-14-338, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, May 2014. Version 0.4.

[HRS13] C. Heinzemann, J. Rieke, and W. Schäfer. Simulating self-adaptive component-based systems using MATLAB/Simulink. In *Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO '13)*, pp. 71–80. IEEE Computer Society Press, Sep. 2013.

[HS02] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, Jan. 2002.

[HSST13] C. Heinzemann, O. Sudmann, W. Schäfer, and M. Tichy. A discipline-spanning development process for self-adaptive mechatronic systems. In *Proceedings of the International Conference on Software and System Process (ICSSP) 2013*. 2013.

[IEC96] International Electrotechnical Commission. IEC 60617 – Graphical symbols for diagrams, 1996.

[INCO04] International Council on Systems Engineering. What is systems engineering? `http://www.incose.org/practice/whatissystemseng.aspx`, 2014.

[ISO9899] ISO/IEC 9899:2011. *Information technology – Programming languages – C*, 2011.

[KLKS10] F. Klar, M. Lauder, A. Königs, and A. Schürr. Extended triple graph grammars with efficient and compatible graph translators. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel (editors), *Graph Transformations and Model-Driven Engineering*, vol. 5765 of *Lecture Notes in Computer Science*, pp. 141–174. Springer Berlin / Heidelberg, 2010.

[KLR⁺12] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer. Model transformation by-example: A survey of the first wave. In A. Düsterhöft, M. Klettke, and K.-D. Schewe (editors), *Conceptual Modelling and Its Theoretical Foundations*, vol. 7260 of *Lecture Notes in Computer Science*, pp. 197–215. Springer Berlin Heidelberg, 2012.

[Kör09] A.-T. Körtgen. *Modellierung und Realisierung von Konsistenzsicherungswerkzeugen für simultane Dokumentenentwicklung*. Ph.D. thesis, RWTH Aachen University, 2009.

[Kör10] A.-T. Körtgen. New strategies to resolve inconsistencies between models of decoupled tools. In *3rd Workshop on Living with Inconsistencies in Software Development (LWI 2010)*. 2010.

[KRW04] E. Kindler, V. Rubin, and R. Wagner. An adaptable TGG interpreter for in-memory model transformation. In A. Schürr and A. Zündorf (editors), *Proceedings of the 2nd International Fujaba Days 2004*, vol. tr-ri-04-253 of *Technical Report*, pp. 35–38. Darmstadt, Germany, September 2004.

[KS05] A. Königs and A. Schürr. Multi-domain integration with MOF and extended triple graph grammars. In J. Bezivin and R. Heckel (editors), *Language Engineering for Model-Driven Software Development*, no. 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2005.

[KS06] A. Königs and A. Schürr. Mdi: A rule-based multi-document and tool integration approach. *Software & Systems Modeling*, 5(4):349–368, 2006.

[KSWR09] A. Kusel, W. Schwinger, M. Wimmer, and W. Retschitzegger. Common pitfalls of using qvt relations - graphical debugging as remedy. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pp. 329–334. IEEE Computer Society, Washington, DC, USA, 2009.

[KW07] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn, Department of Computer Science, University of Paderborn, 2007.

[LAS$^+$14] E. Leblebici, A. Anjorin, A. Schürr, S. Hildebrandt, J. Rieke, and J. Greenyer. A comparison of incremental triple graph grammar tools. In *13th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014)*. 2014.

[Lau13] M. Lauder. *Incremental Model Synchronization with Precedence-Driven Triple Graph Grammars*. Ph.D. thesis, Fachbereich 18 Elektro- und Informationstechnik, Technische Universität Darmstadt, 2013.

[LAVS12a] M. Lauder, A. Anjorin, G. Varró, and A. Schürr. Bidirectional model transformation with precedence triple graph grammars. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos (editors), *Modelling Foundations and Applications*, vol. 7349 of *Lecture Notes in Computer Science*, pp. 287–302. Springer Berlin Heidelberg, 2012.

[LAVS12b] M. Lauder, A. Anjorin, G. Varró, and A. Schürr. Efficient model synchronization with precedence triple graph grammars. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (editors), *Graph Transformations*, vol. 7562 of *Lecture Notes in Computer Science*, pp. 401–415. Springer Berlin Heidelberg, 2012.

[Leh12] S. Lehrig. *Assessing the Quality of Model-to-Model Transformations Based on Scenarios.* Master thesis, Software Engineering Group, University of Paderborn, Software Engineering Group, Paderborn, Germany, Oct. 2012.

[LEO08] L. Lambers, H. Ehrig, and F. Orejas. Efficient conflict detection in graph transformation systems by essential critical pairs. *Electronic Notes in Theoretical Computer Science*, 211(0):17 – 26, 2008. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006).

[LHGO12] L. Lambers, S. Hildebrandt, H. Giese, and F. Orejas. Attribute handling for bidirectional model transformations: The triple graph grammar case. In *Proceedings of the First International Workshop on Bidirectional Transformations (BX 2012)*. 2012.

[Mar10] S. Marlow (editor). *Haskell 2010 Language Report.* 2010.

[MCPW08] L. Murta, C. Corrêa, J. a. G. Prudêncio, and C. Werner. Towards Odyssey-VCS 2: Improvements over a UML-based version control system. In *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*, CVSM '08, pp. 25–30. ACM, New York, NY, USA, 2008.

[MCT08] Y.-S. Ma, G. Chen, and G. Thimm. Paradigm shift: unified and associative feature-based concurrent and collaborative engineering. *Journal of Intelligent Manufacturing*, 19(6):625–641, 2008.

[MG06] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[MSG$^+$13] A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke (editors). *Model-Driven Engineering Languages and Systems – 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, vol. 8107 of *Lecture Notes in Computer Science*. Springer, 2013.

[MTR05] T. Mens, G. Taentzer, and O. Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113 – 128, 2005. Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004) Software Evolution through Transformations: Model-based vs. lmplementation-level Solutions 2004.

[MV11] R. Mannadiar and H. Vangheluwe. Debugging in domain-specific modelling. In *Proceedings of the Third International Conference* *on Software Language Engineering*, SLE'10, pp. 276–285. Springer-Verlag, Berlin, Heidelberg, 2011.

[Ode13] M. Odersky. *The Scala Language Specification.* Programming Methods Laboratory EPFL, Switzerland, November 2013. Version 2.8 Draft.

[OJT$^+$12] F. Oestersötebier, V. Just, A. Trächtler, F. Bauer, and S. Dziwok. Model-based design of mechatronic systems by means of semantic web ontologies and reusable solution elements. In *Proceedings of the ASME 2012 11th Biennial Conference on Engineering Systems Design and Analysis*. ASME, ASME, Nantes, France, 2012.

[OMG01] Object Management Group. Model Driven Architecture (MDA) – a technical perspective, 2001. OMG document `ormsc/01-07-01`.

[OMG06] Object Management Group. Meta Object Facility (MOF) Core 2.0 specification, 2006.

[OMG08] Object Management Group. MOF Query/View/Transformation (QVT) 1.0 specification, 2008.

[OMG10a] Object Management Group. Systems modeling language (OMG SysML™) v1.2, 2010.

[OMG10b] Object Management Group. Unified modeling language (UML) 2.3 superstructure specification, 2010. OMG document `formal/2010-05-05`.

[OMG12] Object Management Group. Object constraint language (OCL) 2.3.1 specification, 2012.

[OMT$^+$08] S. Osmic, E. Münch, A. Trachtler, S. Henkler, W. Schäfer, H. Giese, and M. Hirsch. Safe online-reconfiguration of self-optimzing mechatronic systems. In J. Gausemeier, F. J. Rammig, and W. Schäfer (editors), *Selbstoptimierende mechatronische Systeme: Die Zukunft gestalten. 7. Internationales Heinz Nixdorf Symposium für industrielle Informationstechnik*, pp. 411–426. Feb. 2008.

[PKP05] R. F. Paige, D. S. Kolovos, and F. A. Polack. Refinement via consistency checking in MDA. *Electronic Notes in Theoretical Computer Science*, 137(2):151–161, 2005. Proceedings of the REFINE 2005 Workshop.

[PST13] C. Priesterjahn, D. Steenken, and M. Tichy. Timed hazard analysis of self-healing systems. In J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes (editors), *Assurances for Self-Adaptive Systems Assurances for Self-Adaptive Systems*, vol. 7740 of *Lecture Notes in Computer Science (LNCS)*, pp. 112–151. Springer-Verlag, Berlin, Heidelberg, 2013.

[QACT12a] A. Queralt, A. Artale, D. Calvanese, and E. Teniente. OCL-Lite: A decidable (yet expressive) fragment of OCL. In *Proceedings of the 25th International Workshop on Description Logics (DL 2012)*, vol. 846 of *CEUR Electronic Workshop Proceedings*, pp. 312–322. 2012.

[QACT12b] A. Queralt, A. Artale, D. Calvanese, and E. Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, 73:1–22, 2012.

[RDS$^+$12] J. Rieke, R. Dorociak, O. Sudmann, J. Gausemeier, and W. Schäfer. Management of cross-domain model consistency for behavior models of mechatronic systems. In *Proceedings of the International Design Conference – DESIGN 2012*. 2012.

[RH09] R. Reussner and W. Hasselbring (editors). *Handbuch der Softwarearchitektur. 2. überarbeitete und erweiterte Auflage*. dpunkt.verlag, Heidelberg, 2009.

[Rie08] J. Rieke. *Konsistenzerhaltung zwischen domänenübergreifenden und domänenspezifischen Modellen im Entwicklungsprozess mechatronischer Systeme*. Master's thesis, University of Paderborn, 2008.

[RK10] D. D. Ruscio and D. S. Kolovos (editors). *IWMCP '10: Proceedings of the 1st International Workshop on Model Comparison in Practice*. ACM, New York, NY, USA, 2010.

[RK11] D. D. Ruscio and D. S. Kolovos (editors). *IWMCP '11: Proceedings of the 2nd International Workshop on Model Comparison in Practice*. ACM, New York, NY, USA, 2011.

[Ros96] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[RS12] J. Rieke and O. Sudmann. Specifying refinement relations in vertical model transformations. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*. Springer Berlin/Heidelberg, 2012.

[SC13] M. Stephan and J. R. Cordy. A survey of model comparison approaches and applications. In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2013)*, pp. 265–277. 2013.

[Sch95] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer (editors), *20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94)*, vol. 903 of *Lecture Notes in Computer Science (LNCS)*, pp. 151–163. Springer Verlag, Heidelberg, 1995.

[Sch12] J. Schönböck. *Testing and Debugging of Model Transformations*. Ph.D. thesis, Vienna University of Technology, 2012.

[SEH$^+$10] W. Schäfer, T. Eckardt, C. Henke, L. Kaiser, T. Kerstan, J. Rieke, and M. Tichy. Der Softwareentwurf im Entwicklungsprozess mechatronischer Systeme. In *7. Paderborner Workshop Entwurf mechatronischer Systeme*. 2010.

[SK08a] A. Schürr and F. Klar. 15 years of triple graph grammars. In *Proceedings of the International Conference on Graph Transformations (ICGT 08)*, LNCS 5214, pp. 411–425. Springer, 2008.

[SK08b] M. Seifert and S. Katscher. Debugging triple graph grammar-based model transformations. In *Proceedings of 6th International Fujaba Days, Dresden, Germany*. 2008.

[SKK+10] J. Schoenboeck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer. Catch me if you can – debugging support for model transformations. In S. Ghosh (editor), *Models in Software Engineering*, vol. 6002 of *Lecture Notes in Computer Science*, pp. 5–20. Springer Berlin / Heidelberg, 2010.

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, New York, 1973.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley and Sons, 2006.

[SW07] W. Schäfer and H. Wehrheim. The challenges of building advanced mechatronic systems. In *Future of Software Engineering (FOSE '07)*, pp. 72–84. IEEE Computer Society, 2007.

[Ecl13] The Eclipse Foundation. EMF Compare Project. online, 2013. Accessed on 2013-07-01.

[TMW14] The MathWorks. *MATLAB Documentation*, 2014. `http://www.mathworks.de/de/help/stateflow/index.html`.

[UPB14] University of Paderborn. TGG Interpreter Tool Suite, 2010–2014. `http://www.cs.uni-paderborn.de/index.php?id=tgg-interpreter`.

[Var06] D. Varró. Model transformation by example. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio (editors), *Model Driven Engineering Languages and Systems*, vol. 4199 of *Lecture Notes in Computer Science*, pp. 410–424. Springer Berlin Heidelberg, 2006.

[VBD+13] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. 2013.

[vDHP+12] M. von Detten, C. Heinzemann, M. C. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. Story diagrams – syntax and semantics. Tech. Rep. tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2012. Version 0.2.

[VDI2206] Verein Deutscher Ingenieure. Design methodology for mechatronic systems, 2004.

[vP07] J. von Pilgrim. Mental map and model driven development. *ECEASST*, 7, 2007.

[Wag09] R. Wagner. *Inkrementelle Modellsynchronisation*. Ph.D. thesis, University of Paderborn, 2009.

[WLS+13] K. Wieland, P. Langer, M. Seidl, M. Wimmer, and G. Kappel. Turning conflicts into collaboration. *Computer Supported Cooperative Work (CSCW)*, 22(2-3):181–240, 2013.

[W3C01] World Wide Web Consortium. XML Path Language (XPath) 2.0, 2010.

[XLH+07] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the 22th IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. 2007.

[XSHT09] Y. Xiong, H. Song, Z. Hu, and M. Takeichi. Supporting parallel updates with bidirectional model transformations. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT '09)*. Springer-Verlag, 2009.

[XSHT13] Y. Xiong, H. Song, Z. Hu, and M. Takeichi. Synchronizing concurrent model updates based on bidirectional transformation. *Software & Systems Modeling*, 12(1):89–104, 2013.

[Zün05] A. Zündorf. Story driven modeling: A practical guide to model driven software development. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pp. 714–715. ACM, New York, NY, USA, 2005.

[ZWPG03] F. Zorriassatine, C. Wykes, R. M. Parkin, and N. Gindy. A survey of virtual prototyping techniques for mechanical product development. *Journal of Engineering Manufacture*, 217(4):513–530, 2003.

List of Figures

Das Heinz Nixdorf Institut – Interdisziplinäres Forschungszentrum für Informatik und Technik

Das Heinz Nixdorf Institut ist ein Forschungszentrum der Universität Paderborn. Es entstand 1987 aus der Initiative und mit Förderung von Heinz Nixdorf. Damit wollte er Ingenieurwissenschaften und Informatik zusammenführen, um wesentliche Impulse für neue Produkte und Dienstleistungen zu erzeugen. Dies schließt auch die Wechselwirkungen mit dem gesellschaftlichen Umfeld ein.

Die Forschungsarbeit orientiert sich an dem Programm „Dynamik, Mobilität, Vernetzung: Eine neue Schule des Entwurfs der technischen Systeme von morgen". In der Lehre engagiert sich das Heinz Nixdorf Institut in Studiengängen der Informatik, der Ingenieurwissenschaften und der Wirtschaftswissenschaften.

Heute wirken am Heinz Nixdorf Institut acht Professoren mit insgesamt 200 Mitarbeiterinnen und Mitarbeitern. Etwa ein Viertel der Forschungsprojekte der Universität Paderborn entfallen auf das Heinz Nixdorf Institut und pro Jahr promovieren hier etwa 30 Nachwuchswissenschaftlerinnen und Nachwuchswissenschaftler.

Heinz Nixdorf Institute – Interdisciplinary Research Centre for Computer Science and Technology

The Heinz Nixdorf Institute is a research centre within the University of Paderborn. It was founded in 1987 initiated and supported by Heinz Nixdorf. By doing so he wanted to create a symbiosis of computer science and engineering in order to provide critical impetus for new products and services. This includes interactions with the social environment.

Our research is aligned with the program "Dynamics, Mobility, Integration: Enroute to the technical systems of tomorrow." In training and education the Heinz Nixdorf Institute is involved in many programs of study at the University of Paderborn. The superior goal in education and training is to communicate competencies that are critical in tomorrows economy.

Today eight Professors and 200 researchers work at the Heinz Nixdorf Institute. The Heinz Nixdorf Institute accounts for approximately a quarter of the research projects of the University of Paderborn and per year approximately 30 young researchers receive a doctorate.