

Systematic Development and Re-Use of Model Transformations

Ph.D. Thesis
in partial fulfillment of the requirements for the degree of
"Doctor rerum naturalium (Dr. rer. nat.)"

by

M. Sc. Galina Besova
galina.besova@mail.upb.de
Matr-Nr: 6634727

Main Supervisor: Prof. Dr. Heike Wehrheim
Co-Supervisors: Prof. Dr. Gregor Engels
Prof. Dr. Leena Suhl

Paderborn, June, 2015

Abstract

With an increasing acceptance of model-based software development, where models are used to assist the development of the solution, and a formation of its very promising branch of model-driven approaches, where models are used to (semi-)automatically generate the solution, the research community faces the new challenges. Models and model transformation, as the key concept in model-driven software development (MDSD), require appropriate methods to facilitate their use during the various stages of the process (e.g., in requirements analysis or design). In MDSD, models are usually defined using meta-models, describing the abstract syntax of the languages. Model transformations are then defined based on such descriptions for their source and target languages.

In this thesis, we focus on the research questions related to the development, quality assurance, and re-use of model transformations in the context of MDSD. We consider state-of-the-art techniques in these areas and point out their weaknesses and limitations. We demonstrate that the existing meta-model based approaches to the development of model transformations often suffer from the over-abstraction on the side of meta-models. While meta-models are able to abstract from superfluous details of concrete syntax, they often loose *structural* information inherent in languages, for example, which model elements belong together in particular *structures*. As a consequence, model transformations cannot naturally re-use such language structures, which leads to unnecessary complexity in their development as well as in quality assurance.

To address the above issue, we propose an approach to the development of model transformations, which allows to simplify and improve the quality of the transformations via the exploitation of language structures. The approach is based on context-free *graph grammars*, used for the definition of modeling languages, and produces transformations, defined by *pairing* productions of the source and target grammars in a certain way. This way, the explicit structural information inherent in the grammars can be directly used during the definition of model transformations. We prove that the model transformations, defined using this approach and conforming to its requirements, exhibit four important characteristics: *termination*, *soundness*, *completeness*, and *determinism*.

With respect to the re-use of the developed, high quality model transformations, we demonstrate that existing state-of-the-art approaches require further extensions, when multiple inter-linked models are transformed. In such scenarios, the existing approaches are often inflexible and require changes to the re-used models and/or model transformations. To address these issues, we proposed a modular approach to the re-use of model transformations based on the *model weaving* technique. This approach allows modeling

of inter-relations between the transformed models in a non-invasive and extensible way, as weavings in a *configuration model*. Furthermore, it enables the non-invasive re-use of existing model transformations and of their results (for models that stay the same), by subjecting the woven models and the resulting configuration model to separate (parallel) model transformations.

We also consider a combination of the proposed approaches by defining the re-used model transformations for the weaving based approach using the grammar based development approach.

Throughout this thesis, we exemplify the proposed approaches using a transformation case study between the behavior specification language ADR, which is similar to UML Activity diagrams, and CSP. As the inter-related source models we consider multi-layer systems modeled at different levels of abstraction (e.g., with or without the platform/infrastructure layer models).

Zusammenfassung

Eine steigende Akzeptanz von modell-basierter Softwareentwicklung, in der Modelle benutzt werden um den Entwicklungsprozess der Lösung zu unterstützen, sowie die Entstehung des vielversprechenden Forschungszweigs der modell-getriebenen Ansätze, in dem Modelle benutzt werden, um eine Lösung (semi-)automatisch zu generieren, stellen die Forschung vor neue Herausforderungen. Als die zentralen Konzepte der modell-getriebenen Softwareentwicklung (MDSD), erfordern Modelle und Modelltransformationen passende Methoden, um deren Anwendung in den verschiedenen Phasen des Prozesses (z. B. in der Anforderungsanalyse oder dem Entwurf) zu ermöglichen. Normalerweise sind Modelle in MDSD durch Meta-Modelle definiert, die die abstrakte Syntax der Sprache beschreiben. Die Modelltransformationen basieren dann auf solchen Beschreibungen für die Quell- und Zielsprachen.

In dieser Dissertation liegt der Fokus auf den Fragen die für die Entwicklung, Qualitätssicherung, und Wiederverwendung von Modelltransformationen im MDSD-Kontext relevant sind. Wir betrachten die modernsten Techniken in diesen Bereichen und zeigen deren Schwächen und Einschränkungen auf. Wir zeigen, dass die existierenden meta-modell-basierten Ansätze für die Entwicklung von Modelltransformationen oft unter Überabstraktion durch die Verwendung von Meta-Modellen leiden. Obwohl die Meta-Modelle in der Lage sind von den unnötigen Einzelheiten der konkreten Syntax zu abstrahieren, lassen sie oft inhärente *strukturelle* Informationen der Sprachen aus, wie zum Beispiel welche Modellelemente zusammen zu einer bestimmten Struktur gehören. Als Konsequenz können Modelltransformationen solche Sprachstrukturen nicht verwenden. Dadurch entsteht eine unnötige Komplexität in deren Entwicklung und Qualitätssicherung.

Um die genannten Fragen anzugehen, schlagen wir einen Ansatz für die Entwicklung von Modelltransformationen vor, der eine Vereinfachung der Entwicklung und eine Verbesserung der Qualität der Transformationen durch das Nutzen von Sprachstrukturen erlaubt. Der Ansatz basiert auf kontext-freien *Graphgrammatiken*, die für die Sprachdefinition benutzt werden, und produziert Modelltransformationen, die durch eine *paarweise* Zuordnung der Produktionen der Quell- und Zielgrammatiken definiert werden. Dadurch können explizite Informationen über die Strukturen aus den Grammatiken direkt für die Definition von Modelltransformationen benutzt werden. Wir beweisen, dass die Modelltransformationen, die durch den Ansatz definiert worden sind und dessen Anforderungen erfüllen, die folgenden vier Eigenschaften haben: *Terminierung*, *Korrektheit*, *Vollständigkeit*, und *Determinismus*.

Bei den existierenden Ansätzen zur Wiederverwendung von hochqualitativen Modelltransformationen zeigen wir, dass die modernsten existierenden Techniken eine Weiterentwicklung benötigen, um mehrere verbundene Modelle zu unterstützen. In solchen Fällen sind die existierenden Ansätze häufig unflexibel und erfordern Änderungen an den Modellen und/oder Modelltransformationen. Um die Fragen anzugehen, schlagen wir einen modularen Ansatz für die Wiederverwendung von Modelltransformationen vor, der auf der Technik der *Modellverwebung* basiert. Dieser Ansatz unterstützt eine nicht-invasive und erweiterbare Modellierung von Verbindungen zwischen Modellelementen als Verwebungen (engl.: weavings) in einem *Konfigurationsmodell*. Dadurch, dass die verwobenen Modelle und das Konfigurationsmodell von separaten (parallelen) Modelltransformationen bearbeitet werden, können wir weiterhin eine nicht-invasive Wiederverwendung der Modelltransformationen und deren Ergebnissen ermöglichen.

Zusätzlich kombinieren wir die zwei vorgeschlagenen Ansätze, indem wir die einzelnen Transformationen für den auf Modellverwebung basierenden Ansatz durch den grammatikbasierten Ansatz definieren.

Um die vorgeschlagenen Ansätze zu demonstrieren haben wir in dieser Dissertation ein Transformationsfallbeispiel zwischen der Verhaltensbeschreibungssprache ADR (inspiriert von UML Aktivitätsdiagrammen) und CSP verwendet. Als verbundene Modelle in dem Szenario betrachten wir Modelle von mehrschichtigen Systemen mit unterschiedlichen Abstraktionsebenen (z.B. mit oder ohne Plattform-/Infrastrukturmodell).

Acknowledgments

First of all, I would like to thank my supervisor – Prof. Dr. Heike Wehrheim – for the support, guidance, and patience during this exciting and sometimes challenging journey. I would also like to thank her and my research group colleagues for providing me with an opportunity to be a part of the team. Further, I would also like to thank Prof. Dr. Gregor Engels for his time, dedicated to reviewing my thesis, and Prof. Dr. Leena Suhl, Prof. Dr. Eyke Hüllermeier, and Prof. Dr. Stefan Böttcher for being a part of the committee.

In particular, I would like to thank my colleague Dominik Steenken for fruitful discussions and for taking time to review this thesis. I would also like to explicitly thank my colleagues Sven Walther, for the productive team work, and Oleg Travkin, for the useful discussions. I would also like to thank my student assistants Thomas Haarhoff and Arman Sheikholeslami for supporting me in the discussion and implementation of the new ideas for this thesis.

I also want to thank all members of the CRC 'On-the-Fly Computing' and, especially, Svetlana Arifulina, Marie Christin Platenius, and Matthias Becker, for the inspiring discussions and pleasant environment.

Finally, I want to thank my parents and grandparents for their support and patience during all these years. Thank you for providing me with this opportunity to explore this extraordinary world and to seek my place in it.

Contents

1	Introduction	1
1.1	Problem Statement	5
1.2	Research Objectives and Method	7
1.3	Thesis Structure	9
2	Foundations of Model-Driven Software Development (MDSD)	11
2.1	Conceptual Description	11
2.2	Definition of Modeling Languages	15
2.3	Definition of Model Transformations	21
3	Introduction to Case Study: Transformation of Multi-Layer Systems	24
3.1	Example System: Trip Planning Software (ADR)	25
3.2	Example Platform: Communicating Sequential Processes (CSP)	33
3.3	Example Transformation: ADR-to-CSP	36
4	Systematic Development of Model Transformations	48
4.1	Development of Transformations	48
4.1.1	Transformation Types	49
4.1.2	Transformation Design	51
4.1.3	Transformation Implementation	54
4.2	Quality Assurance of Model Transformations	73
4.2.1	Four Functional Properties of Transformations	74
4.2.2	Quality-By-Construction: Transformation Development	76
4.2.3	Quality-By-Checking: Transformation Analysis	80
4.3	Re-Use of Model Transformations	83
4.3.1	Transformation Chaining	84
4.3.2	Transformation Adaptation: Superimposition	87
5	Grammar Based Approach to the Development of Model Transformations	89
5.1	Description of Source and Target Languages	89
5.1.1	String Grammars	90
5.1.2	Graph Grammars	92
5.2	Conceptual Description of the Approach	98
5.2.1	Transformation Definition	99
5.2.2	Transformation Execution	110

5.3	Formalization of the Approach	112
5.3.1	Formalization of Transformation Definition	114
5.3.2	Formalization of Transformation Execution	129
5.4	Guaranteed Quality Properties of Transformations	135
5.4.1	Termination	136
5.4.2	Soundness and Completeness	138
5.4.3	Determinism	144
5.5	Tool Support	147
5.6	Related Approaches	150
5.7	Evaluation and Discussion	155
6	Weaving Based Approach to the Re-Use of Model Transformations	162
6.1	Conceptual Description of the Approach	165
6.1.1	Weaving Based Modeling of Models Inter-Relations	166
6.1.2	Modular Execution of Transformations	172
6.2	Tool Support	188
6.3	Related Approaches	190
6.4	Evaluation and Discussion	193
7	Weaving Based Re-Use of Grammar Based Model Transformations	198
7.1	Grammar Based Transformation of Layers Models	198
7.2	Grammar Based Transformation of Configuration Models	202
7.2.1	Grammar of Configuration Modeling Language	203
7.2.2	Definition of the Configuration Model Transformation	205
7.3	Discussion	213
8	Conclusion	217
8.1	Thesis Contribution	217
8.2	Analysis of Grammar Based Development Approach	218
8.3	Analysis of Weaving Based Re-Use Approach	221
8.4	Future Research Directions	223
	Bibliography	I
	List of Figures	XXII
	List of Tables	XXV

Chapter 1

Introduction

In the last decades the use of models in software development has been steadily increasing, starting with programs in, e.g., the C language, as abstractions of machine instructions on assembler level, and growing to the comprehensive models of architectural and other aspects of large systems in, e.g., the Unified Modeling Language (UML) [UML], at present. Currently, this tendency progresses further to support the idea of modeling-in-the-large [BJRV05], where individual models are elements of another model, called *mega-model*.

With the above tendency, the idea of model-based software development finds more and more acceptance among software developers [BCW12]. Its main principle is the use of models at different stages of the software development process, usually as main artifacts for the implementation of the end code [BBG05]. This initial idea lead to the formation of a more specific and sophisticated sub-paradigm of *model-driven* software development (MDSD) [GPR06] (see Chapter 2 for details), that at present already provides us with efficient solutions, as well as with the questions for further research.

Like in the case of model-based software development, models play a key role in MDSD. The main idea of MDSD is to allow step-by-step development of a software solution by starting with a quite abstract model and gradually refining and transforming this model to obtain the respective concrete solution through a chain of intermediate system models. The initial abstract model usually represents the requirements that the system, that is to be developed using MDSD, has to fulfill. The resulting solution usually represents system code, in e.g., C/C++ or Java, providing the required functionality and fulfilling further non-functional requirements, if given.

Often the refinement and transformation process of MDSD is combined with various types of model-based analysis, that can be applied both to the initial and the intermediate models. Examples of the types of such model-based analysis include: performance prediction (e.g., [KR10]), reliability prediction (e.g., [BWW11]), verification of functional properties (e.g., [WW13]), check of dead-lock and live-lock freedom (e.g., [BWWB12]), and so on. In their turn, these analysis techniques require other model transformations into analysis formalisms, like Communicating Sequential Processes (CSP) [Hoa85].

The models created and used in the MDSD process can vary in their degree of abstraction, scope, modeling languages (e.g., graph- or text-based), and so on. This makes

the refinement and transformation process quite challenging and gives model transformations, that are usually used to address this challenge, a key position within MDSD.

This brings us to some important research challenges in the MDSD community with respect to its previously identified key concepts. The first challenge is the development of suitable modeling languages, design/development methods, and re-use approaches for models and model transformations in MDSD. The second challenge is the quality assurance for the developed system models and model transformations, that is often considered separately from the first group of questions.

The issues of defining suitable modeling languages and design methods for different types of models, to be used in MDSD, is what first attracted the focus of the research community. One of the central results of this research is the specification of the main modeling language (or rather a set of modeling languages) in MDSD, called *Unified Modeling Language (UML)* [UML], as a standard of the Object Management Group (OMG) [OMG] group. This standard contains a set of modeling concepts often used in MDSD and it also allows the definition of further *domain-specific modeling languages (DSLs)* on the basis of its profile mechanism, when certain domain-specific modeling concepts are needed. Various software development methods have been proposed based on UML, including the most well-known *Rational Unified Process (RUP)* [Sof98], but also the *agile modeling* process [Rum02].

Later, the OMG has introduced a further level of abstraction into MDSD in the form of a new standard called *Meta-Object Facility (MOF)* [MOFa], that serves as a uniform modeling language for other modeling languages (meta-language) in MDSD and allows modeling of different domain specific languages, with UML being one of them, without the use of UML profiles. Instead, a definition of a modeling language is represented as a meta-model by using the concepts of MOF (see Section 2.2 for details). Such meta-model can describe both simple language concepts (as classes) as well as high-level language structures (as additional structuring classes).

As for the re-use and quality assurance for the system models, the former is usually enabled through the model transformations (e.g., model evolution [GDA13], model composition [BSML07, AEN⁺08, KM10]) and related techniques, that we consider later. The latter is a separate topic of active research with respect to validity, completeness, consistency [BL09], and further complex model properties (see [Kan02, RB08, PR10]).

The issues of defining suitable modeling and implementation languages, and design methods for different types of model transformation used in MDSD, came into the focus of the research community a bit later, once such standards like UML and MOF reached certain degree of maturity and allowed convenient system modeling. One of the central results of the research on model transformation modeling/implementation is the specification of the standard for the model transformation languages (as a set of three languages) in MDSD, called *Query/View/Transformation (QVT)* [QVTb], also by the OMG group. Its focus is on providing means to define model transformations on a relatively high level of detail such that these definitions can be executed. This level of detail

is also the focus of most existing modeling/specification languages for model transformations, like ATLAS Transformation Language (ATL) [ATL], Epsilon [Eps], mediniQVT [QVTa], and others. Unfortunately, the question of model transformation design (see Section 4.1.2) is not addressed there.

Model transformations in the MDSD context are defined between the source and target languages, that are defined by meta-models. Thus, their design and implementation highly depends on the structure of such meta-models. In the cases where a source meta-model does not contain any structuring classes (see Section 2.2 for details) for representing the high-level structures of the source language, but the required transformation mappings need to use them, the development of such structure-based model transformation can become quite complex. Imperative constructs are often required from the transformation language in order to implement such model transformation. In the more rare cases when such structuring classes are in the source meta-model, the development of the required structure-based model transformation becomes easier. Yet, the interconnection of the results of the transformation rules for such structure-based mapping can become more complex. We consider these issues on the examples in concrete model transformation languages in Section 4.1.3.

Only a few approaches exist [Küs06, GdLK⁺10, LKR13] that propose methods for the development of model transformations, that contain design and other stages before the executable implementation level of detail is reached. In [GdLK⁺10], the authors propose to apply MDSD to develop model transformations starting with the requirements as transformation mappings and refining them step-by-step to obtain the high- and low-level design mappings, and, finally, their implementation (here in Epsilon [Eps]). In [Küs06], the authors follow the same goal but start with informal under-specified transformation mappings and a combinations matrix in order to design the transformation and later refine this design into a special type of transformation rules. In [LKR13], the authors propose to design model transformation mappings as constraints with their later implementation through automatic code generation. Overall, the modeling language(s) to support different stages of the development of model transformations, as the UML language for the general MDSD based software development area, have not yet been agreed upon and developed for this domain. Although, it is possible to adapt UML for this purpose, approaches that suggest this, like [LKR11], find less acceptance in the community than specialized DSLs for the different stages of the development of model transformations, like in [WKS⁺09, GdLK⁺10].

As for the quality assurance and re-use of model transformations, these topics remain the areas of active research within the MDSD community. Multiple approaches to the re-use of model transformations have been proposed, including those for transformation chaining (e.g., [BPOB13]), superimposition [Wag08], evolution (e.g., [GDA13]), and so on (see Section 4.3 for details). Most of these approaches focus on direct sequencing/orchestration of compatible (through target/source language meta-models) model transformations or on generating such chains based on certain models, like in [AEM12].

The cases when multiple input source models are transformed in parallel are hardly considered by the existing approaches, especially, when those source models are somehow inter-connected. The only work, that we are aware of, that considers a similar scenario [SCGM08], is concerned with the inter-model links between the generated output target models. Thus, we see further research potential here in order to improve and broaden the support of the re-use of model transformations in MDSD.

The question of quality assurance of model transformations is another central issue for the success of MDSD. The characteristics of interest here include such functional properties like transformation termination, correctness of its result with respect to the target language (soundness), completeness with respect to its source language, and the deterministic nature of its result. This last property is often considered as an effect of confluence property. Other more subjective characteristics include such properties as intuitiveness/comprehensibility, maintainability, and so on [LAD⁺14].

In this thesis we focus on the former four properties and discuss them in Section 4.2.1 in detail. Thereby, we consider correctness (soundness) of model transformations on syntactical level and refer to [Sem12] for further details on the topic of semantic correctness of model transformations. For the chosen four properties there are various approaches that aim to guarantee them either by-construction or through the analysis (by-checking) of already existing model transformations. We consider these groups of approaches in detail in Sections 4.2.2 and 4.2.3, respectively.

Overall, none of the existing quality assurance approaches for model transformations from both groups consider all of the selected properties in full. For instance, in [EEdL⁺05, EEHP09] the authors consider sufficient criteria for the termination of model transformations defined based on algebraic graph transformations [CMR⁺97] and Triple Graph Grammars (TGGs) [Sch94], respectively. In [Küs06, HEOG10], the authors consider sufficient criteria for the confluence of graph-based (special form with control structures) and TGGs based model transformations, respectively. This property, in its turn, implies their determinism. All of the above approaches belong to the group of quality-by-construction.

Several approaches [WKS⁺09, CCGdL10] from the second quality-by-checking group aim to verify termination and confluence of model transformations. For this purpose, they transform the definitions of the analyzed model transformations into the chosen formalisms, like Colored Petri Nets (CPNs) [Jen87] or Object Constraint Language (OCL) [OCLb]. In [WKS⁺09], the analyzed model transformations are initially specified using their own DSL, called *Transformation Nets*, and in [CCGdL10] using TGGs. Although, the verification approach from [CCGdL10] considers all four properties, that are of interest here, the bounded verification used there makes its results incomplete.

Only a few of the existing approaches [Küs06, EHS09, EEHP09, CCGdL10] consider sufficient criteria for the properties of correctness and completeness for model transformations. These approaches mostly [EHS09, EEHP09] focus on model transformations

described using TGGs and have a TGGs specific definition of these properties. We discuss these definitions in detail in Section 4.2. Here, we just note that our definitions, discussed in Section 4.2.1, differ from them.

The approach in [Küs06] is the only further work, that we are aware of, that considers sufficient criteria for syntactical correctness of model transformations. The author, however, only requires the source/target parts of transformation rules (of a certain form) to be instances of the respective source/target meta-models. This, on its own, does not guarantee syntactical correctness of the complete model transformations with respect to the meta-models. The author of [Küs06] admits that a stronger notion of syntactical correctness, as the one that we use, is difficult to achieve, since one must take into account all transformation rules as well as their relationships.

In the second, quality-by-checking group of approaches, only a very few approaches [WKC06, CCGdL10] attempt to address syntactical correctness and completeness of model transformation and, in both cases, their results are incomplete or insufficient to show these properties, as we see in Section 4.2.3.

Based on the situation with the quality assurance for model transformations discussed above, we conclude further need for the research in this direction. Especially, with respect to the guarantee of syntactical correctness and completeness properties for model transformations in MDSD, where the involved source and target languages are defined by meta-models. Furthermore, here we see some synergy potential with the new techniques to improve and simplify the development of model transformations, that natively support the description of structure-based transformation mappings between the languages.

Of course, the issues and the remaining research questions, sketched above, do not cover all areas and open questions of the MDSD process. The questions being left out include, for instance the following: change propagation between the models and the code created in the process [EMM⁺12]; model consistency management [GPR11]; reverse engineering [CGdL14, BCDM14], including the integration of manually written code; and more. On the less technical side, there are also questions like the propagation and traceability of system requirements in the MDSD [ZSZ11] and many more, that also need to be addressed. These questions are also the focus of ongoing research activities (see [GPR06, PR10, BCW12] for further reading).

1.1 Problem Statement

The research issues that we tackle in this thesis are twofold. First, we consider the issues related to the development of model transformations, especially those that require structure-based mappings between the languages. Such mappings, compared to simple element to element ones, describe which high-level structure(s) of the source language should be transformed into which corresponding high-level structure(s) in the target

language. As discussed above, this type of mapping is not sufficiently supported by the existing approaches and languages for the development of model transformations.

In the cases when the used definitions of the involved source and target languages, in the form of meta-models, do not contain explicit description of the high-level structures of the languages (e.g., as structuring classes), the developed model transformations can not use them. Thus, the developer is forced to obtain such structural information from each input source model during the transformation execution. For instance, to find the corresponding nodes that form a structure, like a decision/merge block, would require much more complex transformation rules, both in their development, maintenance, and quality assurance. In the cases when the structuring classes are available, which is quite rare, such search is not required, yet another issue persists.

This issue is the connection of the transformation results of the rules, that describe/implement the structural mappings. We show in Section 4.1.3, that it leads to the additional unjustified complexity of the connection mechanism used in the model transformations, even if structuring classes are used to describe high-level language structures.

The question related to the above issues, that we also aim to address in this thesis, is the quality of such model transformations that enable structure-based mappings. The mentioned complexity increase, due to the search for the required structural information and due to the more complex composition of the transformation results, often limits the possibilities to guarantee the quality of the developed model transformations. This is especially true in the cases when imperative constructs are used to define such model transformations. Nevertheless, it is essential to provide methods to guarantee the quality of such model transformations, ideally, by-construction or, at least, via analysis. Our goal is to provide such method as a part of the approach, that tackles the mentioned issues in the development of model transformations that require structure-based mappings.

The second issue, or rather also a group of issues, that we address in this thesis, is concerned with the re-use of model transformations. More precisely, with the re-use scenario where multiple inter-related models need to be transformed into some target language in an efficient way and where the transformations for the individual models into that target language already exist. In this scenario we need to address the questions such as the following: modeling of the inter-model links; chaining of the individual model transformations; and, depending on the chosen chaining strategy, composition of the transformation results into the complete target model. Thereby, the listed questions are related and the choice of the solution for one of them influences the choices available for the remaining issues.

Further complexity to the scenario described above is added by the fact that the individual models can be modeled in different languages (model heterogeneity) and the existing model transformations for them can be defined in different, potentially incompatible, languages (model transformation heterogeneity). Additionally, some target languages might require specific adjustments to the individual model transformations and/or their results in order to allow the composition of these results for further use.

The issue of potential model heterogeneity in the considered scenario brings additional constraints on possible solutions. Here, we can no longer use a simple copying composition of models elements into the complete model or common language elements to describe the inter-model links in such model. The issue of potential heterogeneity of model transformations, in its turn, eliminates the possibility to use the model transformation composition and re-use approaches, that employ language-specific re-use mechanisms of some model transformation languages, like the one for ATL used in [Wag08]. The question of the composition of transformation results is also influenced and complicated by the diversity of existing target languages. Here, the heterogeneity lies in the composition mechanisms available within the different languages.

With the above, the issues of the MDSD process, that are in the focus of this thesis, have been selected and discussed in more details. Next, we provide a description of the goals of this thesis with respect to solving the above issues and the desired characteristics of the proposed solutions.

1.2 Research Objectives and Method

The general goal of this thesis is to develop approaches to solve the above issues based on the existing state-of-the-art methods and new ideas for improvements. We also aim to show that the proposed approaches have the desired characteristics by using extensive transformation examples and poofs, when applicable.

To solve the issues related to the development and quality assurance of model transformations, that require structure-based mappings, we aim to develop an approach that has the following characteristics:

- it allows explicit and convenient modeling of such high-level language structures and mappings between them;
- it allows simple and, ideally, automatic appropriate connection of the transformation results of the individual mappings;
- it allows to keep the resulting model transformations declarative;
- and it guarantees their quality properties by-construction.

Thereby, we focus on the four mentioned quality properties for model transformations – termination, syntactical correctness (soundness), completeness, and determinism – described in Section 4.2.1.

As the starting point for the development of such method we see existing state-of-the-art approaches in the MDSD community (see e.g., [GPR06, BCW12]), graph theoretical results (see e.g., [Roz97, EEPT06]), and existing techniques in the area of compiler construction (see e.g., [ASU86]).

To address the issues related to the re-use scenario for model transformations, considered in Section 1.1, we aim to develop an approach with the following characteristics:

- it allows language-independent modeling of inter-relations between the models in an additional separate model;
- it allows modular, language-independent, and parallel execution of the re-used individual model transformations;
- and it provides a basis for the re-use of their results without the need for re-transformation.

Thereby, the method used for the modeling of inter-model links has to allow convenient modification and extension of its modeling capabilities. The re-used model transformations should be treated as black-boxes and left unchanged during the chaining. As for the re-use of the results of individual model transformation, we do not aim to find a language-independent solution for this, but to provide general criteria required for this and demonstrate them on an example in a particular target language.

As the starting point for the development of such a re-use approach we also see the existing state-of-the-art methods in the MDSD community and the existing composition approaches for the potential target languages, like the one in [FEK⁺07].

With respect to the above objectives, the contribution made in this thesis is twofold. Firstly, we propose an approach to address the issues related to the development and quality assurance of model transformations, especially those that require structure-based mappings. This approach employs, additionally to meta-models, meta-model conform graph grammars in order to define the source and target languages. Such grammar based description allows us to explicitly model higher-level structures of the languages in a constructive and intuitive way.

We then propose a way to profit from this explicit structural information in the development of model transformations (in transformation rules). The resulting model transformation can be kept intuitive and declarative. Furthermore, the second benefit of the used grammar based language definition is an ability to re-use the automatic connection mechanism of the individual productions in order to compose the results of the individual transformation rules.

As for the quality assurance for model transformations, the proposed grammar based approach allows to guarantee the four properties mentioned before and defined later in Section 4.2.1. This is possible due to the inductive nature of grammar based language definitions and due to the way we construct the transformation rules based on them. For the latter, we define a set of conditions sufficient to guarantee the above properties.

Secondly, we propose an approach to address the issues related to the re-use of model transformations in the scenario considered above, where multiple (heterogeneous) models with re-configurable inter-relations need to be transformed into some target language.

This approach employs a technique called model weaving to model inter-relations between the models in a flexible, model language independent, and extensible way. Its result is stored into a separate model and transformed into the chosen target language independently from the initial models, which allows modular parallel execution of the re-used model transformations. In the proposed approach, the results of these model transformations are then composed using the composition constructs of the chosen target language, which allows their re-use for the unchanged models.

1.3 Thesis Structure

The structure of the rest of this thesis is as follows. In Chapter 2, we survey the foundations of MDSD, its main concepts and steps (Section 2.1), and the way modeling languages (Section 2.2) and model transformations (Section 2.3) are defined there.

In Chapter 3, we introduce the transformation case study, that we later use to demonstrate the approaches that we propose in this thesis. This case study is described in the following three parts: the source language, including its meta-model, and an example model in this language (Section 3.1); the target language, including its meta-model (Section 3.2); and the model transformation between these two languages, its purpose and a summarized description of the mappings that it performs in the form of requirements (Section 3.3).

In Chapter 4, we survey the state-of-the-art in the development of model transformations in the context of MDSD. We discuss the questions of the design and implementation (Section 4.1), quality assurance at the design and (re-)use time (Section 4.2), and different types of the re-use of model transformations (Section 4.3).

In Chapter 5, we present the approach that we propose to tackle the issues related to the development of model transformations, that require structure-based mappings. First, we introduce the mechanism that we have chosen in our approach to define the source and target modeling languages (Section 5.1). Then, we give a conceptual example-based introduction to the approach (Section 5.2), including the way model transformations are defined and executed using it. We then formalize the described concepts in detail (Section 5.3). Based on the provided formalization, we prove several quality properties for the resulting model transformations (Section 5.4). Afterward, we discuss the required tool support (Section 5.5) and survey the most closely related approaches (Section 5.6). Finally, we provide an evaluation of the proposed approach and discuss its main differences to the existing state-of-the-art (Section 5.7).

In Chapter 6, we present the approach that we propose to tackle the issues related to the re-use of model transformations in scenarios with multiple inter-connected input source models. First, we give a conceptual example-based introduction to the approach (Section 6.1), including the way the model inter-relations are modeled and the way the model transformations are chained in this approach. Thereby, we use the case study

introduced in Chapter 3 and its target language for the demonstration. Afterward, we discuss the required tool support (Section 6.2) and survey the most closely related approaches (Section 6.3). Finally, we provide an evaluation of the proposed approach and discuss its main differences to the existing state-of-the-art (Section 6.4).

In Chapter 7, we demonstrate how the proposed approach to the development of model transformations from Chapter 5 works together with the proposed approach to the re-use of model transformations from Chapter 6 in the desired scenario. Thereby, we demonstrate how all of the involved model transformations can be successfully developed with the former approach (Sections 7.1 and 7.2) and what is needed to facilitate this. We end with the discussion and interpretation of the results of the performed combination (Sections 7.3).

In Chapter 8, we conclude this thesis by addressing the following topics: stating the contributions of this thesis (Section 8.1); analyzing the proposed approaches, their application scope, the design decisions made, and their strengths and weaknesses (Sections 8.2 and 8.3); and, finally, discussing the directions for future research (Section 8.4).

Chapter 2

Foundations of Model-Driven Software Development (MDSD)

During the past years the software development process has been steadily transitioning from a direct, low-level, manual programming of software code to a high-level, (semi-)automatic, *model-driven software development (MDSD)* process. At the core, various system models for its different aspects with various levels of abstraction have replaced direct code-based representation of the developed software. These models now allow us to build a software system step-by-step, starting with a high-level description of its requirements, defined using suitable domain specific languages.

This high-level description is then used as a basis for developing subsequent models of a system, that fulfills these requirements. Some of these subsequent models are build from scratch, whereas others are created through a manual or automatic refinement of the already existing ones. The more detailed and solution-centered the system models in the MDSD process become, the closer we get to the required executable software system. The refinement used in this process is one of the operations on models in the MDSD context. All of these operations are referred to as *model transformations (MTs)*.

In this Chapter we consider the main ideas and approaches behind MDSD (Section 2.1) and, then, focus on the definition of models (Section 2.2) and model transformations (Section 2.3) as its key concepts.

2.1 Conceptual Description

The model-driven software development (MDSD) process is, as the name indicates, based on and even driven by models. In general, a model can be defined as follows (from [PR10]).

Definition 2.1.0.1 (Model). *A model is an abstracting representation of the existing reality or a prototype of a reality to be created, that is built for a specific purpose.*

This generic definition of a model can be specialized in the context of software systems to define *system models*, used in MDSD, as follows (adapted from [GPR06]).

Definition 2.1.0.2 (System Model). *A system model is a model that describes (or specifies) a system, and contains one or more of its relevant aspects including its structure, behavior, functionality, and the context of use.*

During the MDSD process, multiple system models are created for different purposes to build an overall description of the developed/extended software system. The term *model-driven* emphasizes that this process uses models to direct the course of requirements understanding, system design, and other development tasks [MMea03]. Thereby, some of the created models, like requirements specification models, are usually built manually, whereas others, like implementations, are (semi-)automatically generated by transforming higher-level system design model(s). A *model transformation* in the MDSD context is defined as follows (from [MMea03]).

Definition 2.1.0.3 (Model Transformation). *A model transformation is the process of converting one (source) model to another (target) model of the same system.*

The main idea behind the MDSD is to build a bridge consisting of models between the client's requirements on the system and the executable software as a result of the process. After the requirements have been provided, each next model of this bridge is created by transforming/refining some existing ones. Models that are usually built include, but are not limited to the following: business and requirements models, requirements analysis models, high-level and detailed design models, analysis models, implementation, and deployment models. These models are used in different tasks of the MDSD process to transfer the growing volume of information about the developed software system.

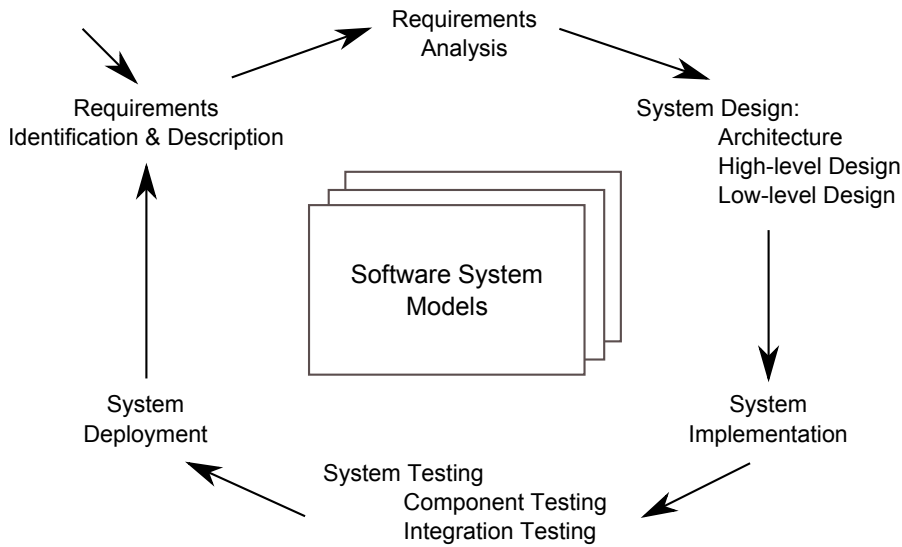


Figure 2.1: Tasks of the MDSD process arranged in a cycle (from [GPR06])

Figure 2.1 show typical tasks and sub-tasks carried out in the MDSD process including requirements description and analysis, software system design, implementation, testing,

and deployment. Some additional tasks like system analysis can extend this list. The above tasks can be carried out using various arrangement and sequencing techniques like the waterfall model [Ben87], the rational unified process (RUP) [Sof98], or any other. In Figure 2.1 we sketch an iterative execution of these tasks, as advocated by RUP.

The arrows in Figure 2.1 represent output/input relations between the tasks. Each next task uses the output models of previous task(s) to refine them and/or build on their basis further models through manual or (semi-)automatic model transformations. In the case of the *'requirements identification'* task, its input also comes from the outside of the cycle (initial requirements).

Since models are central to the complete MDSD approach, their quality and precision can significantly influence its result. Thus, a systematic and re-usable approach to the definitions of models is an essential goal in MDSD. The same is true for the operations on models in MDSD i.e., model transformations. Quality and re-usability of model transformations are also essential for an efficient MDSD.

The most widely-used approach, that has been proposed to addresses the above questions, is called *Model-Driven Architecture (MDA)*. This approach from *Object Management Group (OMG)* [OMG] facilitates the use of MDSD principles by defining a concrete method for using models in the MDSD process and providing the required methods and de facto standards for defining models and model transformations. Due to the wide acceptance of MDA by the MDSD community, we continue our description of the MDSD process, including the model and model transformation definitions, using the techniques it provides.

The main idea behind the MDA is the separation of the specification of system operation from the details of the way this system uses the capabilities of its platform. Thereby, a *platform* is defined in the following generic way (adapted from [MMea03]).

Definition 2.1.0.4 (Platform). *A platform is a set of sub-systems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns/formats, that any system can use without concern for the details of the platform's implementation.*

Based on its definition, a platform in MDA can be an operating system (e.g., Windows, Unix), a middle-ware (e.g., CORBA, Java2EE) or some specialized execution engine. Platforms often have a layered structure, where one platform provides functionality for the next one, as in the case of an operating system and a middle-ware that runs on top.

Inspired by the above separation principle, MDA formulates the following system development steps with their related system models (when applicable):

Step 1 Modeling of the system requirements and its environment in a *computation independent model (CIM)*.

- Step 2 Modeling of the system operation (structure, functionality, and behavior) without any details of platform dependency in a *platform-independent model (PIM)* based on CIM from step 1 of this list.
- Step 3 Choice of a particular platform for the modeled system.
- Step 4 Modeling of the chosen platform in a *platform definition model (PDM)*, if no existing model is available.
- Step 5 Transformation of the system's PIM into a system's model with the details of the use of the chosen platform – a *platform-specific model (PSM)*.

Each model created in one of these steps has its specific role in bridging the gap between the system requirements and the end software system. A CIM allows communication of system knowledge between the domain experts (clients/stakeholders) and the system designers. A PIM allows communication between the system designers and its developers, and, finally, a PSM allows communication between the system developers and the platform, that executes the end result. In order for the developers to be able to communicate with the platform, they need to use the vocabulary defined in the PDM for that platform. Often, PSMs created for a specific programming/specification language environment, as a platform, are separated into their own category called *code*.

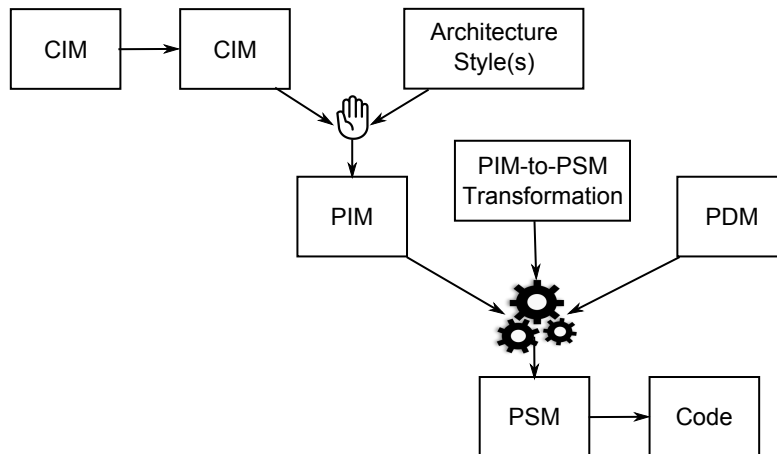


Figure 2.2: Models in the MDA process (based on [GPR06])

Figure 2.2 shows the steps described above, while focusing on the models created/refined/used in these steps. It provides further types of models used in the MDA process – *architectural style(s)* and *PIM-to-PSM* transformation definitions. Models of the first type contain different architectural concepts, patterns, interfaces, and so on. They are used to create a PIM from the given CIM of the system by making first design and architectural decisions for this system. This step is usually done manually or, at most, through a semi-automatic transformation of the CIM (denoted by a ‘hand’ symbol). The second model type contains a PIM-to-PSM transformation description, which should be executed automatically (denoted by a ‘gear’ symbol).

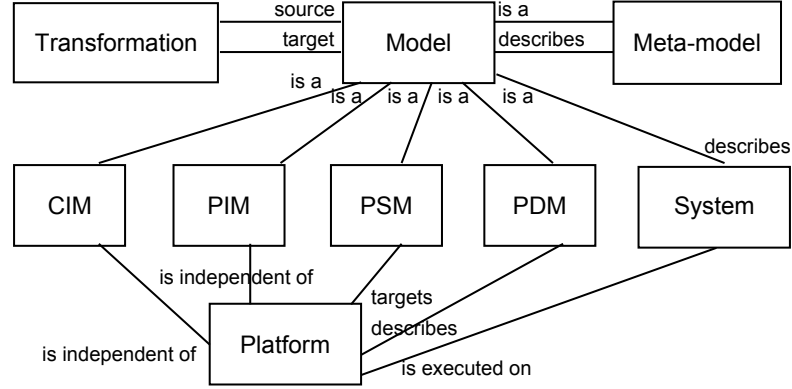


Figure 2.3: Key concepts of MDA (adapted from [GPR06]).

Figure 2.3 summarizes all of the described concepts in the context of MDA and their inter-relations. It also contains an additional concept of a *meta-model*, which we have not discussed yet. This concept represents a special type of model, that is used in MDA to describe other models. It can be defined as follows (from [GPR06]).

Definition 2.1.0.5 (Meta-model). *A meta-model is a model that describes syntax of other model(s) i.e., the types of elements, relations, and their inter-connections, that are allowed in these models.*

The set of models described by, and *conform to*, a meta-model forms a modeling language that is the *language of this meta-model*. We consider this meta-model based approach to the definition of modeling languages in details in Section 2.2.

Each source and target model of a model transformation in MDA belongs to a modeling language, described by some meta-model. Thus, each PIM-to-PSM model transformation has to be defined based on the constructs from these meta-models and, in some cases, some additional information. We consider all further details of the definition of model transformations in MDA in Section 2.3.

2.2 Definition of Modeling Languages

Description of a modeling language usually contains two different aspects (see [GPR06]) – syntax and semantics – that can be detailed to a different extent. A language syntax consists of an abstract and, often, a concrete part. The abstract part defines the element types allowed in the language and their inter-connections, and the concrete part specifies a visual representation of elements of these types. A language semantics consists of a static and, sometimes, a dynamic part. The static semantics contains *well-formedness* rules or constraints [BBG05, FCB⁺13], that describe the allowed combinations of abstract syntax elements in models, and the dynamic or operational semantics describes the allowed behavior of models.

As we have discussed before, in an MDA setting, the syntax of models is given in terms of *meta-models*. More precisely, meta-models define the abstract syntax of modeling languages and, to some degree, their static semantics. A meta-model itself is conform to its own meta-model, which is referred to as *meta-meta-model*.

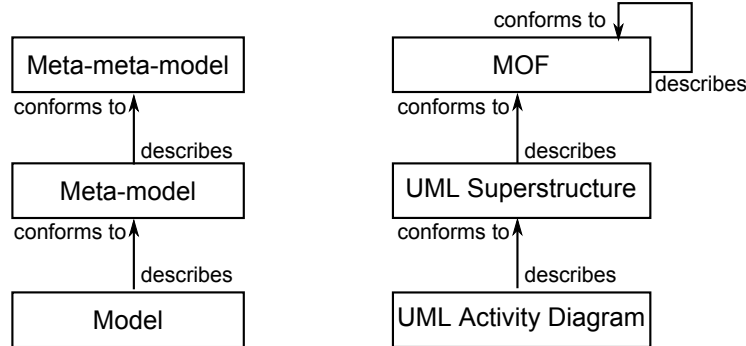


Figure 2.4: Meta-model based language definition stack in general (left) and for UML in MDA (right) (from [GPR06]).

Figure 2.4 shows this relation between a model and its meta-model in general (left), and on an example (right) of the main system modeling language of MDA – *Unified Modeling Language (UML)* [UML]. It contains a *UML Activity Diagram*, as a model, that is conform to the UML meta-model called *UML Superstructure* [UML]. The UML Superstructure is, in its turn, conform to its meta-model called *Meta-Object Facility (MOF)* [MOFa]. Both MOF and UML Superstructure meta-models were developed in the context of MDA. Note that, in general, a meta-meta-model can be defined by some other meta-model, which is not shown here. In MDA, however, MOF meta-model is defined through itself, which makes the language definition stack of MDA finite and relatively compact.

Now, we discuss the main types of elements a meta-model can contain in the MDA context. These types are defined by MOF, a fragment of which is shown in Figure 2.5. MOF describes the following main concepts: *class*, *association*, *property*, and *generalization/inheritance*. Since all the concepts described by MOF are also the concepts used to define it (see [MOFa]), we use these concepts to describe MOF. To model MOF and other example meta-models we use a widely-known realization of MOF called *Ecore* language, which is a part of *Eclipse Modeling Framework (EMF)* [EMF].

MOF contains two main classes (depicted by rectangles) – *Class* and *Association*. The class *Class* is used to model a set of objects that share the same properties and semantics. The class *Association* is used to model a set of links between the objects belonging to two (different) sets/classes. It can also define additional constraints on the number of elements in each set, which are called *multiplicities*.

The class *Class* has an association (depicted as a line) to the class *Property*, that is used to model different attributes/properties of classes. The class *Class* also has an

association to the class *Operation* (not shown in Figure 2.5), that is used to model different operations on classes. The class *Association* is also associated to the class *Property*, that is used to model different link (object of class *Association*) ends. The class *Property* has several attribute properties both inherited from super-classes like *MultiplicityElement* and its own *upper* and *lower multiplicity bounds* (natural numbers), *aggregation kind* (none, shared, or composite [UML]), and some other properties.

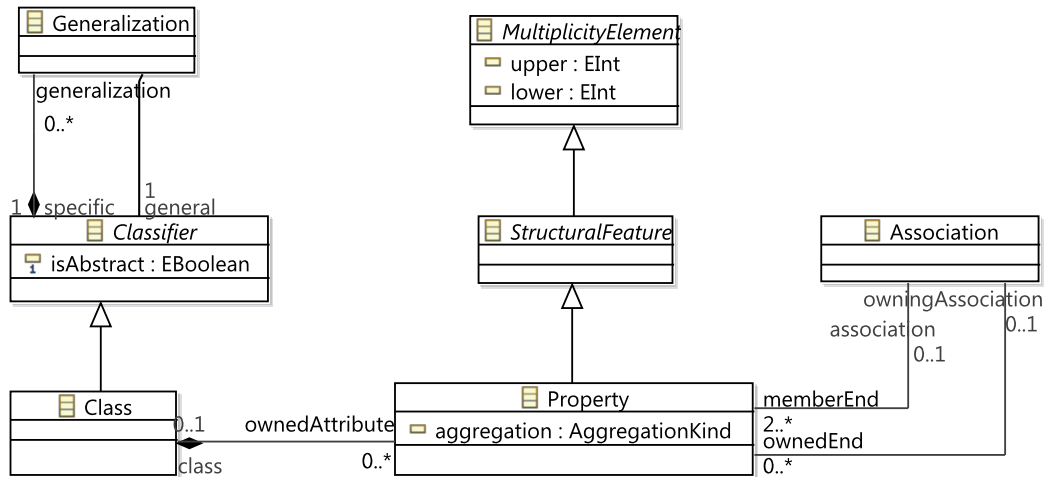


Figure 2.5: Main elements of MOF (from [MOFa]) in Ecore.

Finally, the concept of generalization/inheritance (depicted by a line with Δ arrow) is described in MOF by the class *Generalization*, that is associated to a super-class (*general* property) and a sub-class (*specific* property). Instead of a direct association with the class *Class* MOF uses an additional abstract class *Classifier*, that is a super-class of *Class*, to define the generalization. This class has a property *isAbstract* to represent abstract classes, whose names are then written in italic. Here, we leave out any further motivation for the introduction of this class into MOF.

Now, that we have described the main types of elements used in MDA to define a meta-model, we consider an example meta-model – a part of UML Superstructure for activity diagrams. This meta-model fragment, shown in Figure 2.6, describes different concepts of activity diagrams as objects of MOF classes. It contains classes like *Activity*, *ActivityEdge*, and *ActivityNode* to describe activities, activity edges, and nodes, respectively. The last two classes are abstract and have different sub-classes including *ControlFlow* and *ControlNode* classes to describe control flow edges and nodes respectively. The abstract class *ControlNode* has several sub-classes used to describe different concrete types of control flow nodes. These sub-classes include *InitialNode*, *ActivityFinalNode*, *ForkNode* and *JoinNode*, *DecisionNode*, and *MergeNode*.

The meta-model fragment in Figure 2.6 also contains associations between the class *Activity* and the classes *ActivityNode* and *ActivityEdge*, to describe sets of links between

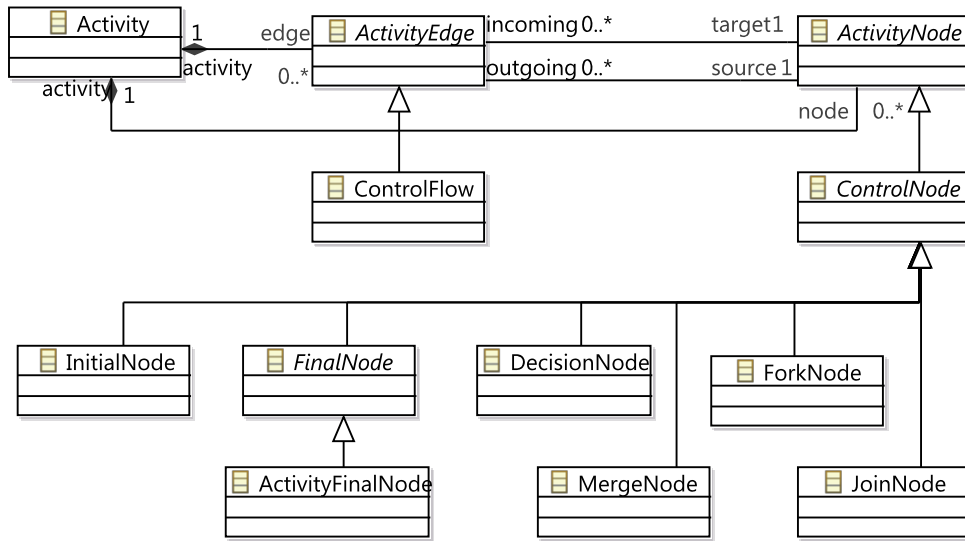


Figure 2.6: Fragment of the UML Superstructure for activity diagrams (from [UML]) in Ecore.

an activity and its nodes and edges. These associations have compositional nature (depicted by a filled diamond), that does not allow nodes and edges to exist without the containing activity. Furthermore, the classes *ActivityNode* and *ActivityEdge* are also directly linked via associations, to describe sets of links between nodes and edges that are in source/outgoing and target/incoming relations. All of the described associations have roles (property names) and multiplicities 1 or 0..* (represents a positive natural number) attached to their ends. For instance, in the top association between the classes *ActivityNode* and *ActivityEdge*, the class *ActivityNode* has a role *target* and the class *ActivityEdge* has an *incoming* edge role. These roles with the multiplicities 1 and 0..* require each edge to have exactly one target node and allow a node to have multiple incoming edges, respectively.

The discussed UML meta-model fragment describes models containing activities in terms of the allowed concepts and their abstract syntax. Figure 2.7 shows how a given simple activity model, described using some common concrete syntax for activities (left), is represented in the abstract syntax (right), defined by the meta-model from Figure 2.6. In this abstract syntax representation all elements of the activity model are depicted as *instances* of the meta-model classes and associations that they belong to. An instance of a class – an *object* – is represented, like a class, by a rectangle with its name, but here the name of the object is followed by the class name (separated via a colon) and they are underlined. An instance of an association – a *link* – is represented, like an association, by a line with a name, but here the name of the link is followed by the association name

(separated via a colon) and they are underlined. In our example, the majority of the instances do not have names¹.

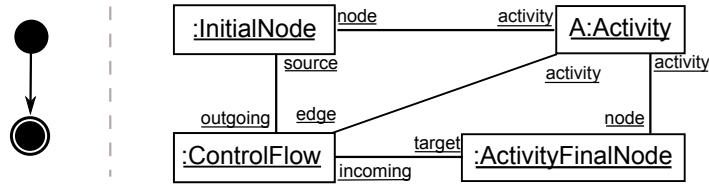


Figure 2.7: Simple activity model in concrete (left) and abstract (right) syntax

In Figure 2.7, we see how a sample activity model can be described in terms of instances of meta-model elements. The way these instances are connected has to comply with all type (including inheritance), association kind, and multiplicity constraints defined in that meta-model. In some cases, we would like to impose additional constraints on a model that can be described by a meta-model. For our activity diagrams example this can include the following structural well-formedness constraints:

1. an activity has exactly one initial and one final node;
2. an initial node has no incoming edges and a final node has no outgoing edges;
3. a decision node always has a corresponding merge node (the same for fork/join node pairs);
4. a merge/join node has exactly one outgoing edge.

These constraints are not covered by the meta-model in Figure 2.6. To add them to this language description, we can either change the meta-model (1), including new associations for *InitialNode* and *FinalNode* classes and new structuring classes, or, alternatively, use a separate constraint language (2). We briefly consider examples of these two options, that can be found in the UML Superstructure for activity diagrams.

As an example of the first option, we find some additional structuring classes that exist in the UML Superstructure. These classes include the following: *ActivityGroup*, *StructuredActivityNode*, *LoopNode*, *ConditionalNode*, and *SequenceNode* class. Figure 2.8 shows them² in relation to the *ActivityNode* and *ActivityEdge* classes discussed before. The first class – *ActivityGroup* – is used to represent sets of activity nodes and edges. The class *StructuredActivityNode* is its sub-class, that the UML Superstructure describes as follows: “A structured activity node represents a structured portion of the activity that is not shared with any other structured node, except for nesting.” ([UML], p. 423).

¹For simplification, we only separate names of objects/links from their class/association names via a colon, if at least one of the objects/links has a name.

²Since Ecore does not support modeling of *subsets* and *redefines* relations between associations (see [UML]), we add these details of the UML Superstructure to the names of the respective associations.

The remaining classes *LoopNode*, *ConditionalNode*, and *SequenceNode* are sub-classes of the *StructuredActivityNode* class, that represent the following concrete structures: loops, conditional/decision blocks, and sequences of nodes, respectively. These classes have additional explicit associations to the classes that define their structure, like *bodyPart* association of the *Loop* class, *clause* of the *ConditionalNode* class, and *executableNode* of the *SequenceNode* class. All these associations can be found in [UML], with the last one also shown in Figure 2.8. This last association links the class *SequenceNode* to the sub-class of *ActivityNode*, called *ExecutableNode*. This class provides new execution modeling possibilities which are, however, not relevant here.

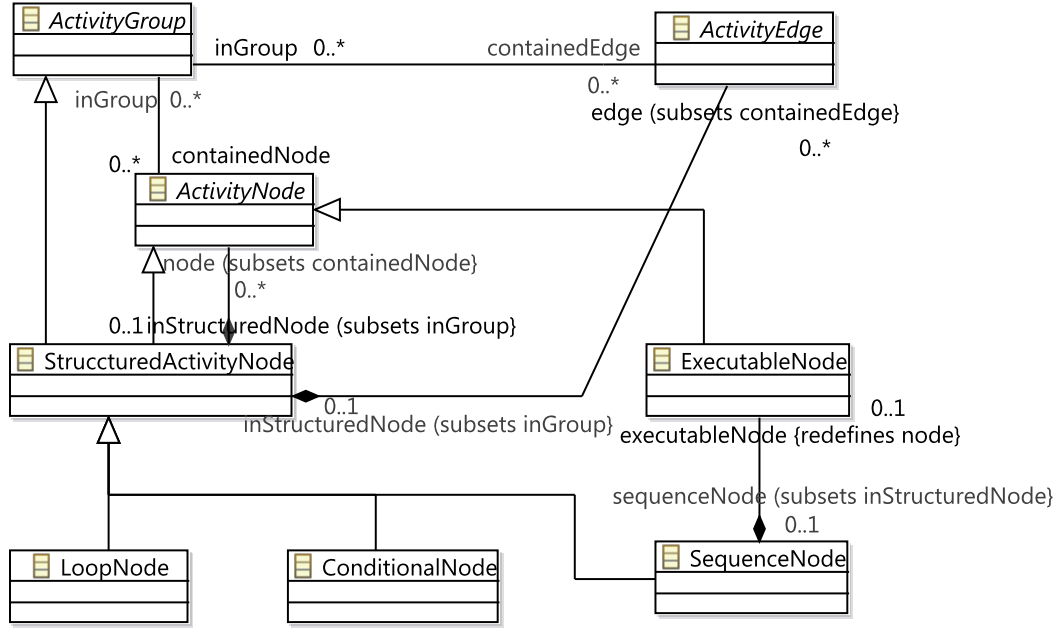


Figure 2.8: Fragment of the UML Superstructure with structuring classes in Ecore

An example of the second constraint definition option can also be found in the part of UML Superstructure for activity diagrams. There, a constraint language from the OMG called the *Object Constraint Language (OCL)* [OCLb], is used to express some well-formedness constraints. For instance, it is used to describe well-formedness constraint 4 for join nodes from the above list, as shown in Figure 2.9.

context JoinNode **inv** joinOneOutgoing: self.outgoing ->size() = 1

Figure 2.9: Well-formedness constraint 4 for a join node in OCL (from [UML], p. 394)

Constraint 4 for join nodes is expressed in OCL as an *invariant* (keyword *inv*) that is called *joinOneOutgoing*. An invariant represents an expression that must always be true

in the context (keyword *context*) of some class. Here, we have an OCL invariant in the context of the class *JoinNode*. The body of this invariant (described after the colon) states that, the number of outgoing edges for a join node (represented by the keyword *self*) must be equal to 1. The number of outgoing edges is obtained by using OCL function *size()*, that returns the number of objects in the collection it is called on (here *outgoing*).

In general, OCL which is also a part of MDA, is quite extensive. Apart from invariants, it allows to define further types of constraints like operation pre- and post-conditions, and attribute derivation rules. All of the defined OCL constraints then need to be checked for each model using an OCL engine (e.g., the one provided by [OCLa]), to ensure that the given model complies with them.

At this point, we conclude our discussion of the meta-model based definition of modeling languages in MDA. Next, we focus on the definition of the second most important concept in MDA – model transformations. We consider how they can be developed based on meta-models that describe their source and target languages.

2.3 Definition of Model Transformations

In MDA, model transformations are usually defined to support the step that creates a PSM of the developed system based on its PIM and the model of the chosen platform – PDM. This step, denoted by a 'gear', is shown in Figure 2.2. Now, we consider how a PIM-to-PSM transformation definition, required for this step, can be constructed.

A transformation between a PIM and its PSM has to define some kind of *mappings* between the elements of these models, based on the developers choice and the information provided by the PDM. In [MMea03], several approaches to the definition of such mappings are proposed – meta-model based mappings, model instance (object) based mappings, and a combination of the first two approaches.

In the first approach, sketched in Figure 2.10, the transformation mappings are defined between the elements of the PIM and the PSM meta-models. These mappings can also relate structures, described by some instances of the meta-model elements, also called *patterns*. The PIM and PSM meta-models, used as a basis for these mappings, are defined in MDA to describe the corresponding languages, as discussed in Section 2.2.

In the second approach, the transformation mappings are defined for specific objects of the transformed PIM. These objects are *marked* by some chosen concepts from the PDM by the developer, before the transformation is executed. For each given combination of a *mark* and the type of object it marks, a mapping is defined into some PSM concepts (meta-model pattern). These mappings are used during the transformation execution. We consider this approach to be a special form of the first approach, where an intermediate PIM meta-model with marker classes is added to the mappings definition chain and the corresponding marked PIM model is created manually, by refining the

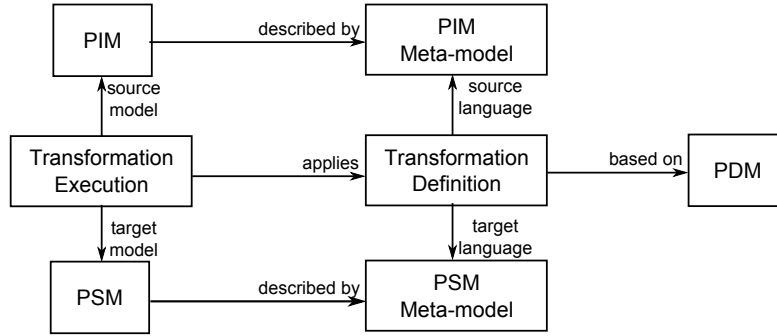


Figure 2.10: Meta-model based definition of transformation mappings (adapted from [MMea03])

original PIM model. Thus, we only consider the concepts of the first approach in further detail, and leave their integration into the second approach to the reader.

The discussed meta-model based transformation mappings are then described using some transformation language. This description may be in a natural language, an algorithm in an action language, or in a model mapping language [MMea03]. The *Query/View/Transformation (QVT)* [QVTb] specification from OMG describes the *de facto* language standard for the specification of model mappings in MDA.

QVT specification contains the following three related transformation languages: *Core*, *Relations*, and *Operational Mappings*. The first two languages are declarative. The *Relations* language is a user-friendly option for the specification of relations between MOF based models [QVTb]. These relations must hold for the transformation to be successful. The *Core* language is a simpler alternative to *Relations* with equivalent expressiveness. However, transformation described using *Core* language are more verbose. Both languages embody the same semantics at two different levels of abstraction. In the QVT specification *Core* is used as a reference for the semantics of the *Relations* language, which is mapped to *Core*, using the *Core* language itself.

The *Operational Mappings* language of QVT is a mechanism for describing and invoking imperative implementations of transformations from *Relations* or *Core* [QVTb]. Such imperative operational mappings are used to implement one or more relations, when it is difficult to provide their purely declarative specification. This allows to complement relational transformations with imperative operations implementing some relations, which represents a *hybrid approach*. An entire transformation can also be written in this imperative language, with some operational mappings invoking others, if needed.

The described QVT languages provide a reference and comparison point for the existing model transformation languages and their implementations. The model transformation languages closest to *QVT Relations* include the following: *mediniQVT* [QVTa], *QVTd* [QVTc], and *TGGs* [Sch94]. The closest ones to *QVT Operational Mappings* lan-

guage include the following: *QVTo* [QVTd], *Kermeta* [Ker], and *ATL* [JABK08, ATL], where the first two languages are purely imperative and the last one is a hybrid.

Next, we consider the detailed definition and implementation of the meta-model based transformation mappings, using some of the mentioned mapping languages. Since model transformation development and re-use are central to this thesis, to make this description more clear and demonstrative, we first introduce an elaborate PIM-to-PSM transformation case study in Chapter 3. Then, in Chapter 4, we use this detailed example to discuss systematic development of model transformations, including their implementation in languages like ATL and TGGs, and their quality properties and the existing re-use mechanisms.

Chapter 3

Introduction to Case Study: Transformation of Multi-Layer Systems

In this Chapter, we describe the example software system model and its transformation, that we are going to use throughout the rest of the thesis for demonstration purposes. In our running example, we transform a PIM model of a software system into its platform-specific representation (PSM), where the chosen platform is a formalism called *Communicating Sequential Processes (CSP)* [Hoa85]. This type of PIM-to-PSM transformation is used in MDSD (see Section 2.1) to obtain a formal specification of the modeled system, in order to check its properties like *deadlock-* and *livelock-freedom* [BWWB12], and other behavioral properties [EKHL03, BH07, XMP09]. Such transformation translates a model, describing system behavior using e.g., UML Activity Diagrams [UML], into its CSP specification. This specification is then used as input for the respective available CSP analysis tools (e.g., FDR [FDR], PAT [SLD08]).

We have chosen this PIM-to-PSM transformation example since its source and target languages contain not only simple atomic elements, but also high-level structures, built from several elements, like decision/merge or fork/join blocks in activity diagrams. Thus, these languages can be defined in an inductive way, as shown in [EKR⁺08, GLKE10, Sem12] (for activities) and [Hoa85] (for CSP). Furthermore, the chosen transformation requires mappings between high-level structures of these languages, as we see in Section 3.3. Plus, the results of the mappings need to be connected in a non-trivial way. Later, in Section 4.1.3, we see that the existing approaches to the development of model transformations struggle with such example requirements and, often, only offer complex and mostly imperative solutions without the guaranteed quality of the resulting model transformations. The latter question is also discussed in detail in Section 4.2.

Our transformation example represents one of the many PIM-to-PSM transformations with similar characteristics, like [FEK⁺07, GGMT08, KR10, BWW11, SE12, WW13], that can benefit from an improved approach to their development. They play a key role in MDSD for obtaining formal representations of the given systems, later subjected to analysis. The target formalisms used there include Markov chains [GGMT08, BWW11], layered queuing networks [KR10], FSP (Finite State Process) [FEK⁺07], and so on. The properties analyzed in these formalisms include, but are not limited to, reliability, performance, and deadlock-freedom, respectively.

In the rest of this Chapter we, first, describe our example software system and how it is modeled (Section 3.1). Then, we introduce the used formal specification language CSP in Section 3.2, and, finally, discuss the example PIM-to-PSM transformation in detail in Section 3.3.

3.1 Example System: Trip Planning Software (ADR)

For our running example, we consider a *trip planning software system* that relies on a map database. Its component-based architecture consists of the following multiple *layers*: an *application layer*, that contains all components for the application business logic, and a *platform layer*, that contains components that support inter-component communication and/or provide certain resources.

Each layer of the system is modeled in one or more PIM models, as shown in Figure 3.1. In our example, the application layer is represented by one model $M_{application}$ and the platform layer is split into the following two models: $M_{platform_RPC}$ model with the components providing the functionality for the RPC¹ based communication of components and model $M_{platform_threads}$ with the component providing the execution threads.

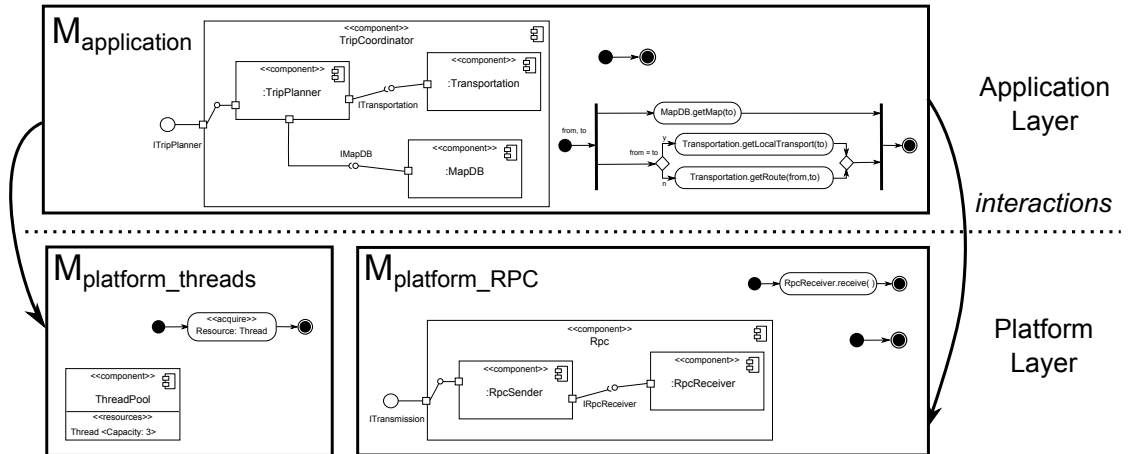


Figure 3.1: PIM models of the example multi-layer software system

Each PIM model describes a set of components modeled by means of UML Component Diagrams [UML], and the behavior of the operations they provide. The behavior is defined using a simplified version of UML Activity Diagrams with some additional concepts for modeling the resources, provided by components, and operations on them. These concepts are adopted from a component-based system architecture description language called *Palladio Component Model (PCM)* [BKR09, RBB⁺11]. We refer to the resulting behavior modeling language as *Activity Diagrams with Resources (ADR)*.

¹Remote Procedure Call (RPC) [BN84].

Now, we consider each of the PIM models of our example layered system in detail, and explain the relevant modeling concepts of UML Component Diagrams and of ADR, when they are first used. Later, we provide a meta-model that describes the ADR language.

Figure 3.2 shows a component diagram of $M_{application}$ model. It contains a set of interfaces and components. An interface contains a list of operations, and its role with respect to a component (*provided* or *required*) is defined by its relation to it.

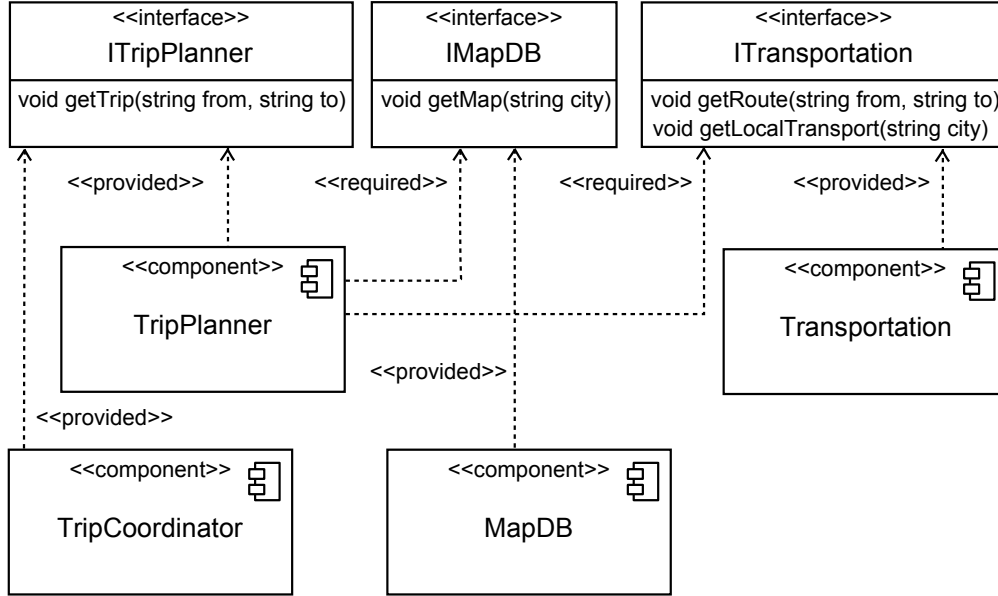
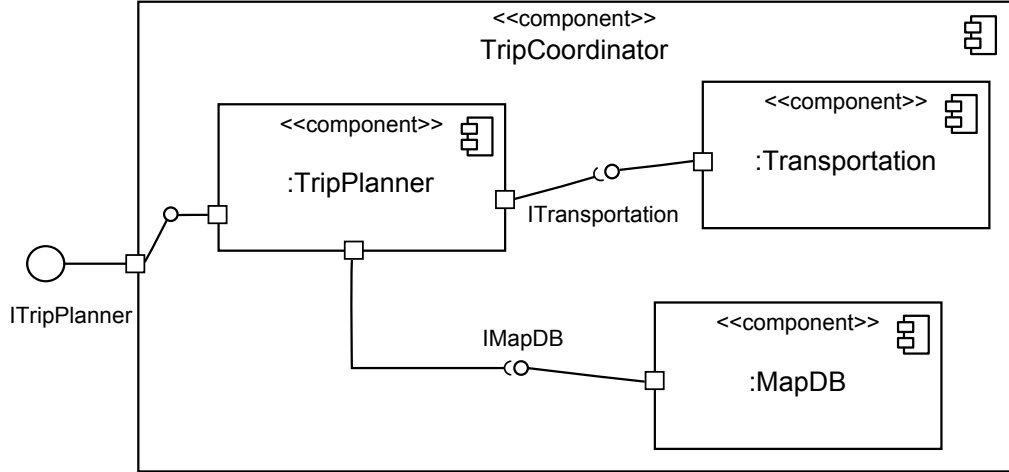


Figure 3.2: Component diagram of the $M_{application}$ model of the application layer

For instance, in our example, the interface *ITripPlanner* is provided by a *composite* component *TripCoordinator* and a *basic* component *TripPlanner*. The basic component requires interfaces *IMapDB* and *ITransportation*, that are provided by the basic components *MapDB* and *Transportation*, respectively. The difference between these two components is that the component *TripPlanner* has some external dependencies (required interfaces), whereas in the *TripCoordinator* component these dependencies have already been satisfied by the connected components. Figure 3.3 shows the inner structure of the *TripCoordinator* component assembled from the basic components *TripPlanner*, *MapDB*, and *Transportation*. It also shows the interface dependencies of the *TripPlanner* component that these basic components satisfy.

In our example, the interface *ITripPlanner* and, more precisely, its operation(s) are required by a system user. We refer to such operations as *entry* operations.

Next, we specify the behavior of the operations provided by the components from Figure 3.2 by using the ADR language. Figure 3.4 shows the following activities describing the behavior of the operations provided by the components of the application layer: *Trip-*

Figure 3.3: Structure of the composite component *TripCoordinator*

*Planner.getTrip()*² (3.4a), *MapDB.getMap()* (3.4b), *Transportation.getLocalTransport()* (3.4c), and *Transportation.getRoute()* (3.4d). The activity for *TripPlanner.getTrip()* operation is the only one with a non-trivial behavior in our example application. It represents a parallel execution of external call actions to the operations, provided by the *MapDB* and *Transportation* components, to obtain the map and the transportation information, respectively. The transportation information required is either local (*to = from*), in which case the operation *Transportation.getLocalTransport()* is called. Otherwise, a route from one city (*from*) to another (*to*) is computed by calling the operation *Transportation.getRoute()*.

Note that in ADR, an external call action is linked directly to the called operation provided by the respective component. This modeling short-cut is taken for the sake of simplicity, since the details of the resolution of an interface provider do not contain any analysis-relevant behavioral information and, thus, are not relevant for our transformation example. For the same reason, all *entry* operations provided by the respective components are also marked directly.

Next, we consider our example platform layer and the models $M_{platform_threads}$ and $M_{platform_RPC}$ representing it. Figure 3.5 shows component diagrams of both of these models. The interface *IThreadPool* and its providing component *ThreadPool* constitute the first diagram. The component *ThreadPool* owns a resource *Thread* of a limited capacity of 3 (3 threads), that is modeled using a *passive resource* modeling concept adopted from PCM. This component also provides operations *IThreadPool.acquire()* and *IThreadPool.release()* that allow acquisition and release of a thread, respectively.

The rest of Figure 3.5 shows interfaces and components of the $M_{platform_RPC}$ model. They provide a high-level view of the functionality used to support RPC communication

²To keep the description concise, we omit the parameters of the operations in the text.

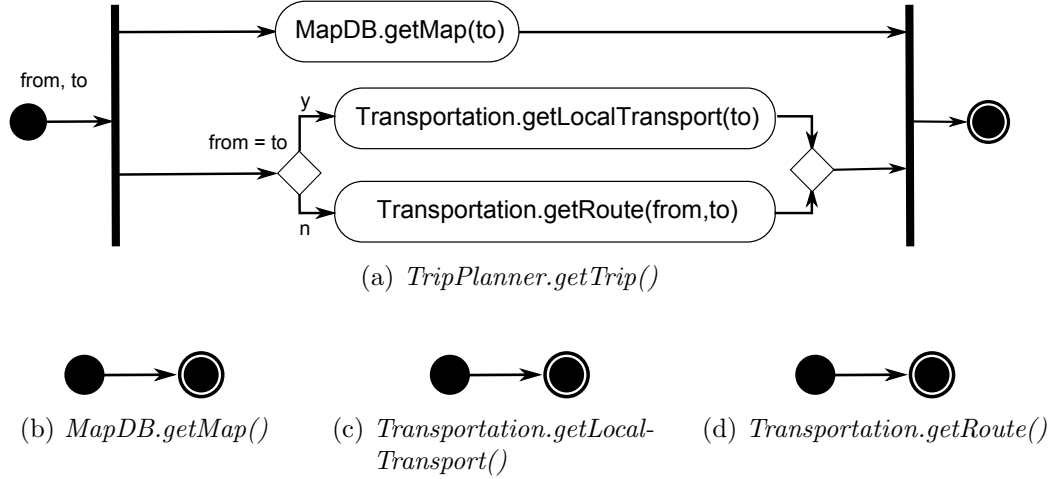


Figure 3.4: Activities modeling the behavior of the operations of the application layer

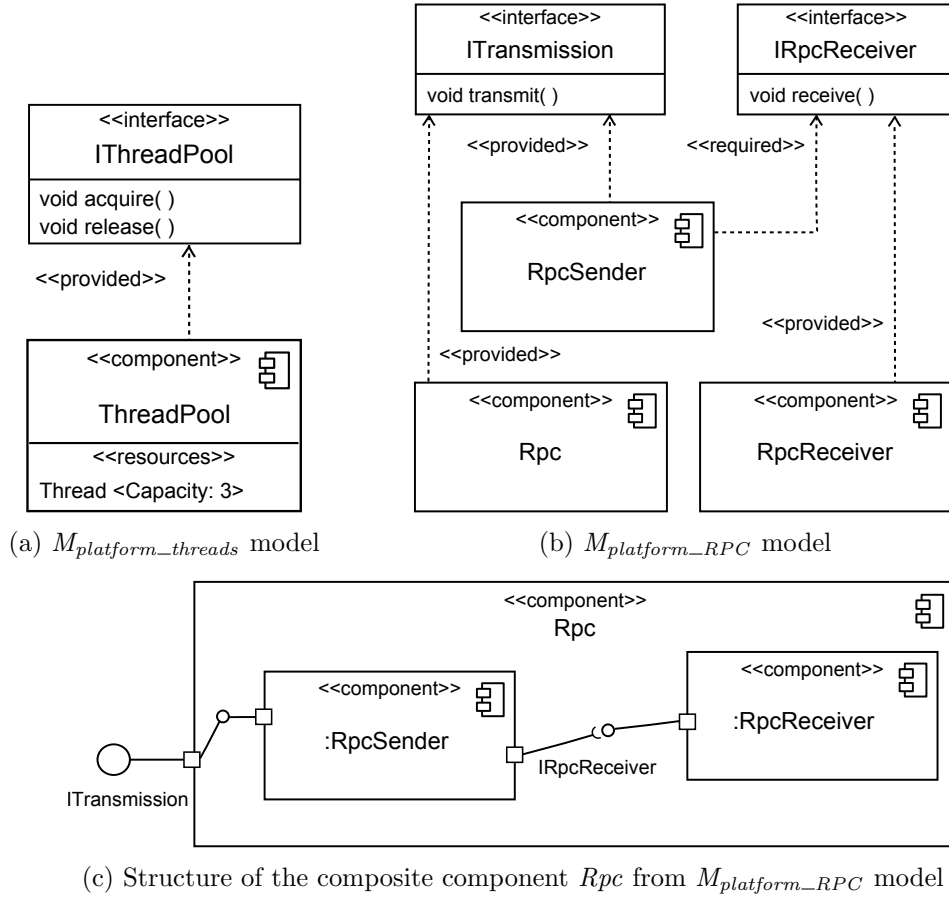


Figure 3.5: Component diagrams of the PIM models of the platform layer

between the components. The interface *ITransmission* contains the operation *transmit()* that is used to start an RPC communication. This operation is provided by the *RpcSender* component that, in its turn, requires a component that provides the *IRpcReceiver* interface to complete the communication process. The component *RpcReceiver*, in its turn, provides this *IRpcReceiver* interface with its operation *receive()*. Like the application layer model (component *TripCoordinator*, Figure 3.3), this diagram contains a composite component called *Rpc* (Figure 3.5c). This component assembles the *RpcSender* and the *RpcReceiver* components based on their interface dependencies, into a self-contained component that provides the *ITransmission.transmit()* operation.

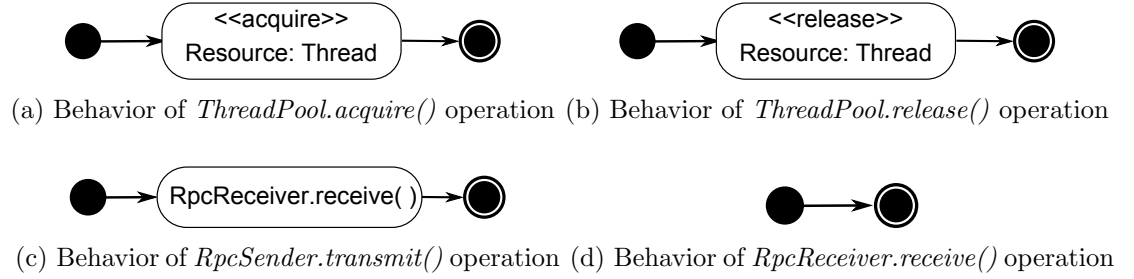


Figure 3.6: Activities for the operations provided by the platform layer

Next, we specify the behavior of the operations provided by the components from Figure 3.5 by using the ADR language. Figure 3.6 shows the following activities describing the behavior of the operations provided by the platform layer: *ThreadPool.acquire()*, *ThreadPool.release()*, *RpcSender.transmit()*, and *RpcReceiver.receive()*. The activities that describe the behavior of *ThreadPool.acquire()* and *ThreadPool.release()* operations contain a single action node each. These action nodes are used to model the acquisition of resource *Thread* (*acquire* action type) and its release (*release* action type). Like the modeling concept of a passive resource, these action types are also adopted from PCM.

The last non-trivial activity in Figure 3.6 describes the behavior of the *RpcSender.transmit()* operation. The RPC components in our example system are modeled on a very abstract level and their internal actions e.g., serialization and de-serialization are omitted. Thus, this activity only contains a single call action to the operation *receive()*, provided by the *RpcReceiver* component to complete the RPC communication.

We have considered models of the individual layers of our example software system. The information describing which operations of the higher application layer use which operations and resources of the lower platform layer also needs to be modeled. It can either be stored in the layers models or as a separate configuration model. At this point, we leave this question open and propose an approach, that supports the second alternative later, in Section 6.1.1.

We have used UML Component diagrams and a simplified version of UML Activity diagrams enriched with the concepts for modeling resources, we referred to as the ADR language, to model the structure and behavior of our example software system,

respectively. Since the PIM-to-PSM transformation we want to develop has its focus on translating models, describing the system behavior (in ADR), we next provide a detailed meta-model based description of this language.

ADR Meta-model As already discussed in Section 2.2, in the context of MDSD, modeling languages are defined in terms of meta-models. Now, we consider a meta-model based definition of our behavior modeling language ADR, that is shown in Figure 3.7.

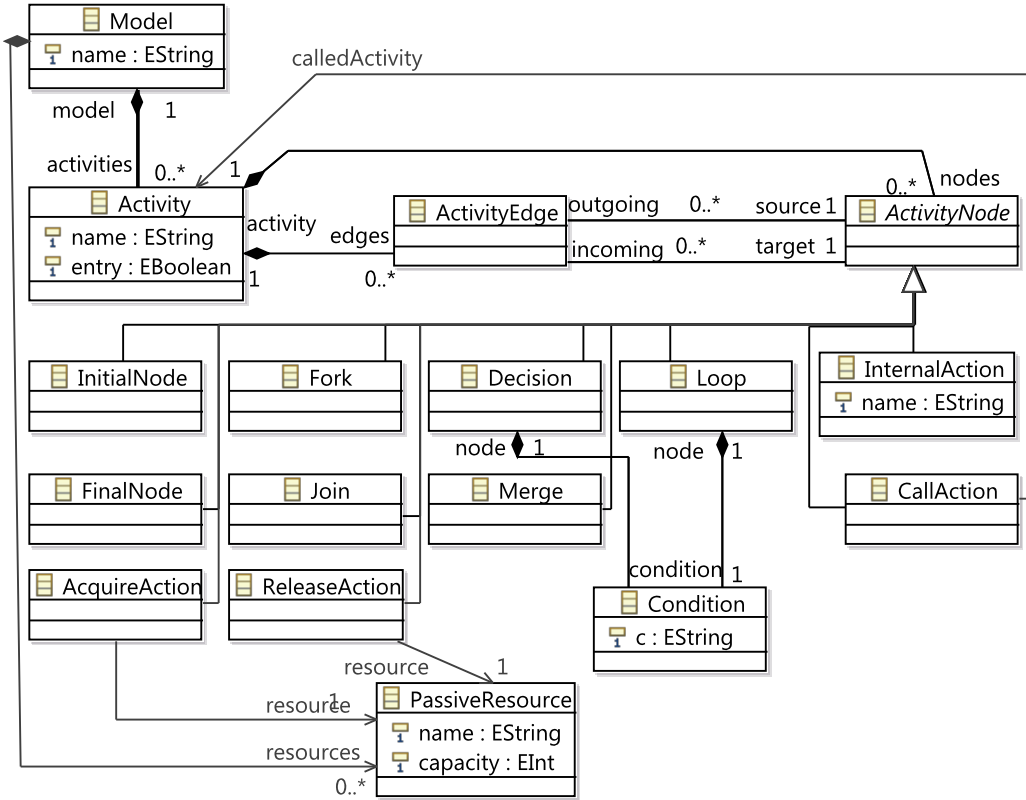


Figure 3.7: Meta-model for the Activity Diagrams with Resources (ADR) language

This meta-model contains the following allowed ADR model element types, represented by classes with attributes: class *Model* represents a whole ADR model, class *Activity* represents a single (entry, attribute *entry*) activity, classes *ActivityNode* and *ActivityEdge* represent an activity node and an activity edge, respectively; different sub-classes of *ActivityNode* represent different sub-types of an activity node; and class *PassiveResource* represents a resource of limited capacity (attribute *capacity*).

Sub-classes of class *ActivityNode* include the following: classes *InitialNode* and *FinalNode* representing, respectively, an initial/start and a final/stop node of an activity;

classes *Decision* and *Merge* representing, respectively, the beginning and the end of a conditional activity block; classes *Fork* and *Join* representing, respectively, the beginning and the end of an activity block with synchronized parallel node sequences; class *Loop* representing the conditional beginning and end of a repeating activity block; classes *CallAction* and *InternalAction* representing a call of an operation modeled by another activity and an internal processing node, respectively; and classes *AcquireAction* and *ReleaseAction* representing an acquisition and a release of a resource, respectively. Two sub-classes of *ActivityNode* class – *Decision* and *Loop* – have a condition modeled in the meta-model by class *Condition* with the conditional expression attribute *c*.

The meta-model in Figure 3.7 also contains allowed types of interrelations/links between the elements of an ADR model. They are modeled as named associations with the related constrictive multiplicities. These associations allow an ADR model to have an arbitrary number of *activities* and *resources*. Each activity can have an arbitrary number of *nodes* and *edges* between those nodes. Additionally, each edge has to have exactly one *source* and one *target* node, and a node can have multiple *incoming* and *outgoing* edges.

The associations defined on an *ActivityNode* class propagate to its sub-classes. Additionally, a *loop* and a *decision* node have to have a linked *condition* (object of class *Condition*); a *call action* has to be linked to exactly one activity (instance of *calledActivity* association) that models the called operation; and an *acquire* and a *release* action have to be linked to exactly one *resource* each. In the cases when an association between two classes has a *compositional nature* (see Section 2.2 for details), its conditions (an instance of the contained class has to belong to an instance of its container class) has to be respected in an ADR model.

In its present form, our ADR meta-model allows definition of models that we, however, do not consider to be valid. For instance, a model where an activity has none or more than one initial node. To forbid such malformed models, our ADR modeling language states some *well-formedness* conditions on models, including the following:

1. An initial node has no incoming edges and exactly one outgoing edge;
2. A final node has no outgoing edges and exactly one incoming edge;
3. An activity has exactly one initial and one final node;
4. A decision node has exactly one corresponding merge node;
5. A fork node has exactly one corresponding join node;
6. A decision/fork node has exactly one incoming node;
7. A merge/join node has exactly one outgoing node;
8. A loop node has exactly one repeating body activity block connected to it.

The structural conditions 1 – 3 and 6, 7 are quite simple and straightforward to express in the OCL language, that we have introduced in Section 2.2. Due to their similarity, we only show the first one in Figure 3.8 as an example. It is expressed by a set of two OCL invariants in the context of the class *InitialNode* – one for the number (*size()*) of incoming and one for the number of outgoing edges, respectively. Unlike this, the conditions 4, 5, and 8 are quite complex and require iterations and/or recursion for their definition in OCL (as queries). UML Superstructures provides some examples of such recursive OCL queries, defined for similar purposes – *containingStatemachine* (p. 558) and *ancestor* (p. 574) queries. Thus, we refrain from giving similar examples here.

context InitialNode **inv** noIncoming: self.incoming ->size() = 0

context InitialNode **inv** oneOutgoing: self.outgoing ->size() = 1

Figure 3.8: Well-formedness constraints on an initial activity node in OCL

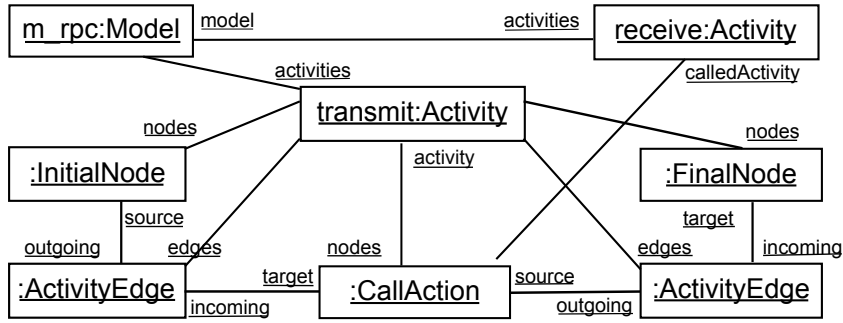


Figure 3.9: ADR model fragment with the activity *RpcSender.transmit()* as an instance of the meta-model from Figure 3.7 (abstract syntax)

To complete the picture, we show an instance that is conform to our ADR meta-model in Figure 3.9. This instance model describes an activity *RpcSender.transmit()* from our example system (see Section 3.1) in abstract syntax. For simplification, names of all model elements are shown before their types separated via a colon (':') and values of other attributes are hidden. We can see that this instance fulfills all of the applicable well-formedness conditions specified before.

Next, we switch to our target language. We provide an introduction into the CSP formalism, into which we are going to transform our multi-layer system, in order to analyze its behavioral properties like deadlock- and livelock-freedom. In terms of MDSD (see Section 2.1), this formalism is a specific platform with related tools, that require a suitable platform-specific representation of our example system model.

3.2 Example Platform: Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) [Hoa85]) is a formal notation that allows to specify behavior of *processes* within a system, which can then be analyzed with respect to various properties, like deadlock- and livelock-freedom (in our case), using available tools (e.g., FDR [FDR]).

CSP processes are essentially defined by sequences of atomic *events* they can participate in. These events can represent method calls, internal computations, and so on. They can also be equipped with *input* and *output parameters*, e.g., $ev.v$ denotes an occurrence of event ev with an output parameter with value v and $ev?x$ denotes an event ev reading some input into an input parameter variable x . The variable x can then be used in the rest of the process, for instance, also as an output in $ev.x$.

When an event ev has a parameter that can take different values, in CSP these values are stored into a so called event *channel*. For example, if an event ev appears in a process description with two different parameter values $ev.v1$ and $ev.v2$, the channel for the event ev then contains both of them as follows: `channel ev : {v1, v2}`.

In CSP processes are built out of events, described above, and out of other processes using various composition operators, including the following:

- $ev \rightarrow P$ – The *event prefix* operator states that the event ev is followed by the process P ;
- $P ; Q$ – The *sequential* process composition operator states that the process P is followed by the process Q ;
- $P \parallel_X Q$ – The *parallel* process composition operator states that the process P is executed interleaved with the process Q , and the processes are synchronized on the events from the synchronization alphabet set X . The alphabet X can contain individual events like $ev.v1$ as well as complete event channels like ev , also denoted by $ev.x$. Here, the parameter variable x can take any value from the pre-defined channel value set ($v1$ and $v2$ for our event ev);
- $P \square Q$ – The *external choice* operator specifies different potential behaviors of a process and states that it can behave as the process P or the process Q . Which behavior is selected is determined externally by the environment of the process.
- **if** c **then** P **else** Q – The *conditional choice* operator specifies different behaviors of a process upon condition c and states, that it behaves as the process P , if c is *true* and as the process Q , otherwise.

Termination of a CSP process is indicated by the special process *SKIP*.

Next, we consider some example CSP specifications. For instance, the recursively defined process $\text{THREAD} = \text{acquire} \rightarrow \text{release} \rightarrow \text{THREAD}$ is constructed using two event prefix operators and the THREAD process itself. The THREAD process then participates in an infinitely alternating sequence of these *acquire* and *release* events.

In another example, the process $\text{RPC} = (\text{send} \rightarrow \text{deliver} \rightarrow \text{RPC}) \sqcap \text{FAIL}$ with $\text{FAIL} = \text{send} \rightarrow \text{FAIL}$ introduces an alternating sequence of *send* and *deliver* events as well as an infinite loop, where the process can only participate in the *send* event. The choice between these alternative behaviors is then done externally and non-deterministically.

In CSP, if two processes can participate in the same events, it is possible to *synchronize* them on these events by using the parallel composition operator $\parallel_X$ with a non-empty *synchronization alphabet* X . For instance, two processes $\text{MAPDB} = \text{acquire} \rightarrow \text{getMap} \rightarrow \text{release} \rightarrow \text{MAPDB}$ and $\text{THREAD} = \text{acquire} \rightarrow \text{release} \rightarrow \text{THREAD}$ can be synchronized on the synchronization alphabet $X = \{\text{acquire}, \text{release}\}$ in the process $\text{SYSTEM} = \text{MAPDB} \parallel_X \text{THREAD}$. A process in this parallel composition can only participate in an *acquire* or a *release* event, if the other process can participate in the same event at that point.

Furthermore, CSP allows parametrization of a process description. A parametrized process description can contain one or more parameters. The values of these parameters are set when an instance of the described process is used in some other process definition. For example, the process TPOOL given below specifies a process that manages a maximum of *max* threads (resources). This process is an instance of another process TP , whose definition is parametrized by the current number of managed threads. It can have the value of 0 , n , or *max* - the maximum number of managed threads.

$$\begin{aligned} \text{TPOOL} &= \text{TP}(\text{max}) \\ \text{TP}(0) &= \text{release} \rightarrow \text{TP}(1) \\ \text{TP}(n) &= (\text{acquire} \rightarrow \text{TP}(n-1)) \sqcap (\text{release} \rightarrow \text{TP}(n+1)) \\ \text{TP}(\text{max}) &= \text{acquire} \rightarrow \text{TP}(\text{max}-1) \end{aligned}$$

Process TP then contains the actual logic for the threads management, that decrements or increments the number of currently available threads n on an *acquire* or *release* event, respectively. It regards the limited nature of threads as a resource and exposes different behaviors for different parameter values. When the current number of managed threads is 0 , only a *release* event can be processed (participated in) and, when the current number of managed threads is *max*, only an *acquire* event can be processed. The values of n and *max* are natural numbers.

After this short introduction into CSP, we can next consider how our layered trip planning software system can be represented using this formalism. Later, we use this transformation as our running example to discuss how it can be developed and re-used. For this purpose, we now provide a detailed meta-model based description of the CSP concepts that we use.

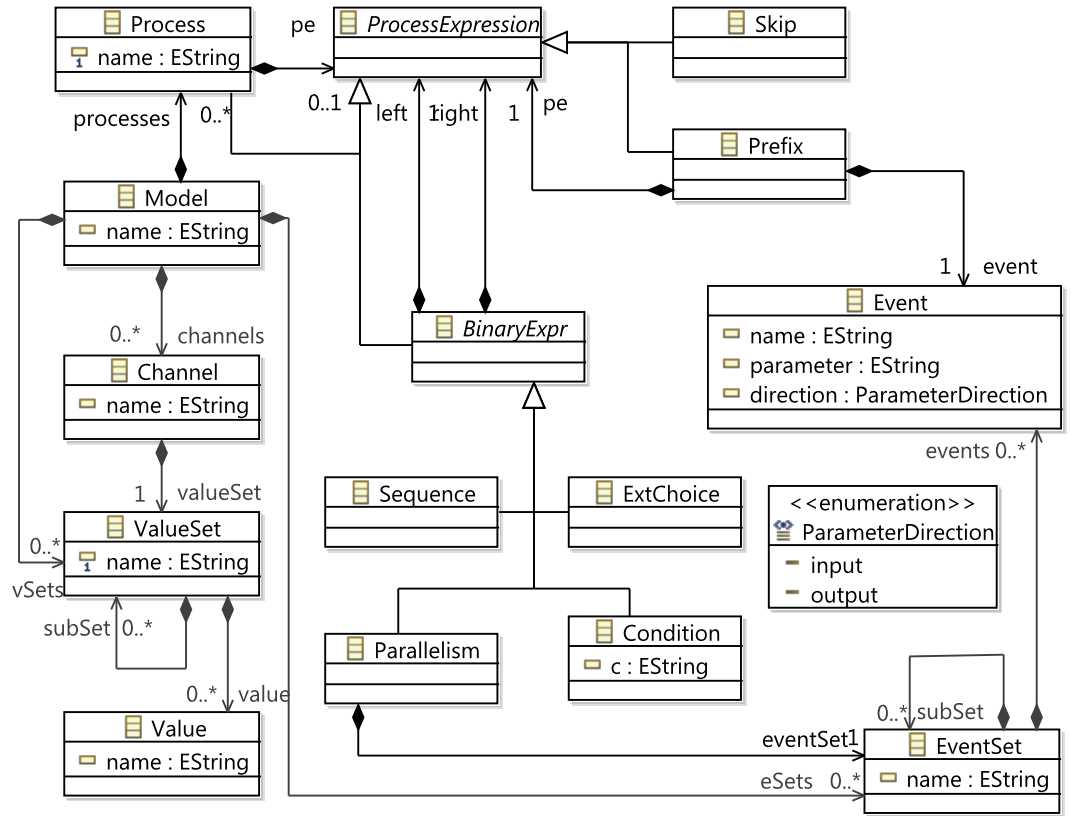


Figure 3.10: Meta-model for the CSP modeling language

CSP Meta-model Several versions of meta-model based description of CSP have been provided in the literature, including [KSW04, VAB⁺08, BWB12]. These descriptions are all based on [Hoa85], yet they have different degree of detail, often varying and unintuitive naming conventions, and different scopes. The meta-model described in [VAB⁺08] appears to have the closest resemblance with the concepts from the original CSP language specification [Hoa85]. Therefore, we use it as a basis for our CSP meta-model and add/remove further classes and associations, defining the required concepts and transferring our understanding of the original CSP specification into the resulting meta-model. For instance, we add the constructs to describe CSP channels, process sequencing, and so on.

Figure 3.10 shows the version of the CSP meta-model that we use. For simplification, it only contains the classes that represent concepts that are relevant for our transformation example. These notions were discussed in the CSP introduction above, and we, thus, do not describe the classes of this meta-model in detail here. We just note that the classes have the names of these concepts or very similar names, and the abstract classes are only used in this meta-model to structure process expressions and to avoid the repetition of

associations for binary process operators. This CSP meta-model does not have any additional OCL constraints attached to it.

Since we have already shown two meta-model instances in Section 2.2 and in Section 3.1, we refrain from showing one for this CSP meta-model and proceed to discussing our PIM-to-PSM (ADR-to-CSP) model transformation.

3.3 Example Transformation: ADR-to-CSP

There are multiple possibilities to define a transformation to translate (representation change, see Section 4.1.1) models from ADR (or another UML Activity diagram like language) into CSP, including [Scu05, BH07, VAB⁺08, XMP09, BSS12]. All these works present model transformation descriptions that sometimes syntactically vary by using different CSP constructs to represent the same concepts of activity diagrams (via mappings). Semantically, however, they aim to represent the same behavior in CSP as defined in the translated input source models. None of these approaches actually prove this semantic equivalence, since such elaborate task, that requires complex techniques [HKR⁺10, Sem12], is not in their focus.

Next, we describe a possibility to define an ADR-to-CSP transformation, that we are going to use as an example, when we later discuss different transformation development approaches. We provide all details of this transformation alternative and demonstrate them by means of several examples, that show how different parts of ADR models of our multi-layer trip planning system from Section 3.1 should be transformed into CSP.

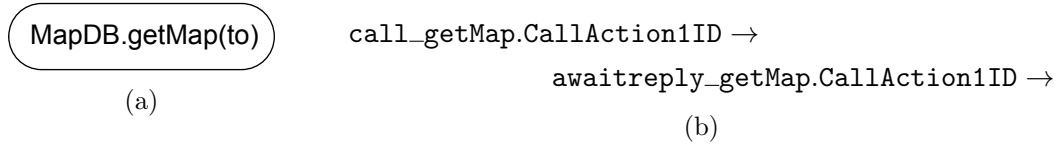
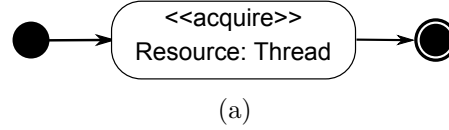


Figure 3.11: Call of the operation *MapDB.getMap()* (a) and a part of the result of its transformation into CSP (b)

First, we consider one of the call actions from the activity *TripPlanner.getTrip()* from the model $M_{application}$ – the one that calls the *MapDB.getMap()* operation. Figure 3.11 shows this action and a part of the CSP specification it is transformed into. The call action is transformed into a process expression, that consists of two event prefixes that represent a call of the operation (event *call_getMap*) and an expectation of its response (event *awaitreply_getMap*). Both of the events in the prefixes have a parameter that stores an *identification (ID)* of the operation caller. The direct caller of the operation is the call action itself and, thus, its ID (in this case *CallAction1ID*) is used. The rest of the CSP specification, generated from a call action, including a process, channels, and so on, is considered later when the complete transformation requirements are summarized.



`ACQUIRE = invoke_acquire?x → call_acquireThread.AcquireActionID →`
`awaitreply_acquireThread.AcquireActionID → reply_acquire.x → ACQUIRE`
 (b)

Figure 3.12: Activity *ThreadPool.acquire()* (a) and a part of its transformation into CSP (b)

Now, we consider the complete activity for the operation *ThreadPool.acquire()* from the model $M_{platform_threads}$ of our example system. It is shown together with a part of its transformation into CSP in Figure 3.12. The activity is transformed into a recursive CSP process. The definition of this process, called *ACQUIRE*, consists of four consecutive event prefixes followed by a reference to itself. The events of the prefixes represent an invocation (event *invoke_acquire?x*) and a response (event *reply_acquire.x*) of the *ThreadPool.acquire()* operation, and a call for a thread acquisition (event *call_acquireThread*) and an expectation of its result (event *awaitreply_acquireThread*).

The first two mentioned events are the transformation results for the start and the end actions of the activity respectively. They have a variable x as their input (*ev?x*) or output (*ev.x*) parameter, which is used to store the ID of the caller of the *ThreadPool.acquire()* operation that this activity models. The remaining two events also have a parameter, but its value is already defined as the ID of the caller of the thread acquisition – the acquire action itself (here *AcquireActionID* is its ID).

Note that in both examples the call action and the resource handling action (acquire action) are transformed in the following very similar way: both actions are represented by a *call event* followed by a *response expectation event*. This way, we aim to represent different types of interactions between activities and between activities and passive resources within a layer model in a universal way. In order to connect these call and response expectation events with the events representing the actual *invocation* and *reply* of the called activity/resource handling operation, we add so-called *call connection* processes to our CSP transformation result.

An example of such call connection process for the call action from Figure 3.11a is shown in Figure 3.13. It relates the events that the call action is transformed into (see Figure 3.11b) to the invocation and response events of the *MapDB.getMap()* activity, which were generated from its start and end nodes (see Figure 3.12a). The events describing this call connection process are also added to the synchronization alphabet, which is used to compose the call connection process with the process that contains the result of the transformation of the call action. Further constructs added to the generated CSP specification for this and other activity node types are discussed in details later.

```

CALLACTION1ID_CALLS_GETMAP =
    call_getMap.CallAction1ID → invoke_getMap.CallAction1ID →
    reply_getMap.CallAction1ID → awaitreply_getMap.CallAction1ID →
CALLACTION1ID_CALLS_GETMAP

```

Figure 3.13: Call connection process in CSP for the sub-activity from Figure 3.11a

Before providing a summary of the requirements for our ADR-to-CSP transformation, we need to consider one more example transformation of a passive resource, that can be used via acquire/release actions, into CSP. We represent each passive resource by an instance of the corresponding parametrized managing process (like *TPOOL*, see Section 3.2). This instance is parametrized by the actual capacity of the resource defined in the model.

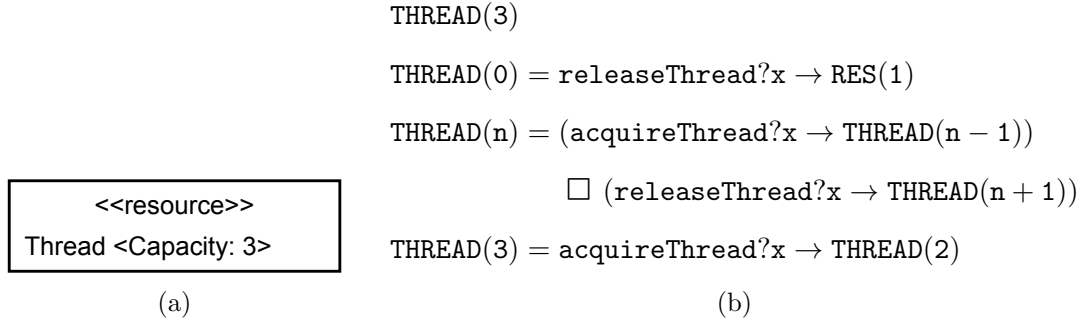


Figure 3.14: Resource *Thread* of the component *ThreadPool* (a) and a part of its transformation into CSP (b)

Figure 3.14 shows a part of the desired transformation for our *Thread* resource with the capacity of 3 threads into CSP. It contains the managing process *THREAD* and its instance with the desired capacity – *THREAD(3)*. The managing process *THREAD* is very similar to the *TP* example introduced in Section 3.2. In the former, the contained events also include the resource name (e.g., *releaseThread*) and have input/output parameter *x*, used for the synchronization with other processes, as discussed later.

Now, after considering several examples of the required transformation for different parts of our layered system, we can summarize these requirements in a concise and complete form in Table 3.1. This requirements overview abstracts from the details of individual examples and provides a general schema of the transformation we want to develop. It contains all source ADR language constructs considered by our example transformation. Thereby, we have reduced a number of interleaved activity blocks in a fork/join structure to two, since all other cases can be represented by combinations of this structure or reduced to a simple block. The provided CSP specifications for the

considered ADR language constructs do not represent a single possible result for each of them, but a class of semantically equivalent CSP expressions that the transformation can generate.

Next, we provide a short description and explanation of each of the requirements. In some cases, we use the notion of an *activity block* (marked by letter B), that represents a part of an activity (similar to single-entry-single-exit fragments in [VVL07]), and the notion of a *process expression* (marked by letters PE), that represents a sub-process in a process definition. The use of these notions with indices (when needed) allows us to keep the requirements description concise yet understandable. For the same purpose, we substitute the CSP operator *union* by the set union operator ($\cup$) in the CSP fragments.

The first requirement in Table 3.1 states which processes and an event set are to be created for an ADR model M , and how they are defined. The process M represents the complete model and consists of a parallel composition of process $M_PROCESSES$ and process M_CALLS synchronized via an event set X_M . The process $M_PROCESSES$ represents an interleaved composition of all processes individual activities and resources are transformed into, and the process M_CALLS represents an interleaved composition of all call connection processes (see example in Figure 3.13) created during the transformation of individual external call and resource handling actions. Both of these processes are initially assigned an empty CSP process $SKIP$. The synchronization set X_M contains all events relevant for the interaction of $M_PROCESSES$ and M_CALLS processes that represent external calls and resource handling operations. This set is initially empty and will be filled during the transformation in the way described by other requirements.

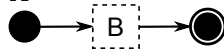
No.	ADR Model Element(s)	CSP Specification Fragment
1	Model M	$M = M_CALLS \parallel M_PROCESSES$ $M_PROCESSES \stackrel{X_M}{=} SKIP$ $M_CALLS = SKIP$ $X_M = \emptyset$ (synchronization events for calls / resource handling)
Activities		
2	Activity A	channel $invoke_a, reply_a : A_CALLERS$ $A_CALLERS = \emptyset$ $M_PROCESSES = (M_PROCESSES \parallel A)$
3	Non-empty activity 	$A = invoke_a?x \rightarrow PE \ ; \ reply_a.x \rightarrow A$

Table 3.1: Overview of requirements on the example ADR-to-CSP transformation

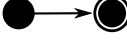
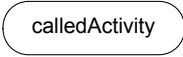
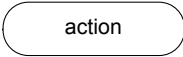
No.	ADR Model Element(s)	CSP Specification Fragment
4	Empty activity A 	$A = \text{invoke_a?x} \rightarrow \text{reply_a.x} \rightarrow A$
5	Entry activity A	$A_CALLERS = A_CALLERS \cup \{\text{externalID}\}$
Activity Nodes		
6	Call action 	$\text{call_calledActivity.callerID} \rightarrow$ $\text{awaitreply_calledActivity.callerID} \rightarrow$ $(\text{caller is the } \textit{CallAction})$ $\text{CALLERID_CALLS_CALLED_ACTIVITY} =$ $\text{call_calledActivity.callerID} \rightarrow$ $\text{invoke_calledActivity.callerID} \rightarrow$ $\text{reply_calledActivity.callerID} \rightarrow$ $\text{awaitreply_calledActivity.callerID} \rightarrow$ $\text{CALLERID_CALLS_CALLED_ACTIVITY}$ $M_CALLS = M_CALLS$ $\parallel \text{CALLERID_CALLS_CALLED_ACTIVITY}$ $X_M = X_M \cup \{\text{call_calledActivity.callerID},$ $\text{awaitreply_calledActivity.callerID},$ $\text{invoke_calledActivity.callerID},$ $\text{reply_calledActivity.callerID}\}$ $\text{channel call_calledActivity},$ $\text{awaitreply_calledActivity} : \{\text{callerID}\}$ $\text{CALLED_ACTIVITY_CALLERS} =$ $\text{CALLED_ACTIVITY_CALLERS} \cup \{\text{callerID}\}$
7	Internal action of activity A 	$\text{action.x} \rightarrow$ $\text{channel action} : A_CALLERS$

Table 3.1: Overview of requirements on the example ADR-to-CSP transformation

No.	ADR Model Element(s)	CSP Specification Fragment
8	Acquire action <<acquire>> Resource: Res	<pre> call_acquireRes.callerID → awaitreply_acquireRes.callerID → (caller is the <i>AcquireAction</i>) CALLERID_CALLS_ACQUIRE_RES = call_acquireRes.callerID → acquireRes.callerID → awaitreply_acquireRes.callerID → CALLERID_CALLS_ACQUIRE_RES M_CALLS = M_CALLS CALLERID_CALLS_ACQUIRE_RES X_M = X_M ∪ {call_acquireRes.callerID, awaitreply_acquireRes.callerID, acquireRes.callerID} channel call_acquireRes, awaitreply_acquireRes : {callerID} ACQUIRE_RES_CALLERS = ACQUIRE_RES_CALLERS ∪ {callerID} </pre>

Table 3.1: Overview of requirements on the example ADR-to-CSP transformation

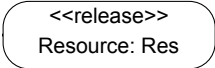
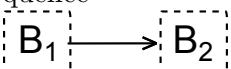
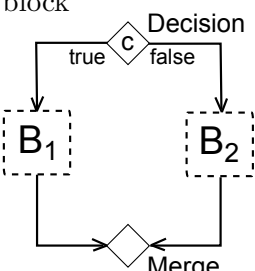
No.	ADR Model Element(s)	CSP Specification Fragment
9	Release action 	<pre> call_releaseRes.callerID → awaitreply_releaseRes.callerID → (caller is the <i>ReleaseAction</i>) CALLERID_CALLS_RELEASE_RES = call_releaseRes.callerID → releaseRes.callerID → awaitreply_releaseRes.callerID → CALLERID_CALLS_RELEASE_RES M_CALLS = M_CALLS CALLERID_CALLS_RELEASE_RES X_M = X_M ∪ {call_releaseRes.callerID, awaitreply_releaseRes.callerID, releaseRes.callerID} channel call_releaseRes, awaitreply_releaseRes : {callerID} RELEASE_RES_CALLERS = RELEASE_RES_CALLERS ∪ {callerID} </pre>
Control Flow Elements		
10	Activity block sequence 	$PE_1 \circ PE_2$
11	Decision/Merge block 	$\text{if } c \text{ then } PE_1 \text{ else } PE_2$

Table 3.1: Overview of requirements on the example ADR-to-CSP transformation

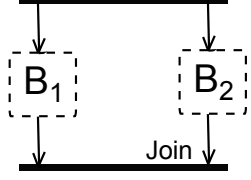
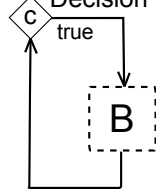
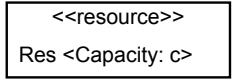
No.	ADR Model Element(s)	CSP Specification Fragment
12	Fork/Join block 	$(PE_1 \parallel PE_2)$
13	Loop block 	$L(x) = \text{if } x \text{ then } PE \circ L(x)$ else SKIP $L(c)$
Further Elements		
14	Resource in model 	$M_PROCESSES = (M_PROCESSES \parallel RES(c))$ $RES(0) = \text{releaseRes?}x \rightarrow RES(1)$ $RES(n) = (\text{acquireRes?}x \rightarrow RES(n-1))$ $\quad \square (\text{releaseRes?}x \rightarrow RES(n+1))$ $RES(c) = \text{acquireRes?}x \rightarrow RES(c-1)$ $\text{channel acquireRes : ACQUIRE_RES_CALLERS}$ $\text{channel releaseRes : RELEASE_RES_CALLERS}$ $ACQUIRE_RES_CALLERS = \emptyset$ $RELEASE_RES_CALLERS = \emptyset$

Table 3.1: Overview of requirements on the example ADR-to-CSP transformation

Requirements 2 – 5 describe different aspects of the activity transformation. Requirement 2 states which CSP channels should be created for each activity A to store all possible parameter values of the events *invoke_a* and *reply_a* in the value set $A_CALLERS$. The set $A_CALLERS$ of callers of the activity is initially empty and will be filled during the transformation as described by other requirements considered later. This requirement also states that the process A , that represent the activity, should be added to the definition of the $M_PROCESSES$ process via a parallel composition. The definition of the process A is described by requirements 3 and 4.

Both requirement 3 and 4 state that a recursive CSP process A , that starts with an event $invoke_a?x$ and ends with an event $reply_a.x$, should be created for an activity A . If the activity is not empty (requirement 3) and, thus, there is the rest of the activity structure that we represent by an underspecified block B , then we add a process expression PE to the definition of the process A . Both the block B and the process expression PE are place holders for the rest of the ADR model/CSP specification, that are implicitly related by this requirement. This relation indicates that the result of the transformation of the model part represented by B should be placed in the resulting CSP specification on the position of PE .

The requirement 5 states how a set of callers of an activity $A - A_CALLERS$ – should be extended by an external caller ID ($externalID$) for an entry activity called by a system user. In our example system, such entry activity is the *TripPlanner.getTrip()* activity.

Further requirements 6 – 9 are concerned with the transformation of different types of activity nodes. Requirement 6 states that an external call action is to be transformed into a process fragment consisting of two events for calling (event $call_calledActivity$) and awaiting the reply (event $awaitreply_calledActivity$) of the called activity (named $calledActivity$). The call connection process $CALLERID_CALLS_CALLED_ACTIVITY$ for this external call action also has to be created and added to the combining process M_CALLS . The events of this process have to be added to the synchronization set X_M . The structure of such call connection process has been previously considered in Figure 3.13 and, thus, its explanation is not repeated here. The rest of this requirement states which CSP channels should be created for an external call action to store the parameter values of the events $call_calledActivity$ and $awaitreply_calledActivity$. In this case, there is only one value – $callerID$ – the ID of the transformed call action itself. Finally, its ID is also added to the set $CALLED_ACTIVITY_CALLERS$ of callers of the activity $calledActivity$.

Since requirements 8 and 9 for the transformation of acquire and release actions are very similar to requirement 6 in their CSP constructs and only differ in the names of processes, events, channels, and sets, we do not consider them in detail here. Therefore, the last requirement for an activity node to be considered is the one for an internal action (requirement 7). Compared to the requirements for other node types, this requirement is quite simple. It states which event prefix process expression and which channel should be created for an internal action – a prefix with an event $action.x$ and a channel $action$. The created event, like the one for a final activity node, does not accept any new input, but transmits the parameter value previously obtained by $?x$. For this reason, the values set of the channel $action$ is the set of IDs of all callers of its containing activity $A - A_CALLERS$.

The last group of requirements to be considered are requirements 10 – 13 for the transformation of different control-flow structures of an activity. Requirement 10 states that a sequence of activity blocks B_1, B_2 is to be transformed into a sequence of process expressions PE_1 and PE_2 related to the blocks with the same number. As in requirement

3, the relations between these place-holders indicate where the result of the transformation of the activity parts represented by the blocks should be placed in the resulting CSP specification. Requirements 11 – 13 use blocks and process expressions for the same purpose.

Requirement 11 states that a decision/merge structure should be transformed into an if-then-else CSP process expression with the same condition and sub-expressions (PE_1 , PE_2), related to the activity blocks (B_1 , B_2) of the structure. Requirement 12 states that a fork/join structure should be transformed into a parallel composition of sub-expressions (PE_1 , PE_2), related to the activity blocks (B_1 , B_2) of the structure.

Finally, requirement 13 states that a loop structure should be transformed into a parametrized recursive process $L(x)$ that simulates loop behavior, takes condition x as a parameter, and executes some process expression PE (related to block B) as long as this condition is true. Additionally, a reference to this process with a parameter c (the actual condition of the transformed loop) is also created for the integration into the main CSP process definition. The process $L(x)$ with its specific structure is used here since CSP language does not provide a dedicated construct for loops.

The last requirement from Table 3.1 to be considered, is on the transformation of a resource (requirement 14). It states that a parametrized process $RES(x)$ should be created for each resource with the same name, and its instance $RES(c)$ with the capacity c should be added via a parallel composition to the $M_PROCESSES$ process definition. As mentioned before, the $M_PROCESSES$ process combines all of the main (no call connection) processes of the transformed model. The structure of the $RES(x)$ process is very similar to the one of the parametrized resource managing process TP described in Section 3.2. Thus, to avoid the repetition, we only state their relation as follows: the process RES can be seen as an extension of the process TP that uses parametrized events.

Furthermore, requirement 14 describes two channels *acquireRes* and *releaseRes*, and two value sets *ACQUIRE_RES_CALLERS* and *RELEASE_RES_CALLERS* that should be added to the resulting CSP specification. The two channels contain all possible values of the parameters of the events *acquireRes* and *releaseRes*, and the values sets contain IDs of all the callers of the resource acquisition and release actions, respectively. Initially, the value sets are empty and are filled through the transformation of acquire and release action nodes of activities.

Now, after all of the requirements have been discussed, to complete the picture, we demonstrate a fragment of the expected transformation result for the *TripPlanner.getTrip()* activity of our layered system model. It is shown in Figure 3.15³. There, the definition of the process *GETTRIP* is composed from the transformation results for the individual activity nodes, based on the requirements from Table 3.1. To make

³To keep the CSP specification concise, we combine union operations on set $X_APPLICATION$ and replace parallel compositions of $APPLICATION_CALLS$ process by a single one with multiple arguments.

```

GETTRIP = invoke_getTrip?x → (
  call_getMap.CallAction1ID → awaitreply_getMap.CallAction1ID → SKIP
  || if from = to then
    call_getLocalTransport.CallAction2ID →
    awaitreply_getLocalTransport.CallAction2ID → SKIP
  else
    call_getRoute.CallAction3ID →
    awaitreply_getRoute.CallAction3ID → SKIP
  ) § reply_getTrip.x → GETTRIP

channel invoke_getTrip,reply_getTrip : GETTRIP_CALLERS
APPLICATION_PROCESSES = (APPLICATION_PROCESSES || GETTRIP)
GETTRIP_CALLERS = GETTRIP_CALLERS ∪ {externalID}

CALLACTION1ID_CALLS_GETMAP =
  call_getMap.CallAction1ID → invoke_getMap.CallAction1ID →
  reply_getMap.CallAction1ID → awaitreply_getMap.CallAction1ID →
CALLACTION1ID_CALLS_GETMAP

APPLICATION_CALLS = APPLICATION_CALLS || CALLACTION1ID_CALLS_GETMAP || ...

X_APPLICATION = X_APPLICATION ∪ {call_getMap.CallAction1ID,
  invoke_getMap.CallAction1ID,reply_getMap.CallAction1ID,...}
channel call_getMap,awaitreply_getMap : {CallAction1ID}
GETMAP_CALLERS = GETMAP_CALLERS ∪ {CallAction1ID}

```

Figure 3.15: Fragment of the transformation of the *TripPlanner.getTrip()* activity into CSP

the complete process expression conform to the CSP syntax (see Section 3.2), we have added *SKIP* processes (in italic) to the event prefixes, obtained from the transformation of call actions. Other processes, channels, event and value sets in Figure 3.15 are also generated according to Table 3.1. Since the requirements were described in detail and this resulting CSP specification shows their example result, we do not give any further explanation here.

The example multi-layer system from Section 3.1 and the ADR-to-CSP transformation requirements, that we have introduced above, trigger the following central questions:

1. How can the interactions between the models of the layers be modeled in a flexible way that supports their exchange?
2. How can a model transformation, that fulfills the requirements and is of high quality, be developed?
3. How to allow non-invasive re-use of existing model transformations from (2) and of their results on models of individual layers, when the system is re-configured (layers added, re-connected, and/or removed)?

To answer these questions we next (Chapter 4) look into the systematic development and re-use of model transformations in MDSD in details, using our example. Then, we introduce and discuss in detail the approaches that we propose to achieve the following:

- Improve the existing model transformation development process and the quality of the resulting transformations (in Chapter 5),
- Allow flexible (re-)configuration and transformation of multi-layer systems, including the re-use of the developed model transformations and their results (in Chapter 6).

Later, in Chapter 7, we combine the above two approaches and discuss the results.

Chapter 4

Systematic Development of Model Transformations

Up to now, we have discussed the key concepts of MDSD, like models and model transformations (Chapter 2), and have introduced our transformation case study (Chapter 3). Now, in Section 4.1, we look at various aspects of the development of model transformations, including different transformation types, their design, and implementation. There, we abstract from particular purposes of the source and target models, like PIM and PSM in MDSD, to make our description more universal.

Afterward, in Section 4.2, we discuss the following questions related to the quality assurance for model transformations: several quality properties that are important for a model transformation; when can they be guaranteed; and which design and analysis approaches exist for this purpose. We evaluate the applicability of these approaches to our transformation case study and, where applicable, their results for it.

Finally, since the development of quality model transformations is often very complex and costly, techniques for the re-use of existing model transformations are very important. In Section 4.3, we look at different existing transformation re-use techniques and at their applicability to our transformation example.

4.1 Development of Transformations

As we have already stated in Section 2.3, a definition of a model transformation maps individual elements (objects) and structures, containing multiple related elements (patterns), within the source language onto those in the target language. The purpose of these mappings is to represent the relations between the source and target the languages, that are specific for this model transformation, based on their elements. The mappings are usually defined on the level of meta-models and in abstract syntax, to keep them re-usable when the concrete syntax of the languages changes.

In MDSD model transformations are used at different steps and for different purposes (see Chapter 2). To better understand their role, we first give a brief classification of different model transformations in MDSD, in Section 4.1.1, before discussing the most

common approaches to their development. Then, we mainly focus on the design and implementation phases in the development of model transformations, in Sections 4.1.2 and 4.1.3, respectively. This is due to the fact that other phases, like requirement engineering or testing for model transformations are out of the scope of this work. We demonstrate application of the discussed design and implementation approaches on our running transformation case study.

4.1.1 Transformation Types

The following classification of model transformations in the MDSO context, adopted from [GPR06], gives us a basic overview of different model transformation types. Its key classification characteristics are the following: the source model structure (modeled concepts) and its abstraction level. For more fine-grained and specific classifications of model transformations in MDSO, we refer to the following surveys [CH06, MG06, DREP12] and to the intent catalog from [LAD⁺14].

Figure 4.1 shows different types of model transformations (depicted as arrows between the models) that are established based on these classification characteristics. There are the following main types in this classification: *horizontal*, *vertical*, and *oblique (diagonal)* transformations.

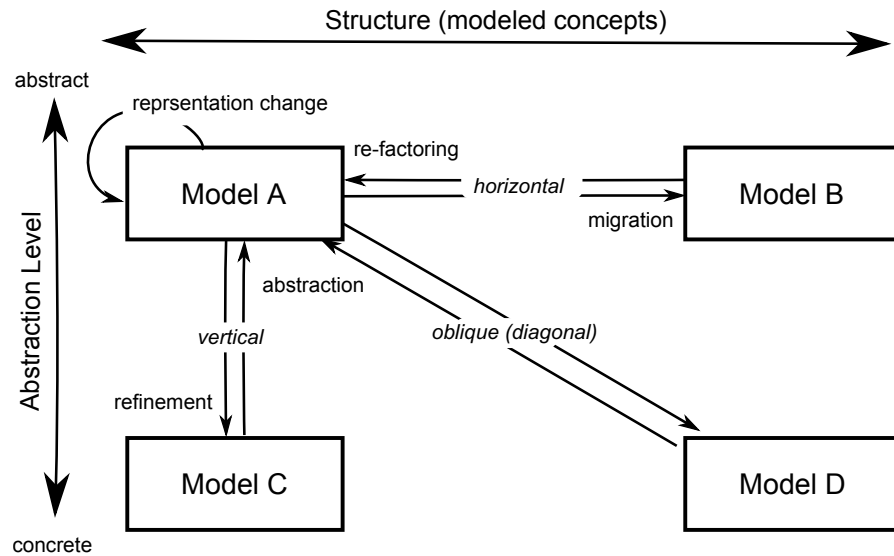


Figure 4.1: Classification of model transformations based on [GPR06]

Horizontal transformations keep the abstraction level of a source model the same but change its structure. Examples of horizontal transformations are *migration* and *re-factoring*. The first sub-type – migration – is mainly used when the platform of the system or of its part has to be changed without affecting the functionality [FBB⁺07, FHRW13]. Model re-factoring is used to improve the readability and maintainability

of the software models by changing their structure but keeping the functionality they describe, like in [MTM08].

Vertical transformations, on the other hand, change the level of abstraction of a source model but keep its structure. Examples of vertical transformations are *abstraction* and *refinement*, that have a more or less abstract target model, respectively. The PIM-to-PSM transformation type, commonly used in MDSD (see Chapter 2), can fall into this sub-category, if the structure of the PIM model does not change as in [KB10]. Abstraction is also used in MDSD, when only certain information needs to be extracted from a source model, like the interaction probabilities in [BWW11].

Oblique (diagonal) transformations change both characteristics i. e., the abstraction level and the structure of a source model. The PIM-to-PSM transformation type from the MDSD context, that we have already mentioned, can also fall into this sub-category, if the structure of the PIM model also changes.

Another type of model transformation also present in this classification, that we have not yet named, is the *representation change*. In this model transformation type, both the abstraction level and the structure of a source model stay the same, which is represented by the self-arrow of *Model A* in Figure 4.1. An example of such model transformation is a change from a graphical to a textual representation of a model [LPH11, BWB12], that is often called *model-to-text* transformation. Such model transformations are often used in MDSD to facilitate tool interoperability [BCC⁺10].

When a transformation changes the representation language of a model and not only its concrete syntax representation, like in [Küs06, KB10], it is referred to as an *exogenous (out-place)* transformation or also as a *translation*. In this case, the structure of the transformed source model and, in some cases, its abstraction level also changes. In terms of the above classification, the translation is then seen as a horizontal/diagonal transformation. When a transformation does not change the representation language, it is labeled as an *endogenous (in-place)* transformation. This transformation characteristic is orthogonal to those shown in Figure 4.1 and can be combined with them.

As we can see, not each model transformation fits into exactly one of the above categories. Some of them combine characteristics of two or more types. Our case study transformation from Chapter 3, when classified with respect to the above categories, is a *horizontal translation*. It change the structure of a source ADR model by translating activities into processes and other CSP constructs. Plus, it changes its representation language (and not only its concrete syntax) by creating a CSP specification from it.

Next, we look at the way a model transformation is developed and demonstrate it on our ADR-to-CSP translation example. We concentrate on the design and implementation tasks of the development process, and consider the most common techniques.

4.1.2 Transformation Design

Each transformation consists of a set of mappings that define which elements and structures of the source and the target languages are mapped onto each other. The design of these mappings often begins at a *high-level*, where the designer states which elements and structures are to be mapped by which mappings. Figure 4.2 shows some selected high-level mappings for our example transformation from Section 3.3, using hexagons to depict mappings and rectangles to depict the constructs of the languages. Here, an ADR model is mapped onto different CSP processes. Furthermore, an activity is mapped onto a process, and a sequence of activity blocks onto a sequence of processes in CSP. These mappings are consistent with the requirements formulated in Table 3.1. Thereby, one requirement can be expressed through several mappings and vice versa.

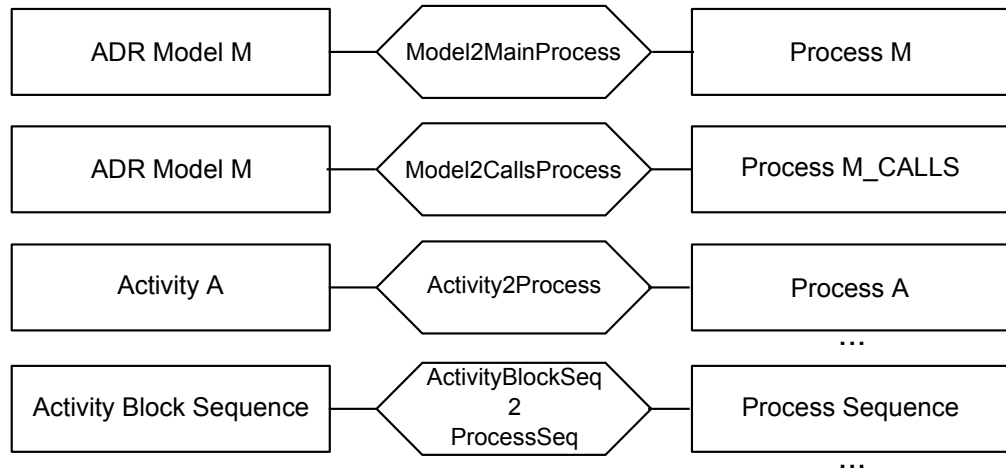


Figure 4.2: Selected high-level mappings for the ADR-to-CSP transformation (notation from [GPR06])

At this point, it is not yet specified how the mapped elements and structures, and the mappings themselves, will be described. Yet, we already know which mappings involve which language elements and structures. Later, the high-level mappings are refined into so-called *detailed mappings*, where the mapped language concepts are replaced by their detailed description. Sometimes, however, the described high-level design step is skipped and the detailed mappings are created, and sometimes even implemented, directly.

To create a detailed mapping from a high-level one, we need to know how the mapped language concepts are described. In MDSD, source and target languages are defined using meta-models (see Section 2.2). These meta-models provide alphabets (allowed object and link types – classes and associations) for the description of the concepts of the languages that are mapped. Figure 4.3 shows a sketch of such meta-model based transformation definition approach. It generalizes the idea shown in Figure 2.10 for the PIM-to-PSM transformations to any source and target models and extends it with further (meta-)meta-models. Thereby, a transformation definition is conform to its own

transformation meta-model and, all meta-models have the same meta-meta-model describing them, like MOF in MDSD.

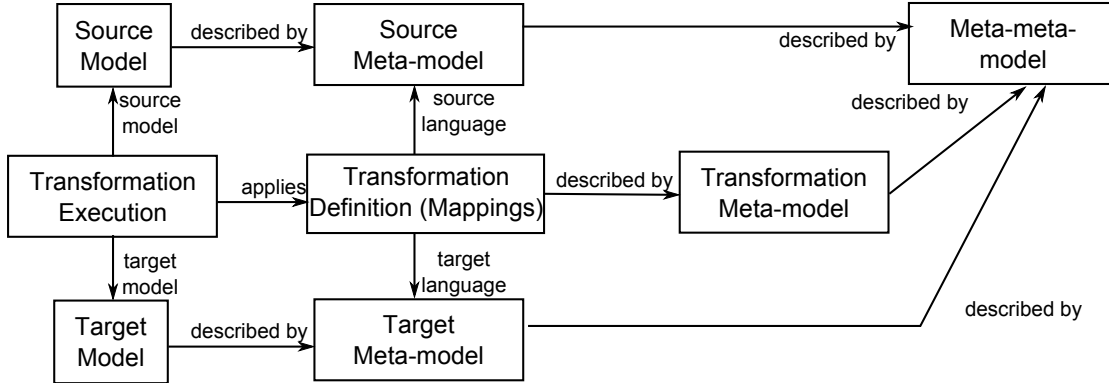


Figure 4.3: Sketch of the meta-model based transformation definition approach

In this meta-model based approach to the transformation definition, each mapping relates model *patterns* from the source and the target languages. A model *pattern* contains instances of classes (objects) and of associations (links) between them, with the allowed classes and associations defined in the source and target meta-models.

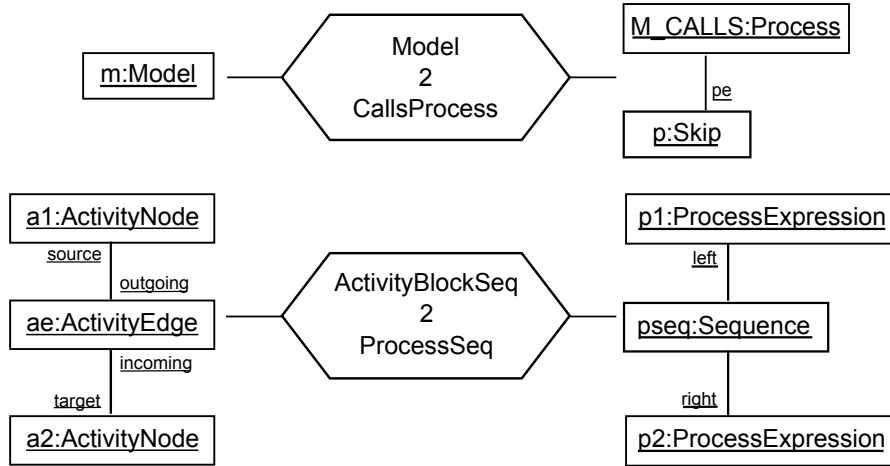


Figure 4.4: Selected detailed meta-model based transformation mappings for the ADR-to-CSP transformation

Figure 4.4 shows two detailed mappings for our example ADR-to-CSP transformation. Here, the high-level constructs from the involved languages are now described by meta-model based patterns. In the first mapping *Model2CallsProcess*, a model object with the name *m* is mapped onto a process object with the name *M_CALLS*, described by a *SKIP* process expression. In the second mapping *ActivityBlockSeq2ProcessSeq*, a sequence of two activity nodes linked through an edge is mapped onto a sequence of process expressions. Note that the second mapping involves instances of abstract classes

ActivityNode and *ProcessExpression*. It indicates that each sub-type of an activity node in a sequence is to be mapped in such a way, and leaves the details of the concrete CSP process expressions for different node sub-types out. These details are then specified in other mappings for the concrete sub-classes of the *ActivityNode* class..

In some cases, the source meta-model also contains structuring classes, introduced in the end of Section 2.2 (see Figure 2.8). Our source ADR meta-model in Figure 3.7 does not contain such classes, but we show a way to add them in Figure 4.5. The structuring classes that we add – *ActivityGroup*, *StructuredActivityNode*, and *SequenceNode* classes – are a sub-set of classes shown in Figure 2.8. They exemplify the same node and edge grouping mechanism for class *SequenceNode* in a simplified form. We consider this extension of our ADR meta-model to cover the case of meta-models with structuring classes in our transformation design and implementation discussion.

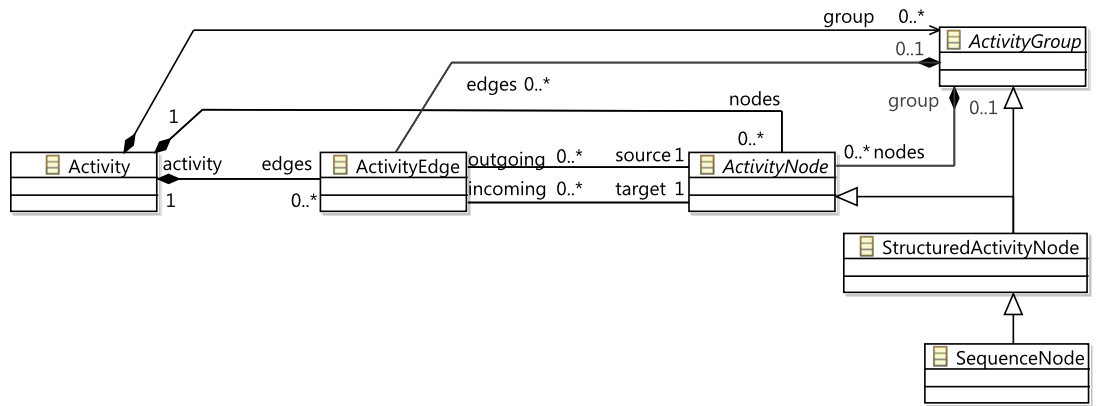


Figure 4.5: Structuring classes for the ADR meta-model from Figure 3.7 in Ecore

When, a meta-model contains the discussed structuring classes, the transformation designer has to define additional mapping(s) for them. For instance, for our class *SequenceNode* from Figure 4.5, we define an additional mapping *SequenceNode2ProcessSequence*. It is shown in Figure 4.6 and is very similar to the *ActivityBlockSeq2ProcessSeq* mapping, we have considered before. The resulting transformation design then has two separate mappings for the sequence of nodes inside (*SequenceNode2ProcessSequence* mapping) and outside (*ActivityBlockSeq2ProcessSeq* mapping) of the structuring sequence node, and further mappings for all concrete activity node types. Later, we show that such structuring classes also do not necessarily simplify the transformation implementation.

Next, the transformation mappings detailed above are used as a basis for the transformation implementation. There, the focus is shifted from the definition of the source and target language constructs to the definition of the mappings themselves. Thereby, the relation between the mappings and their implementations is not always one-to-one.

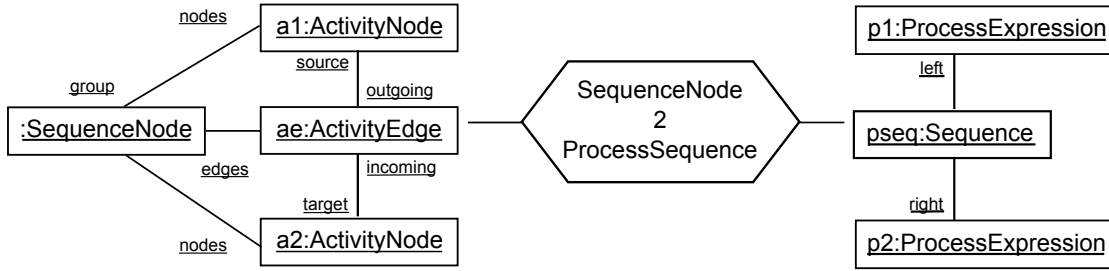


Figure 4.6: Detailed meta-model based mapping for the structuring class *SequenceNode*

A transformation implementation can be defined in different languages and by using various strategies to complete the mappings and make them executable. Next, we consider two languages, widely used in MDSD for the implementation of model transformations, and how they can be employed to implement our ADR-to-CSP transformation case study. Later, in Chapter 5, we then consider an alternative, not meta-model based approach to the transformation design and implementation.

4.1.3 Transformation Implementation

In MDSD there are different ways to implement a model transformation. One can employ a widely used general purpose programming language, like Java, or a specialized transformation language like ATL or TGGs. The second strategy is preferred in MDSD, since these languages are developed specifically for the transformation implementation and include some advanced specialized (domain-specific) features. Here, we give a short overview of both alternatives, before considering the second one in more detail.

In the case, when a general purpose programming language is used, the detailed model transformation mappings are translated into sets of instructions over an in-memory representation (an object graph) of an input source model. These instruction sets are often grouped by methods. The calls of these methods, defined in some order, then constitute the transformation implementation. To access the source and produce the target model, the created transformation program uses a representation of meta-models specific for the chosen implementation language. For instance, for a Java based transformation implementation, one uses Java based specifications code for meta-models, that can be created using EMF [EMF]. This choice eliminates the need to learn specialized transformation languages, yet, it decrease understandability and maintainability of the resulting model transformation. It also contradicts with one of the main ideas of MDSD, that promotes the use of specialized domain-specific languages and tools for specific tasks [GPR06].

In the case, when a specialized transformation language is used, the detailed model transformation mappings are translated into *transformation rules*. A transformation rule consists of two parts – *selection* and *instruction* [GPR06]. The selection part contains a set of constraints, that the elements of the transformed source model and, possibly,

of the created fragment of the target model, have to fulfill in order to be transformed. Such constraints are usually expressed by model patterns and further conditions in e.g., OCL. The instruction part of the rule contains a set of instructions/operations that are to be executed for each set of elements, that fulfills the selection conditions. The form and nature of such instructions highly depends on the selected model transformation implementation language.

From the structural point of view, a transformation rule can have declarative (relational) and/or imperative parts. The declarative part of a rule has a close resemblance with a detailed mapping – it contains source and target model patterns, that may include further conditions and explicit or implicit correspondence links between the patterns elements. Thereby, the patterns within such rule are not necessarily exactly the same as the ones in the detailed mapping(s), that this rule implements.

The imperative part of a rule usually describes the rest of the transformation mapping. It describes everything that is difficult, if not impossible, to describe using the declarative part of a rule. For this purpose, the imperative part often contains the following constructs: collection iteration, call of rules and functions, and so on.

Different transformation implementation languages provide different scopes of constructs and instructions for the definition of transformation rules. Languages like QVTd [QVTc] and TGGs [Sch94] only provide means to define declarative rules. Whereas, languages like QVTo [QVTd] and ATL [ATL] provide means to define imperative constructs in rules. In fact, ATL provides support for both types of implementation constructs.

In general, more expressive model transformation languages allow to implement a wider scope of transformations. Even if languages have the same expressiveness, like Turing complete ATL [Tis14] and TGG [BDK⁺12] languages, implementations of the same transformation in them can have different complexity, due to the language constructs they provide or do not provide. Both the model transformation complexity and the use of imperative constructs in it can hinder its readability and reduce its analysis possibilities. Next, we consider the two mentioned transformation implementation languages in more detail, starting with ATL. We discuss their use, advantages, and shortcomings on our ADR-to-CSP transformation case study from Section 3.3.

Implementation of the ADR-to-CSP Transformation Example with ATL The hybrid model transformation implementation language called *ATL* (*ATLAS Transformation Language*) was first extensively described in [JABK08]. It is a domain-specific language inspired by the requirements on QVT standard specified by the OMG in [QVT02], that builds upon the OCL formalism. It is designed specifically to implement meta-model based transformations in the context of MDSD.

ATL contains a mixture of declarative and imperative constructs. Transformations described with it are unidirectional, they operate on read-only source models, and produce write-only target models. A source model may be navigated in ATL transformation

rules, but changes to it are not allowed and the resulting target model, on the other hand, cannot be directly navigated in the transformation rules. Instead, for this purpose, ATL uses a special *resolution mechanism*, discussed later.

A transformation implementation in ATL is structured into *modules* with a mandatory *header*, an *import* section, a number of *helpers* (functions), and transformation *rules*. Here, we focus on our example implementation of the ADR-to-CSP transformation from Section 3.3 in ATL and, thus, only discuss the ATL concepts that are used in the transformation fragments that we demonstrate.

```

1 module adr2csp;
2 create csp : CSP from adr : ADR;
3
4 rule Model2ProcessesAndEventSet{
5 from in_m : ADR!Model
6 to out_m : CSP!Model(
7     name <- in_m.name, processes <- in_m.activities,
8     processes <- Set{out_main_proc,out_calls_proc,...},
9     eSets <- out_x_m, ...
10 ) ,
11 -- create further processes and event sets...}
12
13 rule Activity2ProcessesChannelsValueSet{
14 from in_a : ADR!Activity
15 to out_a_proc : CSP!Process(
16     name <- in_a.name,
17     pe <- thisModule.resolveTemp(in_a.nodes->select( n |
18         n.ocIsTypeOf(ADR!InitialNode))->first(), 'out_seq')
19 ) ,
20 -- create further processes, channels, and value sets...}

```

Listing 4.1: ATL fragment for high-level mappings of requirements 1 – 4

Listing 4.1 shows the first fragment of our ATL based implementation of the ADR-to-CSP transformation. It contains a header with the transformation name (*adr2csp*) and its source and target models as variables (*adr* and *csp*) typed by the respective meta-models (*ADR* and *CSP*). The used *ADR* and *CSP* meta-models are from Sections 3.1 and 3.2, respectively. Furthermore, this ATL fragment contains the following two transformation rules (shown partially): *Model2ProcessesAndEventSet* and *Activity2ProcessesChannelsValueSet*. They implement requirement(s) 1 and 2 – 4 from Table 3.1, respectively.

Requirement 1 has been described by several high-level mappings, including *Model2MainProcess* and *Model2CallsProcess*, shown in Figure 4.2. In ATL, however, we use one rule *Model2ProcessesAndEventSet* to implement them because ATL does not allow a module to have more than one rule with the same elements in the selection part (*from*). Otherwise, more than one rule may be applicable at the same time, causing non-deterministic behavior of the implementation.

Both rules in Listing 4.1 are declarative *matched rules* with the selection (keyword *from*) and instruction (keyword *to*) model patterns. A selection pattern consists of a combination of instances of the source meta-model classes, OCL collection types, and often an OCL *guard* condition. An instruction pattern consists of a set of instances of the target meta-model classes to be created, and of *bindings*. A binding connects (symbol ' $\leftarrow$ ') a feature (an attribute or an association end) of an element of the target pattern with its *initialization expression*. During the execution of an ATL rule, the engine finds a set of matches for the selection pattern in the source model and, for each of these matches, creates the instruction pattern elements of the rule in the target model and initializes their features according to the bindings.

The selection part of the *Model2ProcessesAndEventSet* ATL rule contains a source ADR model *in_m*. The instruction pattern of the rule described which CSP elements should be created for it based on the requirement as follows: a CSP model *out_m* with the same name and features, like *processes* and event sets (*eSets*). Here, we highlight the binding of, at a first glance, all *activities* of *in_m* to the *processes* feature of *out_m*. However, at this point, ATL uses its *resolution mechanism* to get the transformation results (processes) for individual *activities* and assign them to the *processes* of *out_m*. The individual processes are created by the rule *Activity2ProcessesChannelsValueSet*.

In the ATL rule *Activity2ProcessesChannelsValueSet*, an activity *in_a* is transformed into a set of processes, channels, and value sets. The first process in the instruction pattern – *out_a_proc* – is the equally named process, created for *in_a* to fulfill requirements 3 and 4, and used in the *Model2ProcessesAndEventSet* rule. The highlight here is the binding of the process expression *pe* of *out_a_proc*. It is done using the explicit ATL resolution operation *resolveTemp*(*var*, *trg_pattern*) available in a module. Its parameters are as follows: *var* – the transformed source model elements; and *trg_pattern* – the name of the instruction pattern element, that *var* is transformed into.

Here, *resolveTemp* is called with an OCL query and '*out_seq*', as the first and the second parameter, respectively. The query obtains the initial node of *in_a* by selecting the first¹ node *n* of the respective type (*oclIsTypeOf* operation) from its *nodes*. The selected initial node is used to find/resolve its transformation result, created by another ATL rule *InitialNode2SeqPrefix*, shown in Listing 4.2, and use its part created as the instruction pattern element '*out_seq*' as the expression *pe* of *out_a_proc* process.

This way, rule *InitialNode2SeqPrefix* implements a part of requirements 3 and 4 concerned with the events of the process created for each activity. It creates an event prefix *pref* with the invocation event *ev*, with respective parameter features, and a *SKIP* process expression (to create a correct process expression) from an initial node *in_ac*. This prefix already describes the required transformation result. The sequence *out_seq*, also created in this rule, is a part of the mechanism we use in this implementation to *connect* the results of the individual ATL rules. We discuss later how the second/right expression

¹In a well-formed ADR model, there is only one initial node for each activity.

of this sequence is defined by other rules. The reply event prefix *out_seq*² is created by a similar rule for a final node. There, no new sequence is required.

```

1 rule InitialNode2SeqPrefix{
2 from in_ac : ADR!InitialNode
3 to out_seq : CSP!Sequence(left <- pref),
4   pref : CSP!Prefix(event <- ev, pe <- CSP!Skip.newInstance()),
5   ev : CSP!Event(
6     name <- 'invoke_' + in_ac.activity.name, direction <- #in...
7   ) }

```

Listing 4.2: ATL rule for high-level mapping *InitialNode2Prefix* (req. 3 and 4)

ATL rules that implement high-level mappings of other simple activity node types, like an internal or a call action (requirements 6 – 9), also look similar to the *InitialNode2SeqPrefix* rule with slight structural changes to fulfill the respective requirements. Thus, we do not consider them here to avoid the repetition. However, for high-level mappings of nodes that are part of language structures (requirements 11 – 13), like a decision/merge block, we consider an example rule *Decision2SeqCondition*, in Listing 4.3, to provide a basis for comparing the rules for such structural, and the simple, types.

```

1 rule Decision2SeqCondition{
2 from in_d : ADR!Decision
3 to out_seq : CSP!Sequence(left <- cond),
4   cond : CSP!Condition (
5     c <- in_d.condition,
6     left <- thisModule.resolveTemp(in_d.outgoing->select( e |
7       e.guard = 'y')->first().target, 'out_seq' ),
8     right <- thisModule.resolveTemp(in_d.outgoing->select( e |
9       e.guard = 'n')->first().target, 'out_seq' )
10  ) }

```

Listing 4.3: ATL rule for high-level mapping *DecisionMergeBlock2Condition* (req. 11)

The ATL rule *Decision2SeqCondition* transforms a decision node *in_d* into a sequence *out_seq*, whose left/first sub-expression is a conditional composition *cond* instead of a prefix, like in *InitialNode2SeqPrefix*. This process composition *cond* has the same condition as *in_d*, and its *left* (holds) and *right* (does not hold) sub-expressions are assigned to the transformation results (*out_seq* part) of the respective direct successor nodes of *in_d*. Both transformation results are obtained using the explicit resolution mechanism of ATL. Note that these resolution calls rely on well-formedness of source models, since they only assume one outgoing edge from *in_d* for each case.

We can see that the *Decision2SeqCondition* rule does not consider any merge node. Our ATL implementation of the ADR-to-CSP transformation also does not contain a separate rule for merge nodes, since, based on the requirements, we only need merge nodes to identify when a decision/merge block ends and do not map them.

²The prefix is named *out_seq* here to allow a uniform use of the explicit resolution mechanism of ATL.

The information a merge node provides is used by a separate ATL rule *ActivityBlockSeq_DecisionFirst2ProcessSeq*, shown on Listing 4.4, to integrate the result of the decision node transformation by *Decision2SeqCondition* into the process expression generated for its activity. This rule implements the *ActivityBlockSeq2ProcessSeq* mapping (see Figure 4.2) for the case when the first block in the sequence is a decision/merge. It transforms two activity nodes *in_an1* and *in_an2* with an additional OCL condition (in brackets), that states that *in_an1* should be a decision (*oclIsTypeOf(ADR!Decision)*) and *in_an2* should be its corresponding merge (*in_an2 = thisModule.findCorrespondingMergeNode(in_an1)*). This rule, thus, also relies on the well-formedness of activities.

```

1 rule ActivityBlockSeq_DecisionFirst2ProcessSeq{
2   from in_an1 : ADR!ActivityNode, in_an2 : ADR!ActivityNode
3     (in_an1.oclIsTypeOf(ADR!Decision) and
4      in_an2 = thisModule.findCorrespondingMergeNode(in_an1))
5   do{
6     thisModule.resolveTemp(in_an1,'out_seq').right <- thisModule.
7       resolveTemp(in_an2.outgoing->first().target,'out_seq');
8   }

```

Listing 4.4: ATL rule for high-level mapping *ActivityBlockSeq2ProcessSeq* (req. 10) with a decision/merge first block

Since, like a UML meta-model, our ADR meta-model does not contain a structuring class for a decision/merge block, but separate classes *Decision* and *Merge*, we require the imperative constructs of ATL to search for such related nodes. In our implementation, we define a *helper* function *findCorrespondingMergeNode*, on Listing 4.5, to start recursive search for the merge node, that corresponds to the decision node parameter *d*.

```

1 helper def:findCorrespondingMergeNode(d:ADR!Decision):ADR!Merge =
2   thisModule.recursiveSearchOfMerge(d.outgoing->first().target,0);
3
4 helper def:recursiveSearchOfMerge(curr:ADR!ActivityNode,
5                                   nestedDecision:Integer):ADR!Merge =
6   if curr.oclIsTypeOf(ADR!Merge) then
7     if nestedDecision > 0 then
8       thisModule.recursiveSearchOfMerge(
9         curr.outgoing->first().target,nestedDecision-1)
10    else curr endif
11  else
12    if curr.oclIsTypeOf(ADR!Decision) then
13      thisModule.recursiveSearchOfMerge(
14        curr.outgoing->first().target,nestedDecision+1)
15    else
16      thisModule.recursiveSearchOfMerge(
17        curr.outgoing->first().target,nestedDecision)
18    endif endif;

```

Listing 4.5: ATL helpers for *ActivityBlockSeq_DecisionFirst2ProcessSeq* rule

An ATL helper is a mechanism provided to define a model *decoration* or an *operation/function* (with parameters) in the context of a module or a source meta-model class. In our example, on Listing 4.5, there are two following helpers defined in the context of the module: the mentioned *findCorrespondingMergeNode* that starts the recursive search for the merge; and the *recursiveSearchOfMerge* helper that implements this search.

The helper *findCorrespondingMergeNode* calls the *recursiveSearchOfMerge* helper, using the first activity node descending from the decision node *d* as a start/current node (*curr*), and sets the number of decision/merge blocks found in between (*nestedDecision*) to 0. The called helper checks the type of the current node *curr* and goes to the next node (via a recursive call), if it is not a merge node. If the *curr* node is not a decision nor a merge, the parameter *nestedDecision* stays the same. If the *curr* node is a decision, it is increased by one, and if the *curr* node is a merge, it is decreased by one. This way, with the help of this variable, the helper counts intermediate decision/merge blocks and returns the merge node that is on the same nesting level (0) as the decision *d*. The caller helper delivers the found merge to *ActivityBlockSeq_DecisionFirst2ProcessSeq* rule.

The *ActivityBlockSeq_DecisionFirst2ProcessSeq* rule then implements a part of the *connection mechanism*, that we use to link the transformation results for the individual activity nodes and structures/blocks. Unlike in the previous rules, here the instruction pattern is put into the *do* section³. The *do* section of a matched rule is an optional sequence of imperative statements (loops, iterations, assignments/bindings, etc.) that can be used alone or in a combination with a declarative instruction pattern. It is executed once the initialization of the target model elements generated by the rule has been completed and can, in particular, be used to initialize/modify some features. An ATL rule with such a *do* section is no longer declarative.

The *do* sections of *ActivityBlockSeq_DecisionFirst2ProcessSeq* rule contains one binding, that assigns the right/second sub-expression of the sequence, created from the node *in_an1*, to the sequence, created from the node that follows the merge node *in_an2*. Both sequences are obtained through the explicit resolution and, again, the second resolution relies on the well-formedness of the transformed source models by assuming, that there is only one outgoing edge for the merge.

The considered node-to-sequence and connection rules from Listing 4.3 and 4.4 respectively, are specifically defined for decision/merge blocks. To transform fork/join and loop blocks and connect these transformation results, we also defined node-to-sequence and connection rules, similar to those considered above. Furthermore, we defined a helper similar to the *findCorrespondingMergeNode* helper, to find the corresponding join node for a fork node. For a loop node such search helper is not needed, since it both starts and ends the containing structure. Due to the similar structure of these rules and of the helper, we do not consider them here. Instead, we consider the 'default' connection rule

³There exist other implementation alternatives to connect the individual results in ATL, including the use of called rules and iteration, but all of them require imperative constructs for our example.

in our ATL implementation, shown in Listing 4.6, that covers connection of the transformation results for the remaining combinations of activity block/node sequences.

```

1 rule ActivityBlockSequence2ProcessSeq{
2   from in_an1 : ADR!ActivityNode, in_an2 : ADR!ActivityNode (
3     (not in_an1.occlIsTypeOf(ADR!Decision)) and
4     (not in_an1.occlIsTypeOf(ADR!Merge)) and (-- not fork/join/
        loop)
5     (in_an1.outgoing->collect(e|e.target)->includes(in_an2)))
6   do{
7     if(in_an2.occlIsTypeOf(ADR!Merge) -- or join/loop){
8       thisModule.resolveTemp(in_an1,'out_seq').right <-
9         CSP!Skip.newInstance();
10    }else{
11      thisModule.resolveTemp(in_an1,'out_seq').right <-
12        thisModule.resolveTemp(in_an2,'out_seq');
13    }
14  }

```

Listing 4.6: Default ATL rule for high-level mapping *ActivityBlockSeq2ProcessSeq* (req. 10)

The ATL rule, called *ActivityBlockSequence2ProcessSeq*, transforms two activity nodes *in_an1* and *in_an2*, like the rule *ActivityBlockSeq_DecisionFirst2ProcessSeq*. However, it requires *in_an1* to be a direct predecessor of *in_an2* and excludes decision, merge, and other structure-forming nodes. It also only has an imperative instruction section. This *do* section assigns the right/second sub-expression of the sequence created from *in_an1* either to the sequence created from *in_an2*, or to a new *SKIP* process. The second alternative is only used if *in_an2* is a merge, a join, or a loop node that finishes the structure.

To summarize, the presented ATL implementation of our ADR-to-CSP transformation example consists of the rules to transform the following: an ADR model (rule *Model2ProcessesAndEventSet*); its resources and activities (rule *Activity2ProcessesChannelsValueSe*); initial and final nodes (rule *InitialNode2SeqPrefix*); various simple intermediate activity nodes, like internal actions; and various block structures - decision/merge (rule *Decision2SeqCondition*), fork/join, and loop blocks. Furthermore, it contains rules to connect the individual results of the above rules (e.g., *ActivityBlockSequence2ProcessSeq*, *ActivityBlockSeq_DecisionFirst2ProcessSeq*) and the respective helpers.

During the implementation process, we had to address the following questions:

1. How to implement high-level mappings for one or more ADR elements by ATL rules?
2. How to connect the transformation results of the individual ATL rules?
3. How to incorporate the difference in connecting the result for a simple node, like an internal action, and for a block structure?

For our implementation, we aimed to use the declarative part of ATL, like matched rules, as much as possible. To answer the first question, we had to make design decisions about how to split and/or combine high-level mappings to be implemented (sometimes partially) by one matched ATL rule. Since ATL does not allow multiple matched rules with the same selection pattern, we grouped high-level mappings for one source element (e.g., ADR model) and implemented them in one rule (e.g., *Model2ProcessesAndEventSet* rule). Up until this point we did not have to use any imperative constructs of ATL.

To answer the second question, we employed the resolution mechanism of ATL. In some cases, we use it directly in the declarative instruction patterns of the rules (e.g., rule *Activity2ProcessesChannelsValueSet*). In other cases, however, we implement separate rules to connect the transformation results (e.g., rule *ActivityBlockSeq_DecisionFirst2ProcessSeq*). Such separate rules allow a more clear distinction between the different connection cases, and a separation of node/block transformation logic from the connection of its result. This is important in order to address the last question listed above.

The rules, that we have defined to support the connection of the transformation results of the individual node-to-sequence rules, are not strictly declarative anymore. They use the imperative part of the target pattern, the so called *do* block, to define assignment(s) that connect the results (obtained through resolution). These blocks can also include conditional statements (see rule *ActivityBlockSequence2ProcessSeq*). The use of these blocks makes our implementation imperative.

Although, in the case of simple subsequent activity nodes we could avoid using the *do* block and its imperative constructs, the presence of the block structures like decision/merge in our ADR language requires them. As we can see in rule *ActivityBlockSeq_DecisionFirst2ProcessSeq*, we have to search for the corresponding merge node for a decision node to identify the containing block structure and to connect its transformation result. This recursive search is necessary, since our ADR language does not contain structuring classes for such decision/merge blocks. Such recursive functions and imperative blocks make the transformation more complex and difficult to analyze, as we see in Section 4.2.

Assume, our source ADR meta-model would contain a structuring class, like *SequenceNode* class (see Section 4.1.2, Figure 4.5), for a decision/merge block structure (e.g., *DecisionMergeNode* class). An instance of such class would provide links to the related decision and merge nodes that build the structure and would allow ATL rules to use this information. This would simplify the implementation and eliminate the need for the recursive search helpers.

Unfortunately, such structuring classes are often absent in meta-models. Even a very elaborate modeling language like UML does not define them for structural blocks like decision/merge and fork/join. Instead, it defines separate classes for the decision and merge concepts and allows their independent use. In our ADR meta-model, inspired by UML Activity diagrams, we also have separate classes for these concepts, but we use additional OCL constraints to enforce well-formedness of decision/merge blocks and

other structures in ADR models (see end of Section 3.1). In the context of MDSD, such OCL based adjustments of a language definition are widely used.

This brings us to the second issue that we came across while implementing our ADR-to-CSP transformation case study in ATL. Like any ATL transformation, our implementation has to heavily rely on the well-formedness of the source ADR models in order to produce correct target CSP models. For instance, in order to find a merge node corresponding to a given decision node, the helper *recursiveSearchOfMerge* assumes the following: each intermediate activity node has only one outgoing edge and for each decision node there is exactly one corresponding merge node. Both of these well-formedness requirements have to be expressed in e.g., OCL and added to the source meta-model definition, in order for this implementation to work correctly.

This has been done in our meta-model based definition of our ADR modeling language. However, in general, most modeling languages in MDSD are defined by meta-models with, at most, partially defined further language constraints [FCB⁺13]. Often these constraints are defined in a text form and only rarely in OCL. The UML is an example of such mixed language definition. Such incomplete and partially informal language definition usually leads to incomplete and erroneous transformation implementations, that can not handle every valid input source model and/or also accept invalid models. We refer to such implementations as *incomplete*. Our goal is to avoid this and to find a way to guarantee that the developed transformations are complete. We address these questions in detail in Section 4.2.

When considering the results produced by the rules of our ATL implementation on a source ADR model, we know that each rule that creates elements, creates only elements allowed by the target meta-model. However, at this point ATL only enforces typing constraints of the target meta-model and ignores other constraints like multiplicity and containment/composition. Furthermore, we can not be sure that our connection rules produce a valid target CSP model from these results. If they do, and the resulting output target model for a valid input source model is valid and unique, then we can call our implementation *sound* and *deterministic*, respectively.

Unfortunately, like other transformation implementation languages on their own, ATL does not provide mechanisms to guarantee the mentioned transformation properties. It leaves it to the user to perform the source and target meta-model conformance check, to see if a source model and the resulting target model are valid. Our goal is to find a way to guarantee that the developed transformations have these properties. As in the case with the completeness property, we address these questions in details in Section 4.2.

Before we get to discuss the quality properties of model transformations, in Section 4.2, and see how, and if, they can be guaranteed in general and for our example, we consider one more implementation of the example. We look at a TGGs based implementation of our ADR-to-CSP transformation.

Implementation of the ADR-to-CSP Transformation Example with TGGs

The declarative transformation language called *Triple Graph Grammars (TGGs)* was proposed in [Sch94]. As the name suggests, it is based on a special type of graph grammars called *triple graph grammars*. In this grammar type productions' left- and right-hand sides as well as the start graph consist of three sub-graphs (*left*, *middle*, and *right sub-graph*), connected into a *triple* graph. A single TGG production describes how the three sub-graphs of its left-hand side triple graph can be simultaneously replaced by the corresponding sub-graphs in its right hand-side triple graph. The start graph (also referred to as an *axiom*) of a TGG describes the initial form of the three sub-graphs.

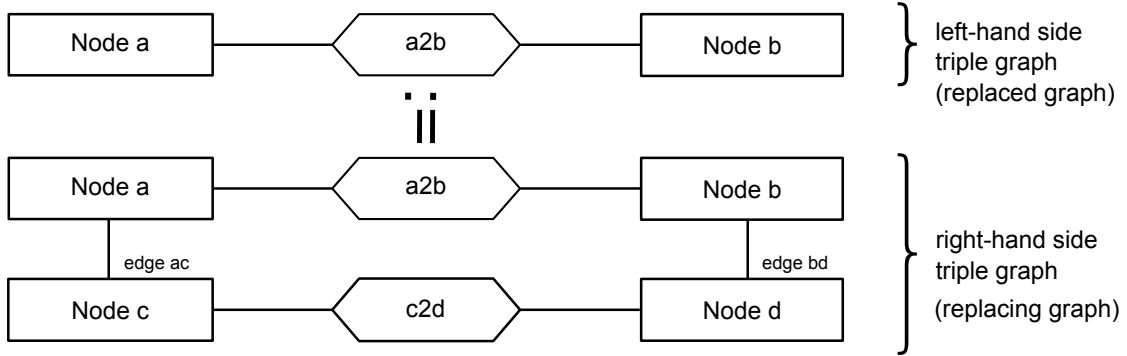


Figure 4.7: TGG production sketch

Figure 4.7 shows a sketch of a TGG production. Its left-hand side consists of three connected sub-graphs, which each contains one node – *Node a*, *a2b*, and *Node b*, respectively. The nodes from the middle sub-graph, also called *correspondence sub-graph*, are used to link the nodes of the left and the right sub-graphs. In this sketch, the node *a2b* (drawn as a hexagon), also called *correspondence node*, is used to link the nodes *Node a* and *Node b*. The right-hand side of the sketched production contains the same elements as its left-hand side plus a new triple sub-graph, linked to them via the links *edge ac* and *edge bd*. This new sub-graph contains nodes *Node c* and *Node d*, linked via the *c2d* correspondence node. In general, a TGG production is context-sensitive (see [Sch94]).

When a TGG production is applied to some triple graph, it replaces a triple sub-graph equivalent to its left-hand side in this triple graph by its right-hand side. This way, the production sketched before, when applied to some triple graph, adds new nodes *Node c*, *c2d*, *Node d*, and links them via *edge ac* and *edge bd* to the graph. In general, TGG productions have to be *monotone* – only add new elements to a triple graph. Nevertheless, one can also describe change/deletion (mark as delete) of triple graph elements using them.

If we assume that the left sub-graph, also referred to as *source*, of some triple graph already exists, we can use a TGG to create its missing correspondence and right sub-graphs, also referred to as *target*, from the start triple graph of this TGG. This method can also be inverted to obtain a source sub-graph from the given target sub-graph or

a correspondence sub-graph, when the former two exist. Depending on the part of the triple graph (i.e. model) we have available, the given TGG productions are then *interpreted* as follows: *forward rules* – for source models; *backward rules* – for target models; or *synchronization rules* – for source and target models.

Each of these rule interpretations forbids application of the productions' part that creates the already given model(s). Instead, during the execution, the elements of the respective productions' part (in the dashed blue frame in Figure 4.8) are mapped onto the elements available in the given model(s), which are then referred to as *bound*. Thereby, must be ensured that each element of the given model(s) is bound by exactly one newly produced element of some TGG production. Furthermore, re-created elements of TGG productions (in the solid blue frame in Figure 4.8) are only allowed to be mapped to the already bound model elements. The described mapping process, referred to as *matching*, replaces the process of source model parsing, usually used with grammars to find out which (TGG) productions can be applied to create this model. This matching directly depends on the monotonicity of potentially context-sensitive TGG productions.

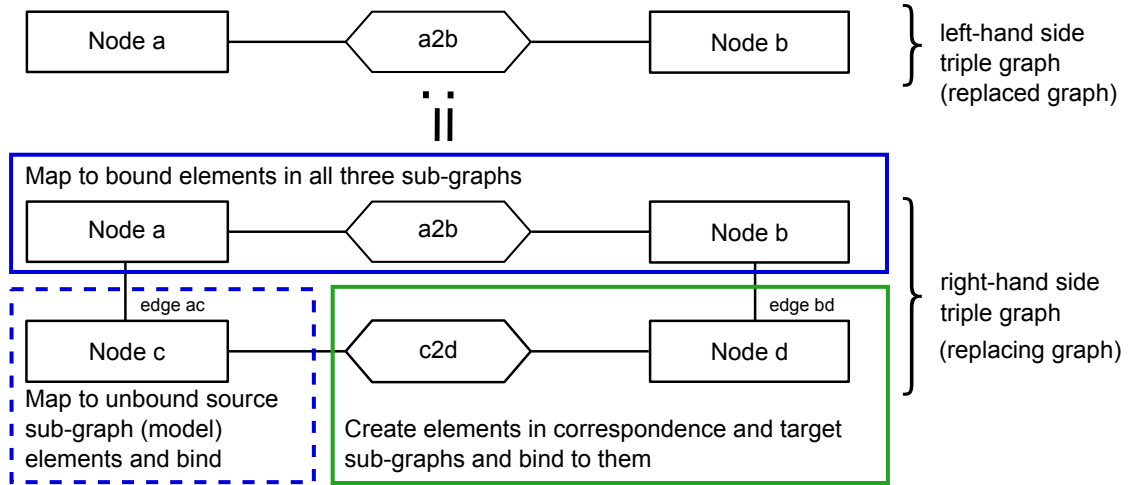


Figure 4.8: Forward interpretation of an example TGG production from Figure 4.7

Once the matching for a TGG rule has been successful, we can apply it and create the missing model elements (in the solid green frame in Figure 4.8), as specified by it. The created elements are directly marked as *'bound'*. If, at some point during the execution of an interpreted TGG transformation, all elements of the given model(s) are bound, then it is successfully completed and the sought sequence of TGG rules is obtained. Otherwise, the execution has to return to a step where an alternative TGG production is applicable (referred to as *backtracking*), to try to complete the execution and bind all elements of the given model(s).

In the context of meta-model based transformations in MDSD, the above interpreted TGG rules can be used to implement model transformations as follows. Each high-level transformation mapping is implemented by TGG rules and each sub-graph in

the left- and right-hand sides of these TGG rules is, thereby, typed by the respective source and target meta-models, and the middle sub-graph is typed over a generic or a transformation-specific correspondence meta-model. The following TGGs based implementation of our ADR-to-CSP transformation from Section 3.3 provides an example.

In general, in our TGGs based implementation of the ADR-to-CSP transformation we adhere to the same goal as in the ATL based one, considered before. We aim to implement high-level mappings, discussed in Section 4.1.2, in a compact and declarative way, and, if possible, separate rules for node-to-expression mappings from the connection rules. Since TGGs are declarative, we do not have an option to use imperative constructs and, thus, to build imperative rules. However, since TGGs, as a type of graph grammars [EEH08], are Turing complete [HP01, BDK⁺12], we can express any computation using them and, thus, also our ADR-to-CSP transformation. As we see later, this means that our TGG based implementation would become more complex and difficult to understand than the previous ATL version. Partially, this is because we have to simulate imperative constructs, like recursive search for a merge node, using declarative TGG rules and also because we need to explicitly model composition of the results of the TGG rules.

In our TGGs based implementation of the ADR-to-CSP the source and target sub-graphs of the TGG rules are typed over the ADR and CSP meta-models from Sections 3.1 and 3.2, respectively. For the correspondence sub-graphs we use the generic meta-model provided by EMorF [KW12] – our choice of TGG editing/execution tool. This meta-model contains one class *TraceInstance* with associations to the *source* and *target* meta-model elements. Note that, such typing over meta-models does not directly guarantee that these TGGs will only be applicable to valid source and will only produce valid target models, with respect to these meta-models and their possible additional constraints (see Section 4.2). Next, we consider several TGG rules of this implementation with the focus on the transformation of language structures like decision/merge block and on the connection of the results of the individual TGG rules.

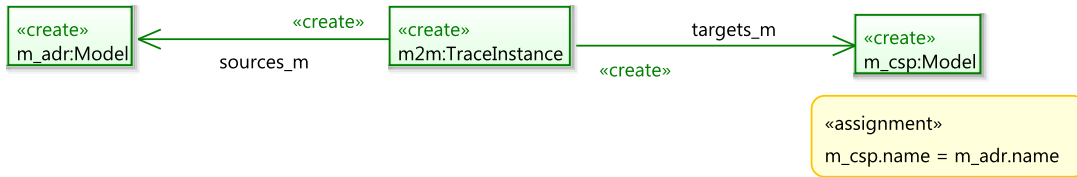


Figure 4.9: Axiom of our TGGs based ADR-to-CSP transformation implementation

The axiom of our TGGs based implementation, shown in Figure 4.9, creates a start triple graph that links an ADR model m_adr and a CSP model m_csp via a *TraceInstance* correspondence $m2m$. To depict this and further rules we use the following *compact representation* of TGG rules from [KW07]: a rule is represented by one connected triple graph, where the re-created elements are shown in blue (labeled "*«bind»*"), the newly

created elements are shown in green (labeled "*«create»*"), and the newly deleted elements are shown in red (labeled "*«delete»*"). Additionally, a rule can have *constraints* and *assignments*, like the model name assignment in Figure 4.9. As we are now interested in the forward TGG rules, we only show the assignments and constraints for them.

The next TGG rule that we consider implements the high-level mapping *Activity2Process*. This rule, shown in Figure 4.10, re-creates the start triple graph and creates a new triple sub-graph with an activity *a* linked to a process *p* (with the same name) via an *a2p* of correspondence of the type *TraceInstance*. The elements of the newly created sub-graph are linked to the re-created part via the suitable new links as follows: the activity *a* is added to the set of *activities* of the ADR model *m_adr*, and the process *p* is added to the set of *processes* of the corresponding CSP model *m_csp*. These links are important in order to connect the new elements to the already existing/re-created ones and, ultimately, to obtain a connected target model as a transformation result. Each TGG rule, apart from the axiom, has to describe such connections between its newly created and existing/re-created graph elements. This specific characteristic of TGGs has its disadvantages – it leads to more complex and unintuitive implementations.

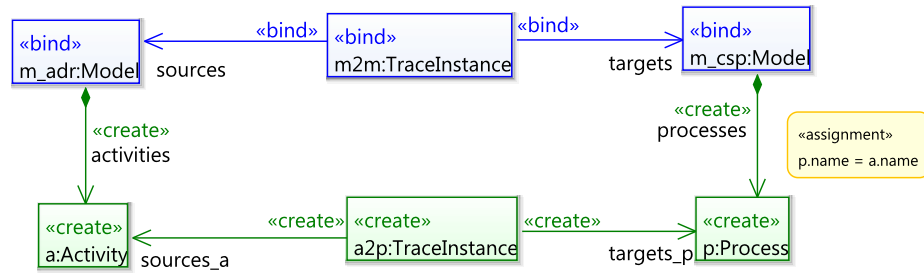


Figure 4.10: TGG rule for the high-level mapping *Activity2Process* of ADR-to-CSP

The TGG rule in Figure 4.10 creates a process for an activity without transforming its internal structure into a process expression. For this, we need to map activity nodes onto various expressions by other rules. Figure 4.11 shows such rule for initial nodes.

The TGG rule for initial nodes re-creates the triple-graph with an activity *a* connected to a process *p*, first created by the TGG rule in Figure 4.10. It creates a triple graph that contains the following parts: an initial node *i* followed by its outgoing edge *ae_out* in the source part; a process sequence *out_seq*, with its left/first sub-expression being a prefix *pref* with an event *ev* (with the related assignments), followed by a *SKIP* process in the target part; and two correspondences *an2pe* and *ae2seq* in the middle part. The correspondences *an2pe* and *ae2seq* link the node *i* to the prefix *pref* and the edge *ae_out* to the sequence *out_seq*, respectively. Plus, the node *i* and the sequence *out_seq* are linked to the activity *a* and the process *p* to connect the newly created triple graph.

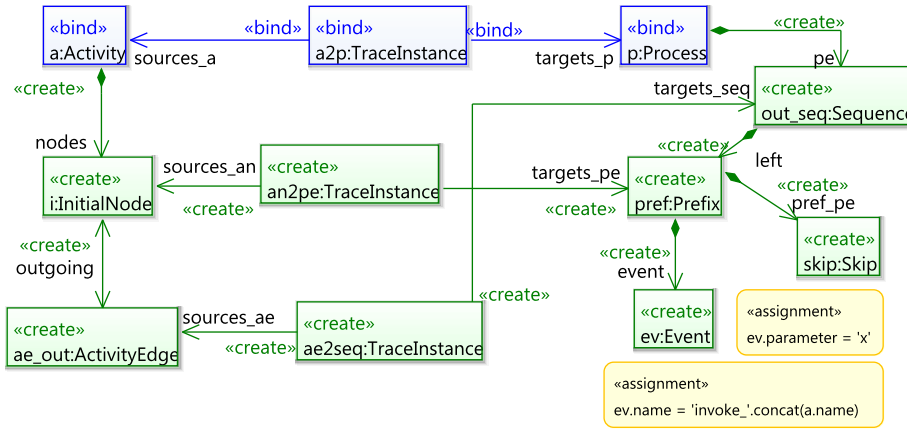


Figure 4.11: TGG rule for the high-level mapping *InitialNode2Prefix* of ADR-to-CSP

As in the case with the ATL rule for initial nodes considered before, the respective TGG rule does not only link the node i to a prefix $pref$ with the required structure, but also links this prefix to the process expression of p via a sequence out_seq . This additional sequence out_seq represents a part of our *connection mechanism*, that has already been used in the ATL implementation. The correspondence $ae2csp$ between the edge ae_out and the sequence out_seq represents the second part of this connection mechanism. With the help of this correspondence, we link the result of the TGG rule applied to the activity node that follows i as the second/right sub-expression of the out_seq sequence. Next, we show how it is done on two TGG rule examples for the simple (e.g., final) and the structure-forming (e.g., decision) activity node types.

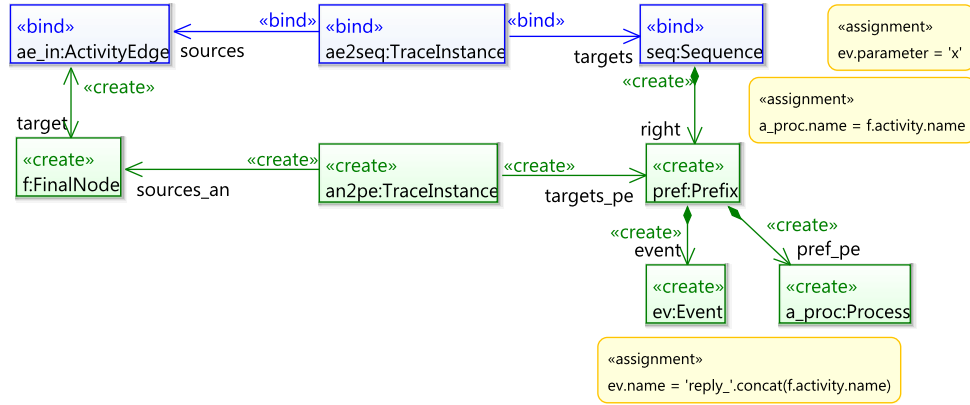


Figure 4.12: TGG rule for the high-level mapping *FinalNode2Prefix* of ADR-to-CSP

The TGG rule for the final nodes, in Figure 4.12, re-creates the triple graph that contains the linked edge ae_in and the sequence seq . This triple graph has to be created before by

a TGG rule (e.g., the rule in Figure 4.11), that is applied to the activity node/structure that precedes the final node. The rule then creates a final node f that is connected to a prefix $pref$ with an event ev , followed by a process a_proc (both with the related assignments), via a correspondence $an2pe$. Since a final node does not have any ascending edges, we do not need to create any further correspondences to support the connection mechanism. The newly created triple graph, however, has to be linked to the existing/re-created one. For the final node f the link is straightforward – it has to be the *target* of the edge ae_in to create the connection at the right place (right sequence seq). The prefix $pref$ is then linked to the sequence seq as its right/second sub-expression.

Finally, the TGG rule in Figure 4.13 implements the *DecisionMergeBlock2Condition* mapping. It transforms decision nodes into conditions and integrates those into the main expression, using our connection mechanism. This rule re-creates a correspondence $ae2seq$ between an edge ae_in and a sequence seq and, based on the discussed mechanism, connects the newly created decision node d with its outgoing edges ae_out_y and ae_out_n for both values of its condition (see related constraint) to the ae_in edge. Furthermore, it connects the corresponding sequence seq to a newly created sequence out_seq as its second/right sub-expression.

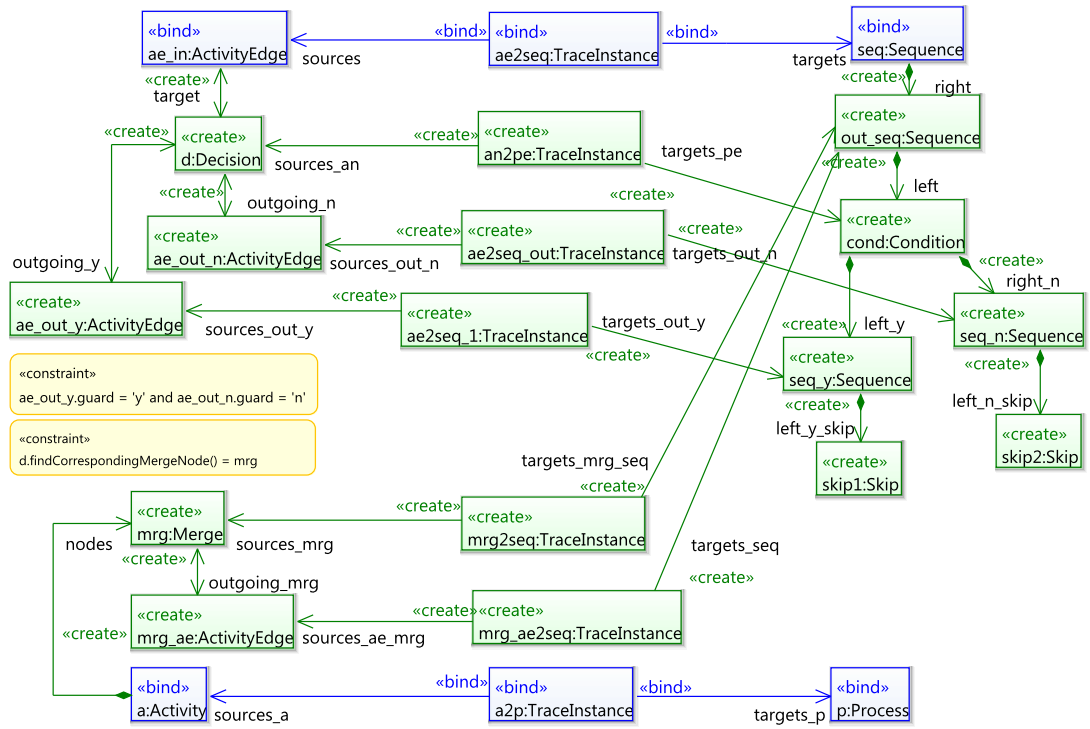


Figure 4.13: TGG rule for the high-level mapping *DecisionMergeBlock2Condition* ADR-to-CSP

Like in the other TGG rules, the sequence *out_seq* has some rule-specific expression as its left sub-expression. In this rule, it is the newly created conditional process composition *cond*, that is linked to the decision *d* through a correspondence *an2pe*. It has sequences *seq_y* and *seq_n* as its left/then and right/else sub-expressions, respectively. These sequences are linked through correspondences *ae_y2seq* and *ae_n2seq* to the respective edges *ae_out_y* and *ae_out_n* to support the connection mechanism for the target nodes of these edges. Because of the role such sequences play in the connection mechanism, only their right sub-expressions are used by other TGG rules. Thus, we assign their left/first sub-expression to a *SKIP* to avoid unset links in our target models.

Unlike by the previous TGG rules, the description of the rule in Figure 4.13 does not end here. This rule, apart from creating a condition for a decision node *d* and connecting it to the main expression, and allowing further connections of the results of other rules using the correspondences *ae_y2seq* and *ae_n2seq*, also has to ensure the correct connection of the sequence *out_seq* to its right/second sub-expression. In the case of a decision/merge block, this sequence has to be connected to the transformation result of the activity node/block that follows the merge node of the block of *d*. For a fork/join block this would be the transformation result of the node/block that follows the join node of the block. In order to create such connections, we need to have the information about the pairs of the corresponding decision and merge, and fork and join nodes.

Since our meta-model based ADR language description from Section 3.1 does not contain any structuring classes for activity blocks like decision/merge and fork/join, it can not provide such information directly. Thus, we need to implement certain functionality to get this information. In our ATL implementation, discussed before, we defined helper functions, like *findCorrespondingMergeNode* (see Listing 4.4), that perform recursive search of the corresponding node for the given one. In our TGGs based implementation we need to create some equivalent of these helpers.

Since TGGs are Turing complete, we can express such recursive helpers by TGG rules. However, specifying such recursive search functionality directly in declarative TGG rules is quite a complex task [LGJS07, AL13]. Plus, the complexity of the resulting implementation increases significantly. Thus, transformations implemented with TGG rules usually avoid the direct declarative definition of such functions. Instead, the developers use the invocation of imperative (post-)processing functions from/after TGG rules, like in [FNTZ98, VG08, AL13], that are usually implemented in imperative languages like Java. In our TGG based implementation of the ADR-to-CSP transformation we also resort to this option to reduce the complexity and use the functionality provided by EMorF to invoke such imperative search functions.

The TGG rule for the decision/merge block, shown in Figure 4.13, requires this search functionality to find and access the merge node *mrg*, that corresponds to its created decision node *d*. In the compact TGG representation, that we use, the sought merge node *mrg* is also created by the rule under the function-based condition that it corresponds to the decision node *d* (*d.findCorresponding MergeNode()* = *mrg*). This merge node

is connected to its newly created outgoing edge *mrg_ae*, and both of them are linked through correspondences *mrg2seq* and *mrg_ae2seq* to the *out_seq*, created in the target part of the rule. The later of these two correspondence triples supports our connection mechanism for the transformation results of the decision/merge blocks. It ensures that the result of the transformation of the node/block, that follows the merge node, is correctly linked as the right sub-expression of the sequence *out_seq*. At last, one can see that the node *mrg* is linked to the re-created activity *a*, which is a part of the second re-created triple sub-graph. This triple sub-graph is initially created by the rule for the *Activity2Process* mapping. This connection between the node *mrg* and the activity *a* is needed for the rule to be valid.

To summarize, the presented TGGs based implementation of our ADR-to-CSP transformation consists of the rules to transform the following: an ADR model; its resources and activities (e.g., rule for *Activity2Process* mapping); initial and final nodes of an activity (rules for *InitialNode2Prefix* and *FinalNode2Prefix* mappings), various simple activity nodes – internal action, call action, and acquire and release actions; and various block structures – decision/merge (rule for *DecisionMergeBlock2Condition* mapping), fork/join, and loop blocks. These rules are, to a large extent, similar to the rules of our ATL implementation, discussed before. In TGGs, however, we can define not only one but multiple rules with the same source sub-graph (source pattern in ATL) and, thus, can implement each high-level mapping of e.g., an ADR model in a separate rule.

Unlike in the ATL implementation, our TGGs based implementation does not contain any additional rules to connect the individual results of the above rules for the activity nodes/blocks. Instead, the connection mechanism with the creation and use of the *edge-to-sequence* connection correspondences, is integrated into the above rules. This design decision is motivated by the constraints put on valid TGG rules, where each non-axiom has to re-create some triple graph (monotonicity). The edge-to-sequence correspondences serve as such triple graphs in our transformation rules and, thereby, allow to integrate the connection mechanism.

Such forced integration has both advantages and shortcomings. On the plus side, one potentially requires fewer TGG rules to implement a transformations, and these rules ensure that the sub-graphs they create (not re-create) are connected to the already existing ones. However, on the minus side, the explicit creation and the connection of the transformation results of the individual rules are not separated, which leads to more complex and difficult to understand rules. The ATL language, which does not have such re-creation constraints, allows a more flexible and concise definition of the individual rules, as seen before. However, this can also lead to unlinked, poorly structured, and fragmented results.

Again, as in the case with our ATL implementation, we had to address the following questions in order to define our TGGs based implementation:

1. How to implement high-level mappings for one or more ADR elements by TGG rules?

2. How to connect the transformation results of the individual TGG rules?
3. How to incorporate the difference in connecting the result for a simple activity node, like an internal action, and for a block structure?

To answer the first question, we had to make design decisions about how to split and/or combine high-level mappings to be implemented by one TGG rule. As we have discussed before, with TGGs we can define multiple rules for the same source sub-graph. Thus, for simplicity and better traceability, we chose to implement each high-level mapping in a separate TGG rule (e.g., the rule for the *Activity2Process* mapping in Figure 4.10).

To answer the second question, we employed a specific connection mechanism with edge-to-sequence correspondences, that we have extensively discussed before. Unlike in ATL, where we had to use imperative statements, declarative TGG rules were sufficient here to facilitate the connection. This is due to the fact that, unlike ATL, TGGs allow the navigation of the target model fragment (right sub-graph), that was created up until the current navigating rule is applied.

Since in our TGGs based implementation, we do not have separate connection rules for the results, like in the ATL implementation, the last question on the above list is addressed within the TGG rules for the respective activity block structures. As an example of such TGG rule in our implementation, we have discussed the rule for *MergeDecision-Block2Condition* mapping (see Figure 4.13). This rule has a quite complex structure. It creates both the required condition for a decision node and connects it to the rest of the expression through a sequence. Furthermore, it directs the connection of the transformation result of the node/block that follows the considered decision/merge block. For this purpose, we had to implement the functionality to find the corresponding decision and merge nodes.

Unlike in ATL, where we had to use imperative recursive helpers for this, TGGs allow declarative definition of this functionality. This, however, comes at a price of additional complexity and increased number of rules. To avoid this in our implementation, we have used the invocation of imperative search functions, similar to those used in our ATL implementation. As in the ATL case, these functions, however, make the transformations more difficult to analyze, as we see in Section 4.2.

Assuming that our source ADR meta-model would contain structuring classes, like *SequenceNode* class, for decision/merge structures (e.g., *DecisionMergeNode* class). An instance of such class would provide links to the related decision and merge nodes, that build the structure, and allow TGG rules to use this information. This would simplify the implementation definition and eliminate the need for the search functions or the complex TGG rules simulating them. The same was true for the ATL based implementation of our example, considered before. Unfortunately, as already pointed out, such structuring classes are often absent in meta-models, including the meta-model of UML. They are also absent in our ADR meta-model.

Like any TGGs (and ATL) based transformation, our implementation also heavily relies on the well-formedness of the source ADR models to produce the correct target CSP models. For instance, the function that searches for a corresponding merge node makes an assumption that for each decision node there is exactly one merge in an activity. Our meta-model based ADR language definition contains OCL constraints to enforce such well-formedness. However, as we have discussed before, in general, most modeling languages in MDSD, including UML, are defined by meta-models with, at most, partially defined further constraints [FCB⁺13]. Often these constraints are defined in a text form and rarely in OCL, leading to incomplete and erroneous transformation implementations. Our goal is to address this problem and to find a way to guarantee the quality of the developed transformations.

When considering the results produced by the rules of our TGGs based implementation on a source ADR model, we know that, like in the ATL implementation, each rule only creates elements allowed by the meta-models. Like in ATL, the TGG rules only enforce typing constraints of the meta-models, at this point, and ignore other constraints, like multiplicity. Thus, we can not be sure that our TGG rules produce valid and unique target CSP models, and our implementation is sound and deterministic. Like other transformation languages on their own, including the considered ATL, TGGs do not provide mechanisms to guarantee the mentioned transformation properties. They leave it to the user to perform the checks for the input source and the resulting output target models. One of our goals is to find a way to guarantee that the developed model transformations have such desired properties. We focus on this question next.

4.2 Quality Assurance of Model Transformations

As a special kind of software, model transformations have to meet certain requirements of their users and, thus, to have certain quality properties. These requirements and the corresponding properties are often split into *functional* and *non-functional* sub-categories [PR10]. Here, we focus on the first functional sub-category of model transformation properties and leave out non-functional properties, like performance and scalability. A discussion of the latter type of properties and references to the related approaches can be found in [NSC13].

Functional properties of model transformations include, but are not limited to (see [LAD⁺14]), the following: guaranteed finiteness of the transformation execution on every input source model; transformation of every valid input source model into a valid output target model; generation of an output target model that is unique across executions for each valid input source model. We refer to these properties as *termination*, *soundness* and *completeness*, and *determinism*, respectively. In [LAD⁺14], the authors refer to the soundness property as a *type correctness* property. Apart from the first property, all of them are concerned with the syntax of the source and target models of the transformation.

On the other hand, there are also functional properties concerned with the semantics of the transformations and, as a result, with the semantics of the target models they generate. One of them is a *semantics preservation* property, which only holds if the considered transformation produces a semantically equivalent output target model for each input source model. Note that it is only possible to guarantee this property, if both source and target languages have explicitly defined semantics [EE08, GL12, Sem12]. Otherwise, one can only make assumptions based on the implicit semantics of the languages.

Below, in Section 4.2.1, we consider the first four of the named functional properties – termination, soundness and completeness, and determinism – in detail. Then, we look at the state-of-the-art model transformation design (Section 4.2.2) and analysis approaches (Section 4.2.3), that guarantee all or some of these properties. We do not consider the mentioned semantics preservation property here, since we assume that our source and target languages are only defined on the syntactical level via meta-models.

4.2.1 Four Functional Properties of Transformations

We now define the chosen functional quality properties starting with *termination*. The termination property guarantees that each execution of the considered model transformation eventually stops. We use the following definition of this property.

Definition 4.2.1.1 (Termination). *A transformation T terminates iff any execution of T on any input source model M_s is performed in a finite number of finite steps.*

This property is generally difficult to guarantee if the considered transformation is defined using a Turing complete language like ATL or TGGs. Later, we consider several proposed approaches to support/guide model transformation development process [EEdL⁺05, K s06, VVGE⁺06, BLA⁺11, LKR13] and analyze the developed [WKS⁺09, CCGdL10] model transformations, to guarantee their termination. We classify these approaches, based on the quality validation method(s) they use, into the following: *quality-by-construction* and *quality-by-checking* categories (inspired by [LAD⁺14]).

Further properties, that we are interested in, are *soundness* and *completeness* of model transformations. These properties are closely linked together. They guarantee that a model transformation can transform any valid input source model (completeness) and that the transformation result, for a valid input source model, is a valid output target model (soundness). We use the following definitions of these properties. Thereby, we see the notion of *syntactical correctness* [EE06, LAD⁺14] as an equivalent to soundness.

Definition 4.2.1.2 (Soundness). *A transformation T is sound iff its result on a valid input source model M_s – $T(M_s)$ – is a valid output target model.*

Definition 4.2.1.3 (Completeness). *A transformation T is complete iff it can transform any valid input source model M_s .*

These last two definitions contain the notion of a *valid source/target* model. The interpretation of this notion depends on the technique used to describe the source and target languages of the transformation T . When the meta-model based technique (see Section 2.2) is used, a valid source/target model is a model that conforms to the given source/target language meta-model, respectively. Note that these definitions do not mention invalid input source models and do not require a transformation to reject them, although, this would prevent potential inconsistent and unpredictable results.

Another notion, that we also need to explicitly define, is used in the completeness definition. It is concerned with the following question: when *can* a transformation *transform* an input source model? We define this *ability to transform* notion as follows.

Definition 4.2.1.4 (Ability to Transform). *A transformation T can transform an input source model M_s , or M_s is transformable by T , when none of the execution steps for T fail on M_s .*

The actual execution steps of T , that are performed, highly depend on the following two aspects: the way this model transformation has been implemented and the execution engine used to run it.

These soundness and completeness properties are also generally difficult to guarantee if the considered transformation is defined using a Turing complete language, like ATL, TGGs, or Java. Although, for soundness, one can show that individual transformation rules produce meta-model conform results [Küs06], this does not guarantee that the connection mechanism for these results produces valid target models. Especially, if such connection mechanism contains imperative constructs, like in our ATL implementation of the ADR-to-CSP transformation case study in Section 4.1.3.

For completeness, one can define coverage criteria and tests [WKC06, KAER07, GC12] to ensure that the considered model transformation can transform various combinations of the source language elements. However, source languages can potentially contain an infinite number of distinctive models, which is defined through unbound multiplicities (denoted by $*$) in meta-models. Thus, such criteria and tests usually can not cover all possible combinations of elements to guarantee the completeness property in full.

Later, we consider several works that propose various quality-by-construction [Küs06, EHS09, EHS09, LKR13] and quality-by-checking [WKC06, KAER07, CCGdL10, SMR11, GC12] approaches to guarantee (often partial) soundness and completeness of model transformations. Some of these approaches [Küs06, LKR13] use the notion of syntactical correctness instead of soundness.

The last model transformation property, that we are going to consider in detail is *determinism*. This property guarantees that each execution of the considered transformation generates the same result (output target model) for a given input source model. It forbids unpredictable transformation behavior, that is often hard to diagnose and eliminate. We use the following definition of this property.

Definition 4.2.1.5 (Determinism). *A transformation T is deterministic iff its result $T(M_s)$ for a valid input source model M_s is the same in each execution.*

Again, like the previously considered properties, determinism is difficult to show. The main idea, that most approaches follow to guarantee this property, is the analysis of the dependencies between the transformation rules [HKT02, Kü06, HEOG10, CCGdL10]. This way, one can find a potentially non-deterministic transformation behavior, that can cause non-deterministic results.

Next, we consider several works that propose various quality-by-construction [Kü06, HEOG10, BLA⁺11, LKR13] and quality-by-checking [KAER07, WKS⁺09, CCGdL10, GC12] approaches to guarantee determinism. Most of these approaches [Kü06, WKS⁺09, BLA⁺11, LKR13] use the notion of *confluence*, that implies our property of determinism and is equivalent to the notion of determinism used in [LAD⁺14]. The confluence property, that originates from the area of state transition systems of theoretical computer science, states that the order of application of rules (transitions) in a model transformation with this property, is not important for the result (see [EE06, EEPT06]).

4.2.2 Quality-By-Construction: Transformation Development

Now, we consider several approaches to the development of model transformations that guarantee *one or more of the above functional quality properties* for the developed transformations. Such quality assurance *by-construction* aims to provide a convenient integrated way to create high quality model transformations.

Note that we do not consider works like [dLT04], where a particular transformation is shown to have various quality properties based on its structure/definition here. Instead, we focus on the approaches that provide languages/techniques and define criteria that the developers can use to build their own transformations with guaranteed quality properties. This means that, in the considered quality-by-construction approaches, properties are shown to hold for all model transformations defined using them. In the most recent surveys of model transformation analysis approaches in [ALS⁺12, CS13, ARW13] these methods are categorized as *transformation-independent*. We, however, put them into a separate, not analysis-related, sub-category, that we have already referred to as quality-by-construction.

The main techniques used in the quality-by-construction category of approaches are the following: direct mathematical proofs [EEdL⁺05, EHS09, EEHP09, Kü06, HEOG10, BLA⁺11, LKR13] and re-use of proving techniques [VVGE⁺06] for existing well-known formalisms (e.g., Petri nets) via translation [ALS⁺12].

The authors of [BLA⁺11] prove termination and confluence of their own transformation language *DSLTrans* by induction, using the layered syntactical structure of the transformations defined in this language. This result guarantees the proven properties

by-construction for each model transformation defined in the proposed language. However, this language is not Turing complete as it is free of loops and recursion and, thus, restricts the scope of model transformations that can be described using it.

In [EEdL⁺05], the authors address termination of model transformations based on algebraic graph transformations [CMR⁺97]. They define criteria for transformation rules based on the deleting and non-deleting layers they are assigned to. These criteria enforce a certain precedence relation over the rules creating and matching/deleting the same elements. Based on these criteria the authors inductively prove termination of all graph-based model transformations fulfilling them. The authors also point out, that the assignment of such layers to the rules in a model transformation can be difficult.

Other works [EHS09, EEHP09], closely related to [EEdL⁺05], consider model transformations defined using TGGs and prove their termination, soundness, and completeness. In the authors' definition of these properties, the source and target models of the transformations are those generated by the respective (source/target) parts of the defining TGGs. In [EHS09, EEHP09], the authors reference their previous work in this direction, which we, therefore, do not consider explicitly here. In [HEOG10], the authors extend the work from [EHS09, EEHP09] by proposing an approach to guarantee determinism of TGGs based model transformations. The main idea is to generate forward translation rules to extend the standard forward rules by the attributes, that keep track of the already translated elements. They show determinism through local confluence and refer to it as *functional behavior*.

In [LKR13], the authors propose an approach for the automated derivation of correct-by-construction transformation implementations from high-level specifications. They use conjunctive-implicative OCL constraints and UML Class diagrams [UML] to define such high-level specification of model transformations, and provide three automatic and proven-correct transformation implementation strategies based on these specifications. The generated implementations are then expressed in a simple procedural language.

The quality properties of the resulting transformation implementations, that are shown in [LKR13], are termination, confluence, and syntactical correctness (soundness). Using this approach one can create quality model transformation implementations. However, the creation of OCL based specifications of model transformations that fulfill several side conditions (see [LKR13] for details), can become a complex task. To simplify this task and facilitate an easier use of their development method, the authors later propose several model transformation patterns in [LKR14].

In [Küs06], the author formulates sufficient criteria for termination, syntactical correctness (soundness), and confluence of graph-based model transformations with control structures. He also provides the relevant manual proofs based on these criteria. The transformation rules considered in [Küs06] have a special *compound* structure with a source model pattern and a target model production. Both of these rule parts can,

in their turn, contain multi-objects. The control structures used to order the compound rules within a so-called *transformation unit* [KKR08b] include sub-sets (non-deterministic choice from a set), sequences, and loops (applied as long as possible).

The sufficient criteria defined in [Küs06] for syntactical correctness concern both the individual compound rules and the transformation units. Thereby, the author uses a more loose notion of syntactical correctness (soundness) for the rules than in our definition, by only requiring their parts to be instances of the respective source/target meta-models. This, however, on its own does not guarantee soundness (syntactical correctness) of the transformation. Thus, several more complex and restricting criteria are also added in [Küs06] to achieve this property. In [Küs06], the author admits that a stronger notion of syntactical correctness (our soundness) is difficult to achieve, since one must take into account all transformation rules as well as their relationships.

The sufficient criteria defined in [Küs06] to guarantee transformation termination and confluence also concern both the individual rules and the transformation units with loops. The defined termination criteria aim to ensure the reduction of the number of non-terminals in the constructed target model after each application of a rule from a loop part of the transformation unit. This criteria significantly restricts the allowed target rules. The defined confluence criteria aim to eliminate conflicting transformation rules (critical pairs [EPT04]), which re-ordering can produce different results, by requiring parallel independence of compound rules. They also consider the match selection, as the second source of non-determinism, and restrict the number of them in any source model for each currently applicable rule to at most one for all rules in sequences and sub-sets of the unit. For the rules in the loops, the author employs critical pair analysis [EPT04] to check, that the transformation is still confluent. When the critical pairs are not confluent, the author suggests resolving the problem by additional constraints on the source/target language or by changing the rules. Overall, the approach suggests several useful quality criteria for a model transformation. However, the restrictions that it contains limit the applicability.

In [VVGE⁺06], the authors prove termination of (a special class of) graph-based model transformations by translating them into Petri nets. Thereby, they abstract from the instance models and re-use the existing termination proofs for this formalism. If they manage to prove that the Petri net runs out of tokens in finitely many steps regardless of the initial marking, then the transformation terminates. By solving algebraic inequalities, the authors obtain sufficient criteria to either prove that a transformation is terminating or get a 'maybe' answer. This approach is useful in the considered class of transformations but its inconclusive answer might be insufficient.

The approaches considered above provide examples of different methods to develop model transformations with quality properties guaranteed by-construction. They include creating a dedicated language [BLA⁺11] and identifying sufficient quality criteria [Küs06, EHS09, EEHP09, HEOG10] as well as creating proven-correct transformation implementation strategies [LKR13]. To the best of our knowledge, these works represent

the main published approaches in this area. None of the considered approaches addresses all four of our properties of interest, defined in Section 4.2.1.

Although, the combination of approaches from [EHS09, EEHP09] and [HEOG10] covers a similar set of properties for TGGs based model transformations, their definition of these properties significantly differs from our. For instance, for soundness and completeness, the authors regard all source/target models of the transformations, generated by the respective (source/target) parts of the defining TGGs, as valid. They do not demand conformance to some additional source/target language definition artifact like meta-models, like we do. This issue has also been pointed out in [NHBE14], where the authors show the first of the components needed to guarantee completeness (domain completeness in [NHBE14]) of model transformations based on non-deleting graph grammars (like TGGs). They propose a method to show that the source language of a given TGG transformation is included in the language, generated by the source rules of this TGG. For this purpose, the authors introduce several conditions and require a fixed upper bound for the size of source models and conflict-freeness of certain rules (see [NHBE14] for details).

Now, we consider the applicability of the above approaches for the development of our ADR-to-CSP example transformation from Section 3.3. The approach proposed in [BLA⁺11] with a Turing incomplete language without loops and recursion is not sufficient for this purpose, since we need to locate structures in source models (e.g. decision/merge blocks) and require such constructs for this purpose. In the case, when we add structuring classes to our meta-model for the source language (see Section 4.1.2, Figure 4.5 for details), this problem would be solved. Yet, with the approach from [BLA⁺11] we can then only guarantee termination and confluence.

If we apply [LKR13] to develop our ADR-to-CSP transformation example, we can employ OCL to specify required source language structure(s) (e.g. decision/merge blocks) as recursive pre-conditions of transformation rules. Such OCL conditions, however, will become quite complex, and it is not clear if the approach supports all of the OCL constructs required to express them or only a sub-set. In the case, when we add structuring classes to our meta-model for the source language, the complexity of OCL constraints will decrease. With the approach [LKR13] we can guarantee termination, soundness, and determinism of the transformation implementations generated from our OCL constraints. Yet, determinism property is not guaranteed.

When we apply a combination of approaches from [EHS09, EEHP09] and [HEOG10] to develop our ADR-to-CSP transformation example, we need to specify it using TGGs. This has been done in Section 4.1.3, where we also point out that the need to locate structures in source models increases the number and the complexity of the rules of such TGGs based transformations. In the case, when we add structuring classes to our meta-model for the source language, the complexity of TGG rules will decrease. With the approaches from [EHS09, EEHP09, HEOG10] we can, at a first glance, guarantee all four quality properties of interest – termination, soundness and completeness, and determinism – for the resulting TGG rules. Yet, as we have mentioned before, the

definition of soundness and completeness properties used by these authors differs from the one we gave in Section 4.2.1.

To apply the approach from [Küs06] to develop our ADR-to-CSP transformation example, we need to specify it using the compound rules proposed there. Although model patterns in the compound rules have to be weakly conform to the source/target meta-model, respectively, it is not entirely clear how the non-terminals on the target side of the rules are chosen and where they are defined. Also, we need to ensure several complex and restricting criteria like non-terminal reducing nature and parallel independence of the rules (see [Küs06] for details) for our transformation definition. Another criteria, that is also quite restrictive and requires analysis of the source language, is concerned with the unique match selection for each currently applicable rule. This criteria is particularly difficult to show, since a source language can contain infinitely many models.

Once all the criteria from [Küs06] have been met, our transformation is guaranteed to terminate, be confluent, and syntactically correct. Thereby, the last correctness property is defined by the author differently – he only requires source/target patterns of transformation rules to be instances of the respective source/target meta-models (without multiplicity constraints). We, on the other hand, in our definition of syntactical correctness (soundness) require the transformation to generate valid (e.g. meta-model conform) output target models for valid input source models.

The approach, that we propose in Chapter 5, combines and extends several ideas from [Küs06] and [EHS09, EEHP09, HEOG10] to guarantee all four quality properties, that we focus on, in the way we defined them in Section 4.2.1. With this approach, we also aim to simplify the development of model transformations (and its result) that map language elements/structures onto each other, like our ADR-to-CSP example.

4.2.3 Quality-By-Checking: Transformation Analysis

The other group of approaches to quality assurance for model transformations, that we consider now, is concerned with *checking* already developed transformations with regards to *one or more of the four functional quality properties*, defined in Section 4.2.1. Such checks are used when the quality-by-construction methods from Section 4.2.2 can not be applied due to the complexity and/or nature of the developed transformation. The checks are either performed during the execution of the considered transformation (*dynamic analysis*) or on its model/specification (*static analysis*). To the first category belong the model transformation testing approaches [KAER07, BKE11, GC12]. To the second category belong various formal model-based verification approaches by model checking [WKS⁺09], constraint solving [CCGdL10], and theorem proving [SMR11] as well as checking of static metrics [WKC06].

The most recent extensive surveys and classifications of model transformation analysis approaches have been published in [ALS⁺12, CS13, ARW13]. Thus, here, we only

demonstrate several selected works for different analysis sub-categories, that focus on our four quality properties, and refer to the mentioned surveys for further details.

First, we consider example approaches for the testing of model transformations. Although, these methods can only show the presence of errors and not their absence, we mention them for the complete picture. The errors, that we are interested in, are violations of one or more of our four properties from Section 4.2.1 on the tested source models. In [GC12], the authors are interested in the coverage of the ATL model transformation internal structure, represented as a dependency graph, for their test case generation. The coverage of the source meta-model is then derived from those test cases to estimate the degree of completeness of the considered model transformation w.r.t. the source language. The soundness and determinism of the transformation is then partially checked using the generated test cases.

Like in [GC12], the authors of [KAER07] also consider the white-box approach to model transformation testing. They propose the following three techniques for generating test cases based on a suitable fault model: rule-based meta model coverage testing, constraint-based coverage testing, and rule pair based testing. The transformation fault model, used by this approach, contains the following most-likely sources of transformation malfunctioning: insufficient source meta-model coverage (completeness); creation of invalid target models (unsoundness); creation of different output target models for a given input source model (non-determinism); and other semantics related errors. Some of these errors are strongly correlated to the transformation soundness, completeness, and determinism properties, that are of interest to us.

In [BKE11], the authors focus on the quality (coverage) of the test cases performed in the approaches, like [KAER07, GC12], in the context of the testing of model transformation chains. By computing a footprint, the approach identifies the missing and redundant test cases in a test suite.

Next, we consider examples of model transformation verification approaches by model checking, constraint solving, theorem proving, and checking of static metrics.

In [WKS⁺09], the authors propose verification of model transformations modeled with their own DSL called *Transformation Nets*. This DSL is defined on top of the Colored Petri Nets (CPNs) [Jen87] formalism, allowing the application of the existing techniques for the state space exploration and the analysis of firing sequences. The properties considered by the approach in [WKS⁺09], that are of interest here, include termination, rule liveness (related to completeness), and confluence (implies determinism). All these properties are shown through analyzing the CPN for a transformation and whether it reaches different types of markings. Although this approach demonstrates beneficial use of CPNs for the verification of several of our properties of interest, it has the following drawbacks: it requires learning of a new DSL, and a detailed and elaborate design of model transformations with it; additionally, it suffers from the state space explosion problem.

In [CCGdL10], the authors propose verification of model transformations, expressed with TGGs and QVT-Relations. They automatically derive OCL invariants from the transformation specifications and their source and target meta-models (possibly) with constraints. These invariants are then used as an input to a constraint solver and as a basis for formulating the verification properties to be checked. The properties considered by the approach include, but not limited to, soundness and completeness (totality in [CCGdL10]), and determinism. The termination is ensured through the declarative nature of the considered model transformations and the additional condition, that requires each type of mapping to be created by at most one rule. Although, one can guarantee all four of our properties of interest using the approach from [CCGdL10], the conditions that it puts on the considered transformations significantly limit its applicability. Additionally, the fact that the approach uses bounded verification (bounded constraint solving) guarantees its termination but also makes it incomplete.

In [SMR11], the authors consider verification of model transformations described in QVT-Operational [QVTb], using the interactive theorem prover KIV [BRS⁺00]. For this purpose, they provide a method to obtain algebraic specifications for source/target meta-models and models, and define a calculus for QVT-Operational in the theorem prover used. The main goal of this work is to prove that a transformation in QVT-Operational always generates type correct (valid) target models and has certain semantic properties. The former property is of interest to us as it relates to our termination, and soundness and completeness properties. However, the focus of the paper is set on the type-correctness of target models (soundness property), and it is not clear how termination and completeness are shown. Furthermore, there are several well known drawbacks of the approaches based on theorem proving, including the complexity of the required calculus/proofs and the need of expert knowledge to create them.

In [WKC06], the authors choose a different direction and propose several criteria for the source/target meta-model coverage by model transformations. By checking these criteria, they aim to address the question of model transformation completeness. The proposed criteria concern the coverage of various core modeling constructs of meta-models like attributes, classes, and associations. Although a transformation developer can benefit from such metrics, the full source meta-model coverage by these criteria would not guarantee transformation completeness.

The approaches considered above provide examples of different methods for checking the four quality properties of model transformations, defined in Section 4.2.1. Only one of the considered approaches, from [CCGdL10], addresses all four of them for declarative transformation specifications. This approach, however, as mentioned before, puts additional conditions on the analyzed transformations which can not always be met, and it is incomplete (bounded). It is also possible to combine the above approaches to get around these conditions and show all four properties. However, this leads to a significant increase of complexity, since the considered approaches use different input formats and require different expertise.

Now, we consider the applicability of the above approaches for the analysis of our ADR-to-CSP example transformation from Section 3.3. Both testing approaches from [KAER07, GC12] can be applied to check the properties of our example transformation either directly on its ATL implementation (Section 4.1.3) by [GC12], or by [KAER07], that requires an additional definition of conceptual rules. Yet, due to the incompleteness of testing, we can not fully guarantee the properties using these approaches.

The approach in [WKS⁺09] seems to be applicable to our example, if we model it using the Transformation Nets DSL they propose. It is not clear how complex this modeling task can become and whether this language can express the operations that locate the structures in input source models (e.g. decision/merge blocks). In the case, when we add the structuring classes to our meta-model for the source language (see Section 4.1.2, Figure 4.5 for details), this complexity will most likely be lower. Yet, with this approach we can not guarantee transformation soundness and completeness (in full).

The approach proposed in [CCGdL10] might be applicable to the TGGs based implementation of our example (Section 4.1.3), if we also provide TGGs for locating structures in input source models (e.g. decision/merge blocks). Furthermore, we need to ensure that none of the TGG rules violates the condition that: *"each type of mapping is created by at most one rule"*. In our understanding, this condition is, however, violated by our implementation (edge-to-sequence mappings) and, thus, we can not apply the approach from [CCGdL10] to analyze it. In the case, when we add structuring classes to our meta-model for the source language, this problem would not be solved.

The approach proposed in [SMR11] can be applied to our example ADR-to-CSP transformation, if we implement it in QVT-Operational. This transformation language allows us to use loops/recursion to locate structures in input source models (e.g. decision/merge blocks). Thus, we do not need to introduce structuring classes in our source meta-models for this purpose. With the approach from [SMR11], we can then guarantee transformation soundness, and (possibly) termination and completeness. Yet, this might require interaction with KIV theorem prover, used by the authors, and, thus, an additional expertise in this complex domain.

Finally, we can apply the coverage criteria from [WKC06] to check if our example ADR-to-CSP transformation implementations from Section 4.1.3 cover the source meta-model. This, however, as mentioned before, does not guarantee their completeness.

4.3 Re-Use of Model Transformations

Once a model transformation has been developed and its quality has been guaranteed (to a larger or smaller extent) by the approaches mentioned in Section 4.2, it can be used in MDSD projects. Since its development and quality assurance usually take a significant amount of time and developer resources, it is in the interest of each transformation

developer to re-use the existing transformations. This re-use idea is also consistent with the central goals of MDSD (see Chapter 2).

There are two fundamental ways of re-using model transformations:

1. *Transformation chaining*, where single transformations are connected through their inputs/outputs, without any alterations to them, and executed in a pre-defined order (e.g., sequential or parallel) to produce the combined result (output target models) from the given input source models.
2. *Transformation adaptation* using techniques like *superimposition* [Wag08], where a new model transformation is defined by changing (e.g., overriding, extending, etc.) the original model transformation. The superimposition technique has been chosen here to represent such re-use strategies, that *change* the existing transformations and/or their source/target meta-models. We, however, focus on the *non-changing* (non-invasive) re-use strategies from 1 and refer to [WKK⁺12, KSW⁺13] for a detailed survey of invasive approaches.

These different re-use techniques are applicable in different cases during the software development process. The choice between them highly depends on the nature of the developed complete model transformation, the nature of source and target languages and models, and the nature of the model transformations that are available for re-use.

Next, we consider each of these transformation re-use strategies in more detail, mention several approaches that support them, and sketch their application examples for the transformation of our example layered software system, described in Chapter 3.

4.3.1 Transformation Chaining

Transformation chaining is a collective term for all kinds of compositions of individual model transformations including sequencing, parallelization, conditional choice, etc. Its main idea is to combine the existing individual model transformations into one by connecting them through their outputs/inputs (source/target models), as sketched in Figure 4.14. Thereby, the individual model transformations are usually treated as black-boxes that can not be adapted/changed.

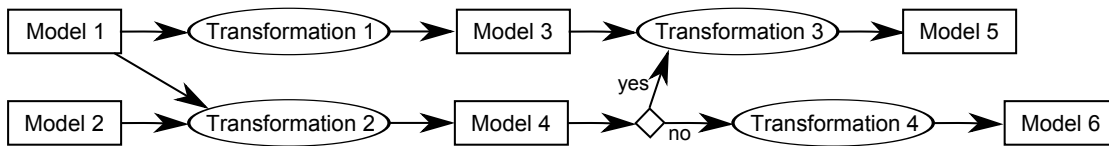


Figure 4.14: Transformation chaining sketch

Transformation chaining requires individual model transformations to be *compatible* with their direct neighbors – the source language of a transformation has to include the target language of its preceding neighbor and the target language of a transformation

has to be included in the source language of its succeeding neighbor. The respective languages are usually described using meta-models (see Section 2.2). Furthermore, in the case of parallelization, separation of source/target models between the re-used model transformations requires their independence of each other and an *absence of direct inter-model links* in the source and target models.

Several authors [WH05, Old05, Kle06, VAVB⁺07, RRLB09, GdLK⁺10, HKA10, AEM12, EABP12, SHNG12, BPOB13] have considered model transformation chaining. Most of them proposed to use UML Activity and/or UML Component diagrams, or similar notations, to chain several re-used model transformations and model their parameters (input models) to create a new transformation.

In [WH05], the authors propose an approach to address the incompatibility of chained model transformations. It is based on the translation to and from the transformation-specific meta-models for each transformation in the chain, if required. The authors encapsulate this translation into a so-called *side transformation pattern* and use it to compose model transformations in a re-usable way. The side pattern consists of a fork, a source model projection, a computation (individual transformation execution), and source model merge operations. This approach promotes the isolation of the transformation logic from its application context (meta-models of source models) to allow its re-use. However, usage of such side pattern adds further complexity to the structure of model transformation chains and their development process. The merge operation can be particularly difficult to implement. Furthermore, it is not clear how the approach deals with inter-model links by multiple source/target models.

The authors of [EABP12] also focus on the challenge of determining if two chained transformations are compatible. More precisely, that they do not perform any conflicting operations. Since such conflicting operations, like deletion of model elements required by the next model transformation, or creation of model elements prohibited/unknown for the next model transformation, can break the developed chain. The authors propose a static analysis method to identify if the given model transformations can be chained. Their method translates transformations and their meta-models into the *Alloy* [Jac06] specification language and uses its SAT solver to identify possible conflicts. To reduce the complexity of the resulting Alloy specification and of its solving, the proposed method only considers one source and one target model and, thus, does not address inter-model links.

In [SCGM08], amongst others, the authors propose an approach to deal with inter-model links between the target models, that are generated by different re-usable model transformations, composed into a chain. The problem arises when we need to relate model elements, that are typed by independent meta-models, or there are no constructs/classes to model such links, in the case of the same meta-model. In their example transformation chain several inter-linked PSM target models are derived from one PIM input source model (see Chapter 2). The idea of the proposed approach is to create a new meta-model for bridging the existing target meta-models. There, for each class that needs to be linked with another class, a new sub-class with the respective link is

created. This new meta-model then contains all of the required link types between the target models.

Once the bridging meta-model in [SCGM08] is created, the authors describe a subsequent model transformation integration, needed to create the links (bridges) between the target models. This integration is defined as a separate model transformation that imports all of the re-used model transformations and contains the refinement rules, used to create the links. The authors do not consider incompatible transformations, as they assume the same source meta-model for all of the re-used model transformations.

On the other side, the authors of [AEM12] focus on the automation of model transformation chaining. They classify available re-usable model transformations, using a feature model, and analyze their constraints (including compatibility) and dependencies. Based on this information, the proposed approach allows the transformation developer to configure/select features and, thus, the model transformations that implement them. The method then attempts to generate valid model transformation chains based on the given selection. In [AEM12], the authors only generate sequences of model transformations. Thus, it is not clear if their method supports multiple source/target models and inter-model links.

The re-use approaches considered above and all other transformation chaining approaches, that we are aware of, do not address model transformation chaining for input source models with inter-model links. We aim to fill in this gap and propose a model transformation chaining approach based on model weaving [DdF07]. We discuss it in details in Chapter 6.

With regards to our example layered system (Section 3.1) and its transformation (Section 3.3), transformation chaining can be applied here in different contexts. For instance, if one or more of our input source models would be described using UML Activity diagrams, we would need an additional model transformation to obtain ADR models from them. The result of this transformation would then be translated into CSP by our main ADR-to-CSP transformation. In the sense of the transformation classification from Section 4.1.1, both transformation in such sequence are translations.

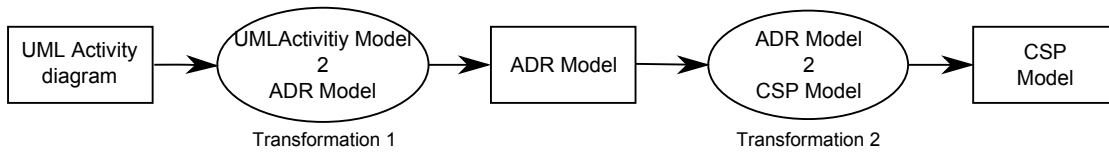


Figure 4.15: Transformation sequencing: UML Activity diagram to CSP through ADR

Figure 4.15 shows a sketch of a model transformation sequence for the purpose described above. It contains the following two transformations (drawn as circles): the first one, to transform a UML Activity diagram into an ADR model and, the second one, to transform the resulting ADR model into a CSP model. These input and output models (drawn as rectangles) are connected to the transformations via in/out arrows.

The transformation parallelization technique also perfectly fits into our ADR-to-CSP transformation case study. There, we want to transform ADR models of individual system layers and their inter-model links, that connect them together. For this purpose, we want to re-use our ADR-to-CSP transformations, available for the individual layer models, and compose their results. This is possible if our target language provides a *composition mechanism*. In CSP one of such composition mechanisms is, for instance, a parallel composition operator $\parallel$ (see Section 3.2 for details). In the sense of the transformation classification from Section 4.1.1, the single transformations of ADR models into CSP can then be classified as translations and the composing model transformation as a refinement.

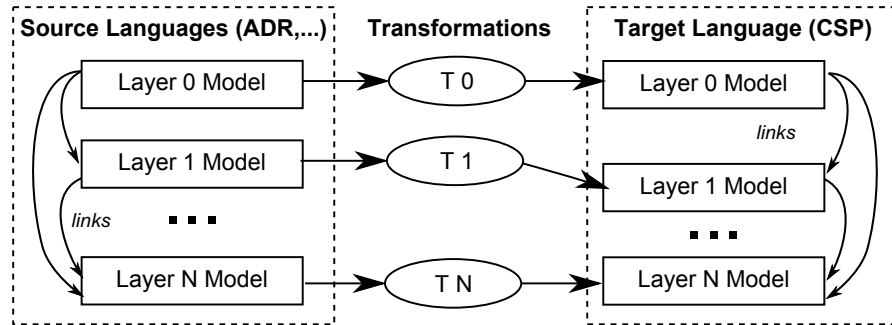


Figure 4.16: Transformation parallelization: ADR models of system layers and their interactions/inter-relations into CSP

Figure 4.16 shows a sketch of transformation parallelization for our example ADR-to-CSP transformation. It shows how an N-layer system, that consists of models in one or more languages (in our example only ADR), is configured through inter-model links, and is transformed by multiple parallel model transformations. Their results in CSP are then composed into the final target model. This sketch in Figure 4.16 abstracts from the details of concrete source/target models and model transformations, used in this parallel chain, and only names them. Later, in Chapter 6, we give a more detailed description of such chain, when we describe our transformation parallelization approach for layered systems with source inter-model links, based on our example system and the ADR-to-CSP transformation.

4.3.2 Transformation Adaptation: Superimposition

Transformation extension using a technique called superimposition was proposed in [Wag08]. Its main idea is to re-use the existing model transformations not as a whole, as in chaining, but on the level of the individual transformation rules. This technique allows the transformation developer to override or extend one or more of the rules of an existing model transformation while keeping the rest unchanged. The newly created model transformation then combines the rules of the re-used transformation with the newly created overriding or extending rules of the *superimposed* transformation. This can be

seen as a cross-transformation extension of the rule inheritance mechanism supported by such transformation languages as ATL.

The use of this re-use mechanism requires changes in the transformation engine, that execute the model transformation with superimposed rules. There exists such support of superimposition for ATL and QVT-Relations (see [Wag08] for details).

Concerning our example layered system and its transformation described in Chapter 3, superimposition can also be applied here in different contexts. For instance, to override a transformation rule for e.g., an internal action, to *split* the event of the produced process prefix into two partial events for the event *start* and *end*. We will discuss the transformation scenario where this is needed later. In the sense of the transformation classification from Section 4.1.1, such a model transformation is a refinement.

```

1 module adr2csp;
2 create csp : CSP from adr : ADR;
3
4 rule InternalAction2SeqPrefixAndChannel{
5 from in_ac : ADR!InternalAction
6 to
7   out_seq : CSP!Sequence(left <- pref_s),
8   pref_s : CSP!Prefix(event <- evs,pe <- pref_e),
9   pref_e : CSP!Prefix(event <- eve,pe <- CSP!Skip.newInstance()),
10
11   evs : CSP!Event( name <- in_ac.name + '_start',
12                   direction <- #output, parameter <- 'x'),
13   eve : CSP!Event( name <- in_ac.name + '_end',
14                   direction <- #output, parameter <- 'x'),
15
16   out_ch_ac : CSP!Channel( ...
17 }

```

Listing 4.7: Superimposed rule *InternalAction2SeqPrefixAndChannel* for the ATL transformation from Section 4.1.3

Listing 4.7 shows a superimposed event splitting transformation implemented in ATL. This model transformation only contains one rule for the transformation of an internal action into a process prefix. It overrides the rule with the same name *InternalAction2SeqPrefix AndChannel* from the original transformation we have considered in Section 4.1.3. Instead of one process prefix it now produces the following two: the one with the *start* event, followed by the one with the *end* event. The rest of the rule stays the same. However, the transformation result of the superimposed rule affects the application context of the rules of the original model transformation. This has to be addressed by some additional rules in the superimposed model transformations which we do not consider here.

Chapter 5

Grammar Based Approach to the Development of Model Transformations

In Section 4.1.3 we have shown that the meta-model based approach to model transformation development does not always provide the best way to implement certain transformation logic. In the cases when this transformation logic requires mappings between the structures within the source and target languages (see requirements 3, 10 – 13 in Table 3.1), meta-models may fail to provide an explicit description of these structures. Although this issue can be partially solved by introducing structuring classes (see Section 2.2), their expressiveness is inferior to the structuring possibilities provided by a *grammar based* method for the language definition [ASU86].

In this Chapter, we propose a model transformation development approach that utilizes the possibilities provided by the grammar based language description method. Subsequently, we refer to a model transformation developed using our approach as a *Grammar Based Model Transformation (GBMT)*.

Below, we first discuss grammar based definition of modeling languages in Section 5.1. Then, we provide a description of our approach. First, on a conceptual level in Section 5.2, and later, on a formal level in Section 5.3. Afterward, in Section 5.4, we discuss the transformation quality properties – *termination*, *soundness*, *completeness*, and *determinism* (see Section 4.2) – and prove that they are guaranteed by our approach. In Section 5.5, we describe the tool support for our approach. Finally, we give an overview of the most closely related approaches existing in the field (Section 5.6) and evaluate our approach in comparison with them (Section 5.7).

5.1 Description of Source and Target Languages

In contrast to the meta-models in MDSD (see Section 2.1), traditional approaches to the language definition (and translation) in the area of compiler construction [ASU86] define languages by *grammars*. There, a grammar is often given in an *Extended Backus-Naur Form (EBNF)* [ISO01].

Initially, grammars were proposed to define string-based languages, and later they were extended to define graph-based ones, like our source and target modeling languages. Since the main concepts of the grammar based language definition are the same for the string and the graph-based languages, we first introduce them for the former in Section 5.1.1 and show how they can be lifted to the latter in Section 5.1.2.

5.1.1 String Grammars

In its original usage, a *string grammar* defines a language of strings via a set of generative rules called *productions*. These productions define the structures (here sub-strings) allowed in the language and, thereby, form so-called *well-formedness* constraints [BBG05] similar to those defined using OCL in Section 3.1. The productions are then used to transform a single initial (*start*) symbol into a number of derived strings. The information about the productions used to transform the start symbol into some string forms a so-called *derivation* of this string, that represents its *structure* in that grammar.

Now, we give a formal definition of the central concepts of string grammars based on [ASU86] starting with the grammar itself and its language¹.

Definition 5.1.1.1 (Grammar). A grammar $G = (N, \Sigma, P, S)$ consists of a set of non-terminal symbols N , a set of terminal symbols Σ , a set of productions P , and a designated start symbol $S \in N$. Each production $p \in P$ is of the form $p = (l, r)$, with $l \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$, i.e. a string of symbols with at least one non-terminal, and $r \in (\Sigma \cup N)^*$.

A grammar $G = (N, \Sigma, P, S)$ is called context-free iff every production $p \in P$ has the form $p = (n, r)$ with $n \in N$.

The name context-free is inspired by the fact that the productions in a context-free grammar have no symbols in their left-hand side l , that just define the context and are not replaced by the production.

By applying grammar productions one derives new strings from the given ones (with non-terminals) by a process called *rewriting*. A string $s \in (N \cup \Sigma)^*$ is rewritten into a new string by, for instance, a context-free production $(n, r) \in P$ by finding n in s and replacing it with r . This way, a grammar defines the set of strings that can be *derived* from its start symbol S , and this set is called the *language* of the grammar G .

Definition 5.1.1.2 (Language). Let $G = (N, \Sigma, P, S)$ be a grammar. A sentence of G is a string $s \in \Sigma^*$ of terminal symbols that can be derived from S by using a finite sequence of productions from P . The language $\mathcal{L}(G)$ of G is defined as follows:

$$\mathcal{L}(G) = \{s \in \Sigma^* \mid s \text{ is sentence of } G\}$$

¹From now on, we refer to a string grammar simply as a grammar.

Based on this definition, only strings of terminal symbols can be in a language of a grammar, and these symbols have to be combined in the way defined by the grammar productions. Thus, the set of terminal symbols T of a grammar represents all element types allowed in the language and the set of productions P represent the static well-formedness constraints of the language. Compared to the meta-model based language definition (see Section 2.2) that uses declarative (descriptive) constraints combined with OCL, the grammar based definition technique uses an operational (constructive), production based, and structure-oriented way.

When defining a language using a grammar, we need to ensure that the *membership problem* for this type of grammar, i.e. the ability to identify if a sentence belongs to the language, has to be decidable. Otherwise, we can not distinguish the correct sentences of the language from the rest in an effective way.

The advantage of context-free grammars is that the membership problem for them, that is solved through *parsing* a sentence in the grammar, can be solved efficiently in polynomial time [ASU86]. During the parsing process, a combination of productions used to derive the parsed sentence from the start symbol S is sought. If it is found, the sentence belongs to the language of this grammar, otherwise, it does not. Furthermore, parsing a sentence using a context-free grammar gives us, if the sentence belongs to the grammar's language, a derivation in tree form called a *derivation tree*. Such tree structure clearly shows the applied productions starting with the start symbol and leading to the derived (parsed) sentence.

A derivation tree for a sentence defines its *structure* and confirms its language membership in $L(G)$. The inner nodes of such a tree are labeled with non-terminals and productions that rewrite/consume them. The root is labeled with a production that consumes the start symbol of the grammar. Every leaf of the tree is labeled with a terminal symbol. Figure 5.1 shows an example derivation tree for the sentence $\sigma_2\sigma_1\sigma_2\sigma_3$ in the grammar $G_\sigma = \{\{S, n_1, n_2, n_3\}, \{\sigma_1, \sigma_2, \sigma_3\}, \{(S, n_2\sigma_1n_1), (n_1, n_3\sigma_3), (n_2, \sigma_2), (n_3, \sigma_2)\}, S\}$.

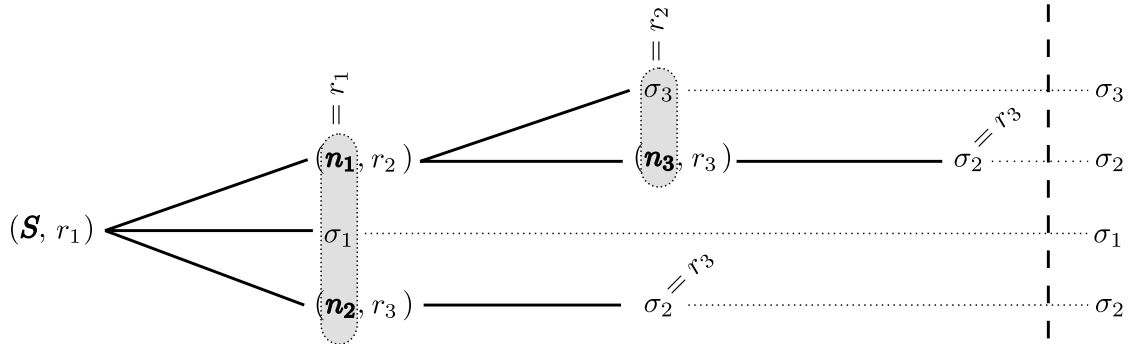


Figure 5.1: A derivation tree deriving $\sigma_2\sigma_1\sigma_2\sigma_3$ from S in G_σ

In general, there can be multiple, not structurally equivalent derivation trees for a sentence in a grammar, making this grammar *ambiguous*. Here, we only consider *unam-*

biguous grammars for defining our source and target modeling languages. The reason for this will become clear when we introduce our grammar based approach to the development of model transformations in Section 5.2. Formally, an unambiguous grammar is defined as follows.

Definition 5.1.1.3 (Unambiguous Grammar). *A grammar $G = (N, \Sigma, P, S)$ is called unambiguous iff for every sentence $s \in \mathcal{L}(G)$ its derivation from S is unique if performed “leftmost derivation first”.*

The “*leftmost derivation first*” strategy introduced in the above definition, represents the string rewriting process, where the left-most non-terminal in a string is always rewritten first.

Since, the languages we consider here are, like most languages in the MDSD context (see Section 2.1), graph-based, the discussed string grammars are insufficient to define such languages. We require a context-free grammar that produces graphs as sentences instead of strings. This type of grammar is called a *graph grammar*. Since there are multiple sub-types of graph grammars [Roz97], we have chosen one of its context-free sub-types to define our source and target modeling languages. Next, we introduce this sub-type and show how it can be used to define our example languages from Chapter 3.

5.1.2 Graph Grammars

To define our modeling languages, we have chosen a context-free sub-type of graph grammars called *Hyperedge Replacement Graph Grammars (HR grammars)* [Roz97].

HR grammars operate on *hypergraphs*, a generalization of graphs where edges, called *hyperedges*, can have more than two end points. These end points are called *attachment points* and their number is the *arity* of a hyperedge. Each attachment point has a corresponding *attachment node* within the hypergraph. All hyperedges are labeled and play a role of non-terminal or terminal symbols in the grammar, depending on the label.

In order to replace a hyperedge in a host hypergraph by a sub-graph during the rewriting, one has to specify how this new sub-graph is to be embedded in the remaining host hypergraph. To achieve this in HR grammars, the replacing sub-graphs are equipped with pairwise distinct *external nodes*. The rewriting then proceeds by replacing the hyperedge with the sub-graph and gluing together each external node of the sub-graph with the attachment node of the corresponding attachment point of the replaced hyperedge. The correspondences between the attachment points of a hyperedge and the external nodes of the replacing hypergraph (sub-graph) are defined in HR productions via mappings.

Overall, a *hyperedge replacement production* $l_1 := H$ has the following three parts:

- A single n -ary non-terminal hyperedge² l_1 ,
- A replacing hypergraph H with $k \geq n$ external nodes, and
- An (injective) *mapping* g of the k external nodes of H onto the n attachment points of l_1 .

Unlike the string grammar productions, graph replacement productions need to explicitly define how the new graph is attached to the remaining context, and, in HR grammars, this is done by defining such mapping g .

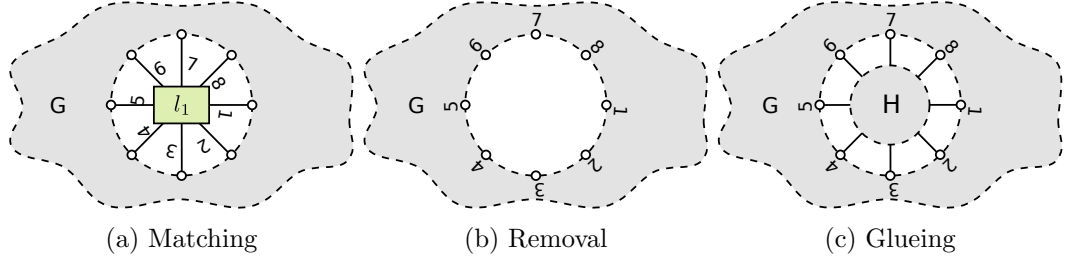


Figure 5.2: Sketch of production application in HR grammars

Figure 5.2 shows a sketch of the rewriting process (for $k = n$) that we have described. First, the hyperedge l_1 is found by *matching* (a) in the host hypergraph G and *removed* (b), then the hypergraph H is inserted (c) by gluing all its external nodes with the attachment points of l_1 according to the mapping g . Here, the mapping g maps an external node onto an attachment point with the same number.

At this point, we refrain from giving a more formal definition of HR grammars and related concepts (see Section 5.3 for details) and only mention that all definitions of string grammars carry over to HR grammars. An HR grammar has the same structure as a string grammar and thus consists of non-terminals (hyperedge labels), terminals (hyperedge labels), productions, and a start symbol (hyperedge label); its language is a set of hypergraphs. Furthermore, the membership problem for HR grammars is decidable ([Roz97], p. 99), which guarantees termination of the parsing process and the existence of derivation trees for the hypergraphs (models) from an HR grammar language. Under some conditions on the used HR grammar and/or input graphs [Lau90, CAB⁺13], and with some parser combinations [MM08], such parsing can also be made more efficient.

Since, any hypergraph can be transformed into a graph and vice-versa, we can use HR grammars to describe graph based modeling languages. Next, we use them to define both modeling languages of our transformation case study from Chapter 3.

As a set of terminal labels, we use classes and associations previously provided in the form of meta-models in Sections 3.1 and 3.2. This way, our source HR grammar has the terminal labels shown in Figure 3.7, while our target HR grammar has the ones from Figure 3.10. The non-terminal labels are chosen based on the high-level syntactical

²For brevity, we refer to a hyperedge with a non-terminal label as a *non-terminal hyperedge*.

structures, that exist in the modeling language that is to be defined. In our source language ADR it is e.g., an activity block B and in our target language CSP it is e.g., a process expression PE .

The number and the types of attachment points of a terminal hyperedge is determined by its label (type), more precisely, its association ends in the meta-model. In the case of non-terminal hyperedges, these attachment points are determined through the productions and, more precisely, the types of link ends they may have in the replacing sub-graphs.

For our ADR language, we defined the set of HR grammar productions shown in Figure 5.3. The productions are given in abstract syntax, in the form $l := H$, using bars to distinguish the different right-hand sides of the productions. Terminal and non-terminal hyperedges are depicted by the continuous and the dashed lines, respectively. Thereby, terminals for the class instances and non-terminals are depicted as rectangles.

Each mapping between the attachment points of a hyperedge l and the external nodes of a hypergraph H is depicted by using the same numbers, the former are given in diamonds and the later in circles. Thereby, as we see on the Figure, multiple external nodes can be mapped onto one attachment point, and, not all external nodes have to be connected in the hypergraph H .

The HR grammar in Figure 5.3 describes a language of well-formed ADR models containing zero or more activities, represented by non-terminals of type A , and passive resources (productions 1 – 4). These activities have an activity description, represented by non-terminal hyperedges of type AD , and an entry point attribute (productions 5,6). An activity description has exactly one *initial* and *final* node, each of which is connected either directly by an edge or via an activity block hyperedge of type B (productions 7,8). Further productions of this HR grammar define the following kinds of activity blocks a non-terminal of type B can become: a *block sequence* (production 9), a *fork/join block* (production 10), a *decision/merge block* (production 11), a *loop* (production 12), a *call action* (production 13), an *internal action* (production 14), or a resource *acquire* or *release* action (productions 15 – 16).

Note that, for instance, all elements of a decision/merge activity block are only created all together connected in a structure by one production. All other described kinds of activity blocks are also produced in this systematic way.

As for the attachment points of the non-terminal hyperedges of types M , A , AD , and B of the presented grammar, they are derived based on the quantity and types of attachment points they are connected to in the right-hand side hypergraphs of the productions of this HR grammar. The types themselves and the constraints on the number of attachment points are taken from the meta-model from Section 3.1, that we use to label terminals and non-terminals of this grammar. For instance, a non-terminal of type B has four attachment points – *nodes*, *edges*, *source*, and *target*. They are derived from the attachment nodes, that a hyperedge of type B is connected to in the right-hand sides of the productions. The external nodes of the right-hand side of a production in

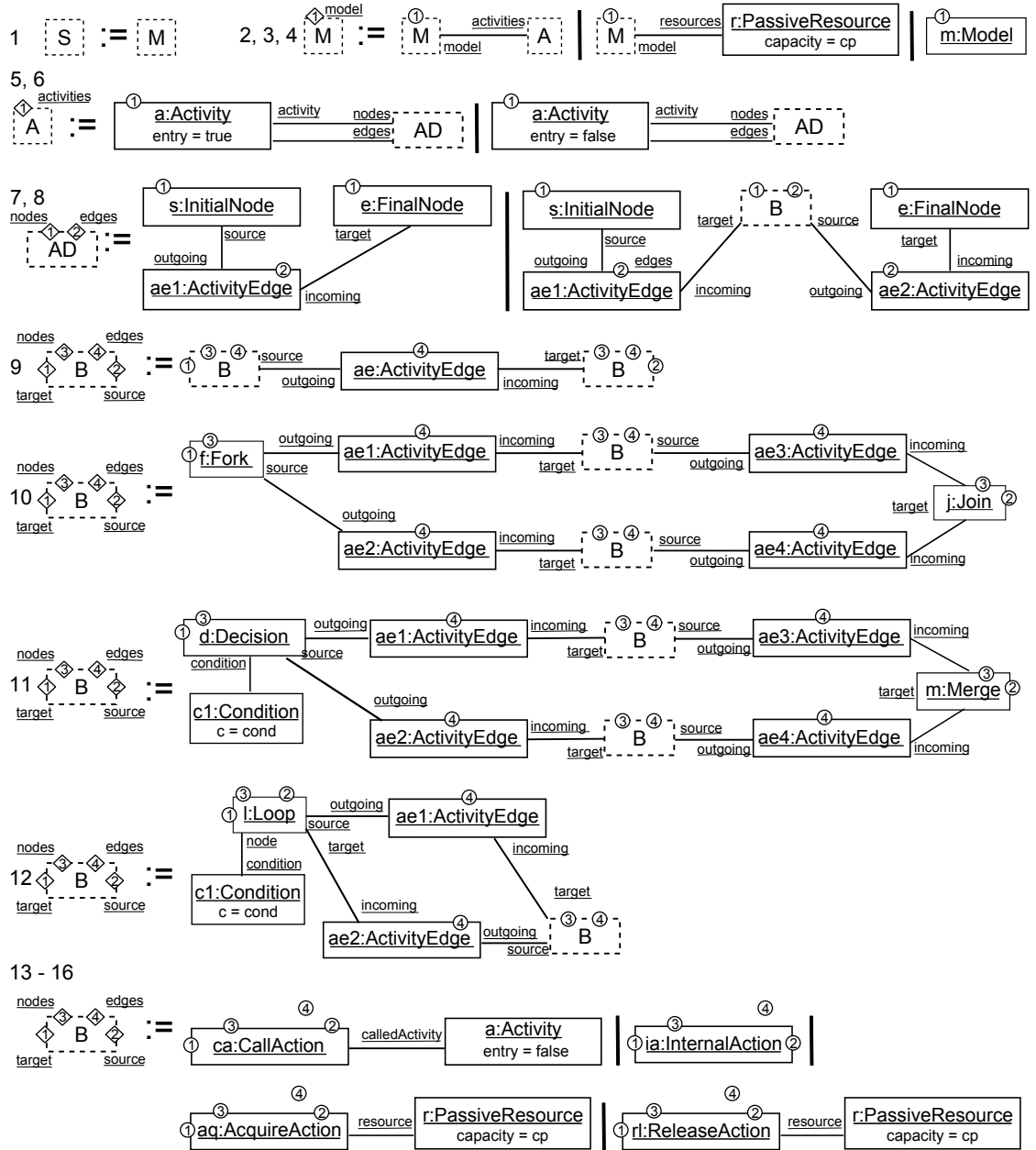


Figure 5.3: HR grammar productions for the ADR modeling language (abstract syntax)

this grammar are then defined accordingly, in order to connect its hyperedges to the attachment points of the replaced hyperedge l in the correct way.

Finally, since most of our ADR model elements have a concrete syntax representation, in Figure 5.4 we show a more intuitive representation of some of the productions of the

defined HR grammar. For simplification, we hide the details of attributes, attachment points, and external nodes here.

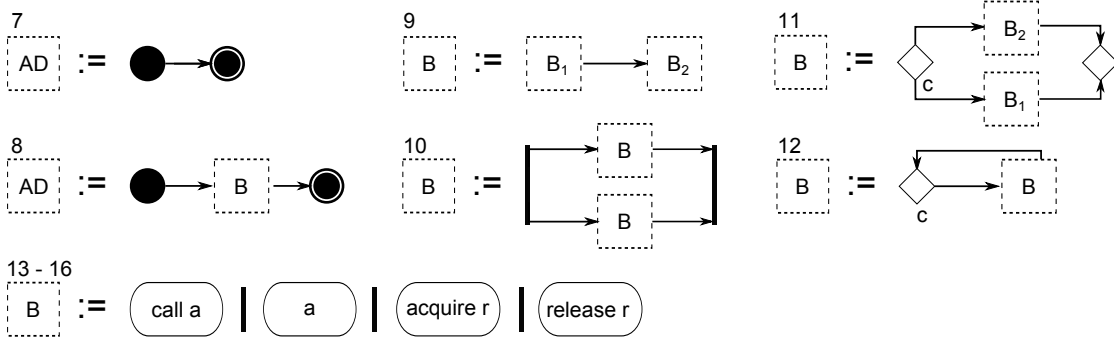


Figure 5.4: Some productions of the HR grammar from Figure 5.3 in concrete syntax

An HR grammar we use to define our target modeling language CSP introduced in Section 3.2, is specified using the same principle as the ADR HR grammar. We use the CSP meta-model from Section 3.2 for the grammar typing/labeling.

Figure 5.5 shows this HR grammar for CSP using the notation conventions introduced in Figure 5.3. A *model* described by this grammar and denoted by non-terminal of type M can contain a set of *processes*, denoted by non-terminals of type P (productions 1 – 3). A process can be described by a *process expression*, represented by a non-terminal of type PE (production 4). A process expression can be one of the following kinds: a *sequential* process composition, an *external choice*, or a *parallel* process composition with an alphabet, represented by a non-terminal hyperedge of type ES (productions 7 – 9); a *conditional* process composition (production 10); an *event prefix* followed by another process expression hyperedge of type PE with the different event parameter directions (productions 11, 12); an another *process* (production 6), or an empty process *SKIP* (production 5). The events in the synchronization alphabet of a parallel composition are represented by non-terminals of type E and created by productions 13 and 14.

Based on this grammar, a CSP model can also have an arbitrary number of *channels*, *value sets*, and *event sets*, represented by non-terminal hyperedges of types CH , VS , and ES respectively (productions 13 – 15). The value sets then define the ranges of the channels (production 16) through multiple values, represented by non-terminals of type V (productions 17, 19, 23). The event sets define the synchronization alphabet of parallel compositions (production 9) as sets of events with output parameters, represented by hyperedges of type E (productions 20, 22, 24). Finally, both value and event sets can be combined from sub-sets (productions 18, 21).

The above HR grammars, that we have defined for our source and target modeling languages, produce the models that only contain the (terminal) hyperedges of the types defined in the corresponding meta-models. Furthermore, the right-hand side of each production (H), more precisely, its terminals are only linked in the way that does not

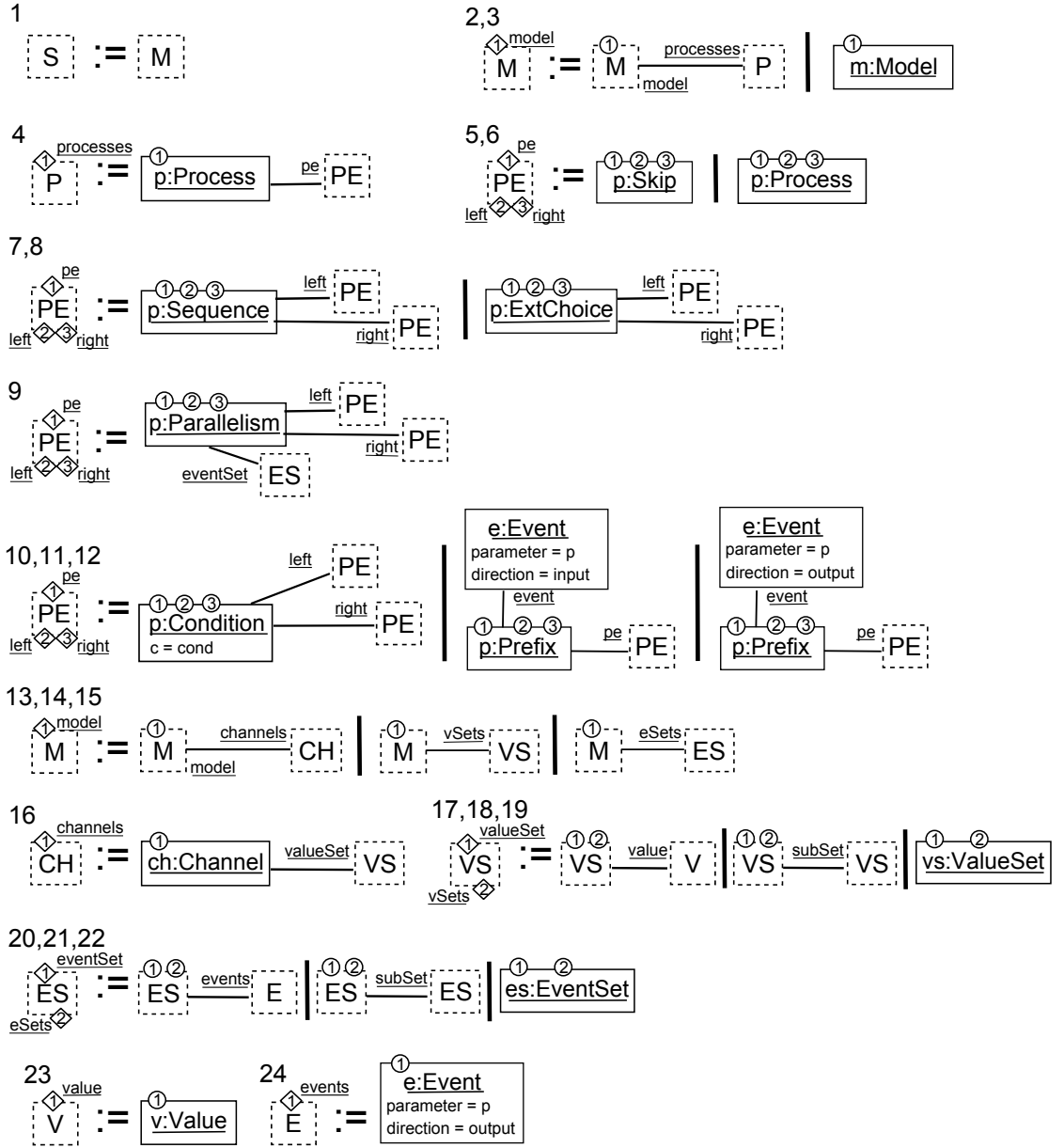


Figure 5.5: HR grammar productions for the CSP language (abstract syntax)

violate the meta-model constraints. For these example languages and their HR grammars, we can see that the HR grammars' languages are sub-sets of the languages allowed by the corresponding meta-models.

In general, however, this language inclusion property must be guaranteed in each case, when we define an HR grammar that should only produce models (hypergraphs) allowed by a given meta-model. Unfortunately, this can not always be guaranteed, since such

language inclusion check is, in general, undecidable [AN99]. In some cases, it might be possible for the grammar designer to perform changes to it to make such a check straightforward. Alternatively, some approaches [WK06, WTEK08, HM11] proposed for generating a language-equivalent graph grammar for a given meta-model (some with OCL), can be used to avoid this check. They can be adapted and combined with additional productions re-factoring to introduce non-terminals for complex language structures.

Now, that we have introduced the grammar based method for the language definition and shown its application to define our source and target languages, we can discuss our grammar based approach to the definition of model transformations. We start with its conceptual description and demonstration on our example ADR-to-CSP transformation from Section 3.3.

5.2 Conceptual Description of the Approach

The main goal of our transformation development approach is to allow an intuitive transformation definition by mapping not only single elements but also high-level syntactical structures in the source and target languages onto each other. In terms of grammars, this means relating (or *pairing*) source and target productions, that create the corresponding structures.

The approach that we propose here uses the context-free HR grammar based language definition discussed in Section 5.1.2. It extends an idea proposed by Pratt in [Pra71]. Pratt defines *pair grammars* as a way of relating the grammars of two languages, thus, obtaining a natural way of relating language structures and building translations from one language to the other. Our approach brings the idea of pair grammar based translation to the world of MDSD and model transformations, lifting it to the level of abstract syntax while preserving its advantages. We also extend it in order to cover a broader variety of model transformations.

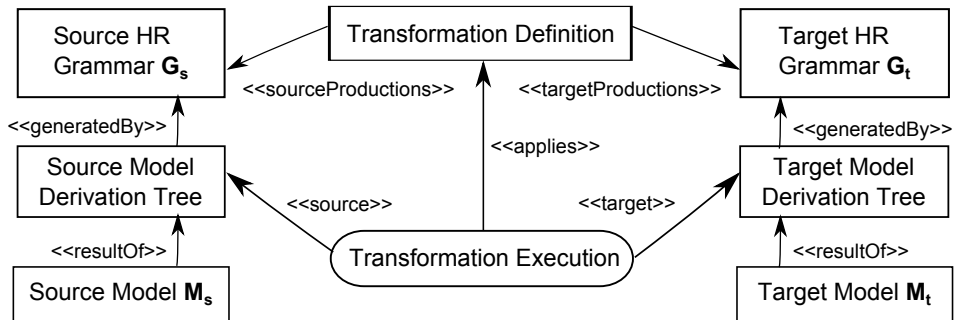


Figure 5.6: Overview of our grammar based approach to transformation development

Figure 5.6 gives an overview of our *Grammar Based Model Transformation (GBMT)* development approach. The transformation mappings – like in pair grammars – are

defined as a relation of productions of the source HR grammar G_s with those of the target HR grammar G_t . For our example transformation, we would use the HR grammars defined in Section 5.1.2.

The source and target models M_s and M_t are described in abstract syntax as typed hypergraphs. The defined grammar based transformation is then executed as follows: given a valid input source model $M_s \in L(G_s)$, its derivation tree in the source grammar G_s is obtained by parsing and used by the transformation to apply the mappings and produce a derivation tree in the target grammar G_t , and, thereby, the corresponding valid output target model $M_t \in L(G_t)$. If an input source model M_s is not valid i.e., is not in the language of the source HR grammar $M_s \notin L(G_s)$, the transformation is not executed, and *non-transformable* is returned as a result.

Next, we describe in details how the grammar based transformation mappings are defined (Section 5.2.1) and executed (Section 5.2.2) using our approach. We demonstrate its application by creating a GBMT that fulfills our ADR-to-CSP transformation requirements from Section 3.3.

5.2.1 Transformation Definition

In its *basic form*, a *grammar based model transformation rule (GBMZ rule)* relates two productions – one production p_s from the source and one production p_t from the target grammar. Additionally, it defines a 1-to-1 correspondence of some type C between the left-hand side non-terminals of these productions. This correspondence is required in order for the rule to be applied at the correct place (a target non-terminal hyperedge) and, thus, it is later referred to as a *correspondence type pre-condition* relation.

The non-terminal hyperedges on the right-hand sides of the related productions are also related via correspondences, such that, each non-terminal hyperedge created by the production p_t has exactly one correspondence to a non-terminal hyperedge created by the production p_s . This requirement ensures that each non-terminal hyperedge created in the target derivation tree, constructed during the transformation execution, has a related non-terminal hyperedge in the source derivation tree. Once that source non-terminal hyperedge is rewritten by some source production, which happens eventually for a valid input model $M_s \in L(G_s)$, the related target non-terminal hyperedge is also rewritten by the related target production. This, of course, requires the existence of an applicable rule in the model transformation definition, that is ensured by a set of conditions put on GBMTs considered later.

Figure 5.7 shows a sample GBMT rule relating the source production for a sequence of activity blocks to the target production for a sequence of process expressions³. It states that, when a block sequence is replacing a non-terminal hyperedge of type B (block) in a source derivation tree, a sequence of process expressions should replace the

³To simplify, we show all GBMT rules without the HR grammar details.

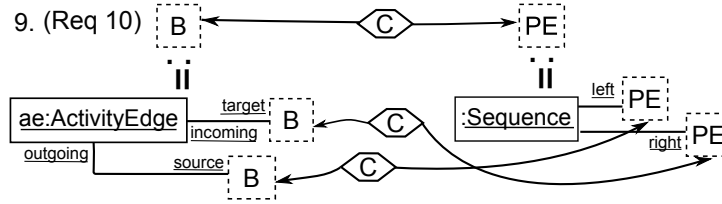


Figure 5.7: GBMT rule: Activity block sequence to process expression sequence

corresponding (correspondence relation of type C) non-terminal hyperedge of type PE (process expression) in the constructed target derivation tree.

In addition, the considered transformation rule links the source right-hand side non-terminal hyperedges of type B and the target right-hand side non-terminal hyperedges of type PE via two 1-to-1 correspondences of type C . This way, we transfer the order between the elements to CSP. These correspondences are to be created between the block sequence hyperedges in the source derivation tree and the newly created process expressions hyperedges in the target derivation tree. The notion of correspondences that we use is inspired by the correspondence links in TGGs (see TGG rules in Section 4.1.3).

Figure 5.8 shows some further rules of the GBMT definition for our example ADR-to-CSP transformation. They relate productions in the following way:

- Rule 1: Productions for non-terminal hyperedges of type M , representing *models* in the source and target grammars, are related. Their left-hand side non-terminals are connected via a correspondence of a special *default* type C_d , used for relations between the start symbols of the grammars. Their right-hand side non-terminal hyperedges are connected via a correspondence of type C . In this GBMT example we only use the above two types of correspondences.
- Rule 2: The production for a non-terminal hyperedge of type A , representing an *activity*, in the source grammar is related to the production for a non-terminal hyperedge of type P , representing a *process*, in the target grammar. The left-hand side non-terminals here are connected via a correspondence of type C , and so are the right-hand side non-terminal hyperedges re-created for models (of type M), and the hyperedges for an activity and a process are connected as well.
- Rule 7: The production of a terminal structure, representing an *empty activity* description, is related to the production of a terminal hyperedge, representing an empty *SKIP* process. Their left-hand side non-terminals are connected via a correspondence of type C . Since no non-terminal hyperedges are created by the related productions, we do not have any correspondences between the right-hand sides in this rule. This rule addresses a part of the requirement 4 from Table 3.1. Although it creates a new element in the process definition, this element (*SKIP*) is semantically equivalent to a direct transition between the events (created by other transformation rules).

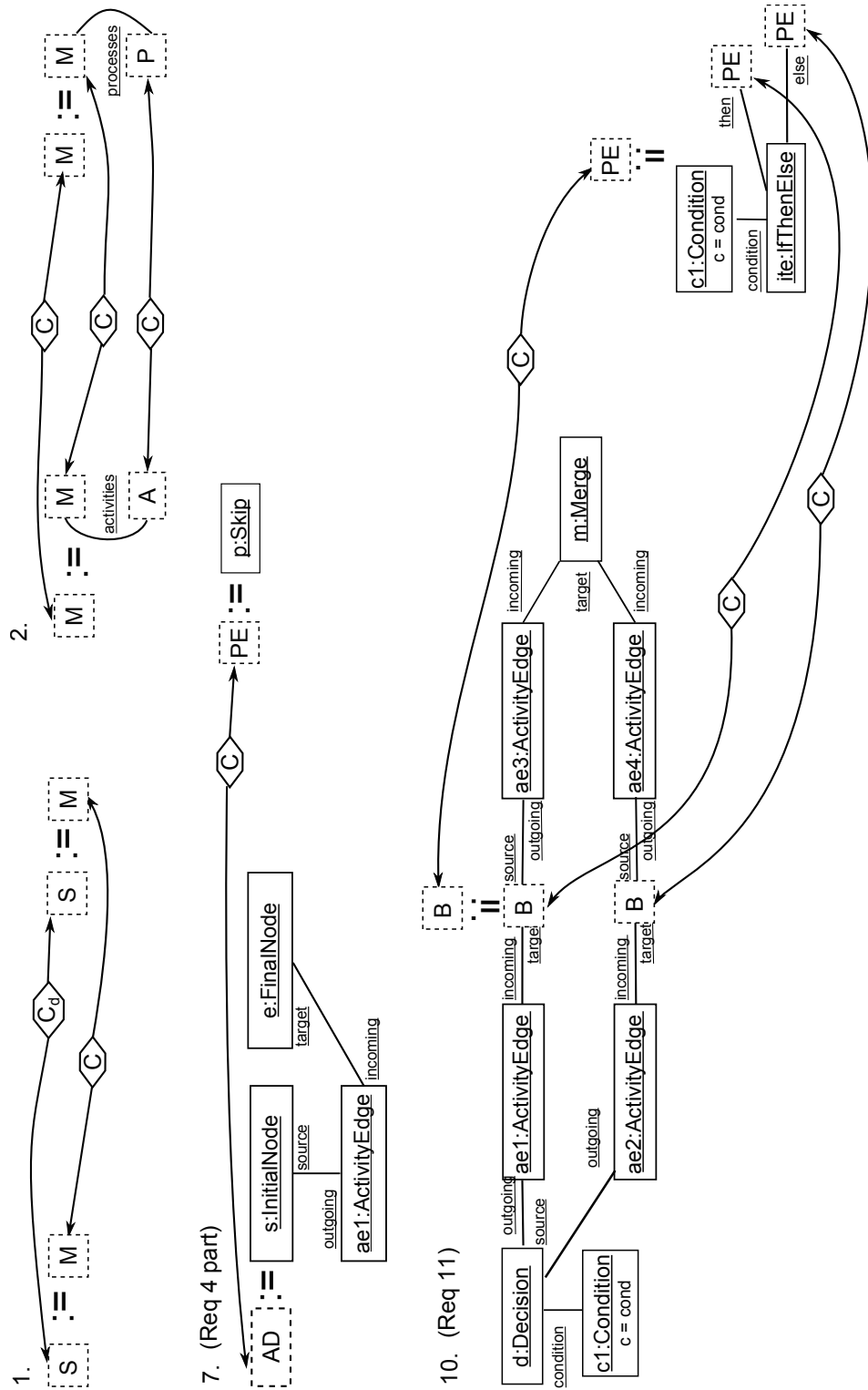


Figure 5.8: Further GBMT rules for the ADR-to-CSP transformation

- Rule 10: The production of a *decision/merge* block structure, consisting of terminal and non-terminal hyperedges, is related to the production of an *if-then-else* process expression. As in the previous rules, the left-hand side non-terminals are connected via a correspondence of type C . The right-hand side non-terminal hyperedges are also connected via correspondences of type C based on their location in the structures – the block on the top branch, where the condition c holds, is linked to the *then* sub-expression of the *if-then-else* process expression, and the block on the branch, where it does not, is linked to the *else* sub-expression, respectively.

Based on the numbers of the rules, one can see that this list is incomplete. We consider further rules for our example later. Before that, we summarize the conditions we put on the form of basic transformation rules and GBMTs consisting of them, in an example independent form.

In general, if a GBMT definition between two HR grammars G_s and G_t has the following form:

1. In each rule:
 - a) A source production $p_s = (n_s, r_s)$ is related to target production $p_t = (n_t, r_t)$,
 - b) The left-hand side non-terminals n_s and n_t are linked via a correspondence (start non-terminals S_s, S_t are always linked by a correspondence of type C_d),
 - c) Each non-terminal hyperedge in r_t has exactly one corresponding non-terminal hyperedge in r_s ;
2. For each source production $p_s = (n_a, r_a)$ in P_s and each type of correspondence between the hyperedges of type n_a and n_b in some rule (or the initial S_s to S_t correspondence of type C_d), there is exactly one GBMT rule relating p_s to a target production p_t , where $p_t = (n_b, r_t)$, to cover all types of the corresponding pairs (n_a, n_b) ;

and if G_s is unambiguous, then the resulting transformation has some important quality properties, which we show in Section 5.4.

What if a 1-to-1 relation of productions in the described basic GBMT rules is not enough to describe the desired transformation logic? For example, when one source production application should trigger a longer sequence of productions in the target grammar, or vice versa. Our requirements on the example ADR-to-CSP transformation in Table 3.1 provide several examples of such transformation logic, starting with the requirement 1. In order to be able to express GBMT rules to address such complex requirements, we now consider an *extended* type of GBMT rules. This type of rules allows *n-to-m relations* between the productions of the source and the target grammar.

To specify an extended rule, we choose the derivations we want to relate – a sequence of n productions from the source grammar and a sequence of m productions from the target grammar, where $n, m \geq 1$. The same production can appear in such sequence multiple times. In some cases, it is clear from the order and the left-hand sides of the

productions in the sequence, how they are connected into a derivation. If this is not clear, additional *links* must be specified between the non-terminal hyperedges, produced by the previous productions, and the later productions that replace them. Since, HR grammars that we use for our transformation definition are context-free (see Section 5.1.2), we use derivation sub-trees to describe such links.

As for the conditions that we imposed on our GBMT in order for it to have some desired properties (see Section 5.4), they now need to be adjusted for this extension. Here, we only provide an idea of the required adjustments and give their full definition and explanation during the formalization in Section 5.3. First, for multiple properties, when a rule relates two derivation sub-trees, all non-terminal leafs of the target sub-tree have to have the corresponding non-terminal leafs in the source sub-tree. Next, for determinism, the absence of two simultaneously applicable GBMT rules need to be guaranteed either by disallowing rules with the same correspondence and one source sub-tree being a sub-tree of the other, or by introducing rule priorities. Furthermore, for completeness, the presence of rules covering all possible source model derivation steps must be guaranteed either via the coverage analysis, or by forbidding the use of source sub-trees with more than one production.

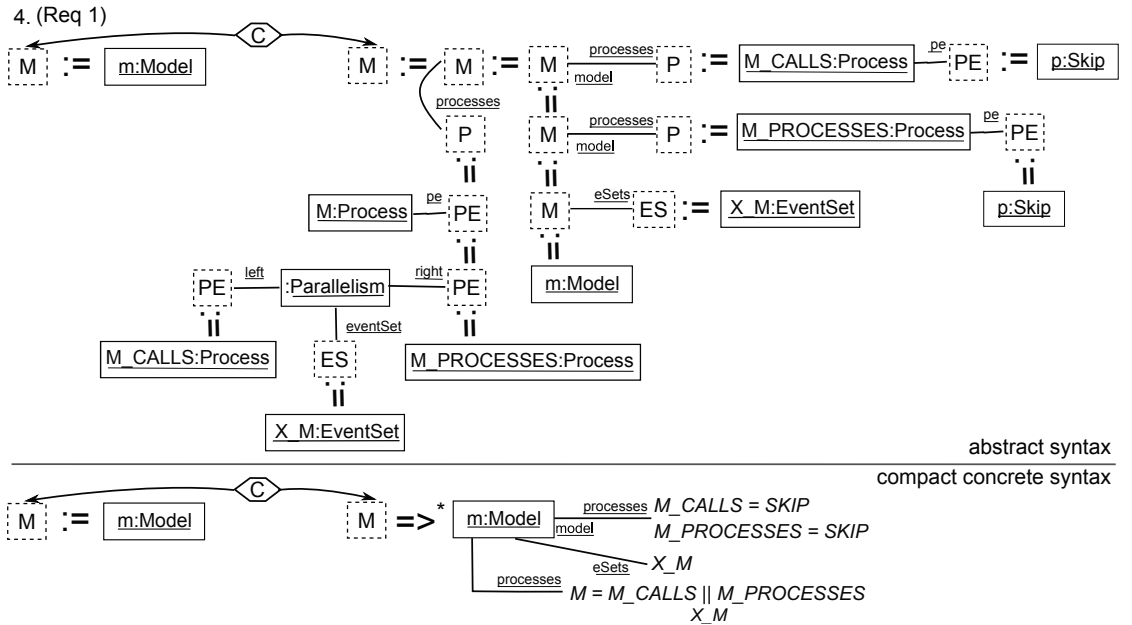


Figure 5.9: GBMT rule: ADR model to a set of processes and an event set

Now, we consider further GBMT rules for our ADR-to-CSP transformation example. We still have to describe several transformation rules to complete its definition. We start with the GBMT rule for requirement 1 from Table 3.1.

Figure 5.9 shows an extended GBMT rule for requirement 1 in abstract syntax (top) and concrete syntax (bottom). The latter, more compact, representation of the rule also

omits the intermediate replacements and only shows the derivation results, and it uses concrete syntax (in *italic*) for the terminal CSP elements, when available.

The rule relates the production of a terminal hyperedge representing an ADR model to a derivation sub-tree combining multiple target productions to create the processes M , M_CALLS , $M_PROCESSES$, and the event set X_M according to the requirement. If, for instance, one production for the *SKIP* process would not exist in this rule, one non-terminal hyperedge PE would be a leaf. However, it could not have any corresponding source non-terminal hyperedges which are leaves, since the right-hand side of the source production only contains one terminal hyperedge. This would violate the conditions on an extended GBMT rule, that we have briefly discussed before. With all the *SKIP* productions present, all non-terminal hyperedges created by the target productions, have been consumed in the target sub-tree. So we only need a correspondence between the left-hand side non-terminals of the first productions to comply with the conditions.

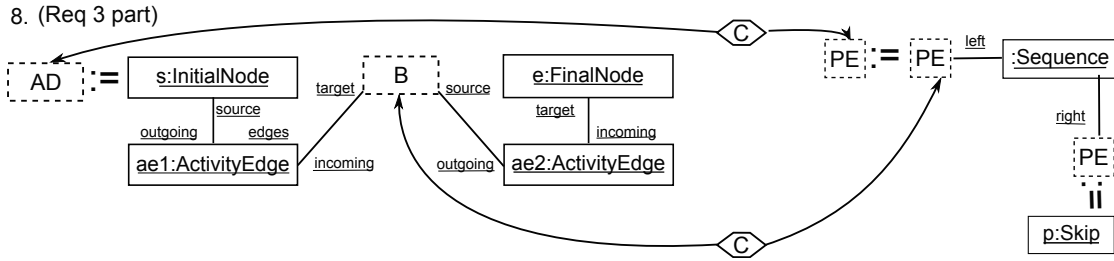


Figure 5.10: GBMT rule: Non-empty activity to a processes sub-expression

Next, we consider an extended GBMT rule for a part of requirement 3. This rule, shown in Figure 5.10, relates the production for a non-empty activity description structure, containing a block hyperedge of type B , to a sub-tree, combining two target productions to create a process sequence with a sub-expression hyperedge of type PE followed by a terminal *SKIP* process. The left-hand sides of the first productions are linked via a correspondence of type C . The block hyperedge is linked to the sub-expression hyperedge via a correspondence of the same type. This rule addresses a part of the requirement 3 by creating the sub-expression PE . Since our CSP HR grammar does not have a production to (re-)create a single sub-expression, we use a semantically equivalent sequential composition with a *SKIP* process to achieve this.

Next, we consider an extended GBMT rule for requirement 12 that is shown in Figure 5.11. It relates the production for a fork/join structure with two sub-blocks, represented by hyperedges of type B , to a sub-tree combining two target productions to create a parallel composition of two sub-expressions, represented by the hyperedges of type PE . Here, we only have to use an extended GBMT rule because we need to additionally assign an empty synchronization alphabet for this composition. The left-hand sides of the first productions are linked via a correspondence of type C . The block hyperedges and sub-expression hyperedges are also linked pairwise via this type of correspondence. Since

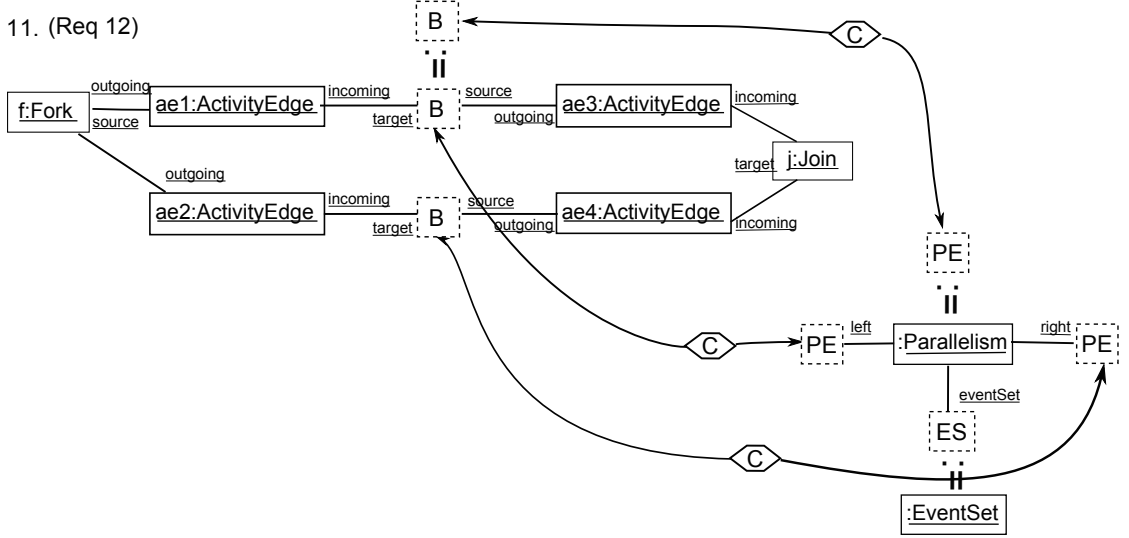


Figure 5.11: GBMT rule: Fork/join structure to a parallel process composition

we have a parallel process composition where the order is not important, the hyperedges can be connected in two different ways. We show one of them.

We consider one more extended GBMT rule for requirement 14, that is shown in Figure 5.12. Since its target sub-tree is relatively large and has very similar parts, we only show some parts of it in detail and use some named place-holders in square brackets for the rest. This rule relates the production of a passive resource to a target sub-tree combining multiple productions to create all the processes, channels, and value sets described in the requirement. Here, we show the complete sub-tree for the process $R(0)$ and hide the rest of its definition for the other parameter values, that is analogical. We also show a sub-tree for one of the channels and those for the value sets. Finally, we define a sub-tree that adds the process $R(cp)$ to the other processes of the model. As usual, the left-hand sides of the first productions in the rule are linked via a correspondence of type C . The right-hand side leaf model hyperedges are also linked via this type of correspondence.

Note that in the GBMT rule in Figure 5.12, in order to find the name of the model ($mname$), we had to use the link of the resource r to the model to access its *name* attribute ($r.model.name$). In our approach, we currently allow simple assignment of target attributes with the values of the source attributes, accessible within the rule. These source attributes can either be located in the source sub-tree of the rule directly, or be found via the navigation through the non-optional references of the elements in the source sub-tree, like in our example GBMT ($r.model.name$).

By now, we have already considered several GBMT rules for our ADR-to-CSP transformation. Some of them have a basic and some of them have an extended form. With the extended GBMT rules, we related multiple source and target productions that build

3. Req 14

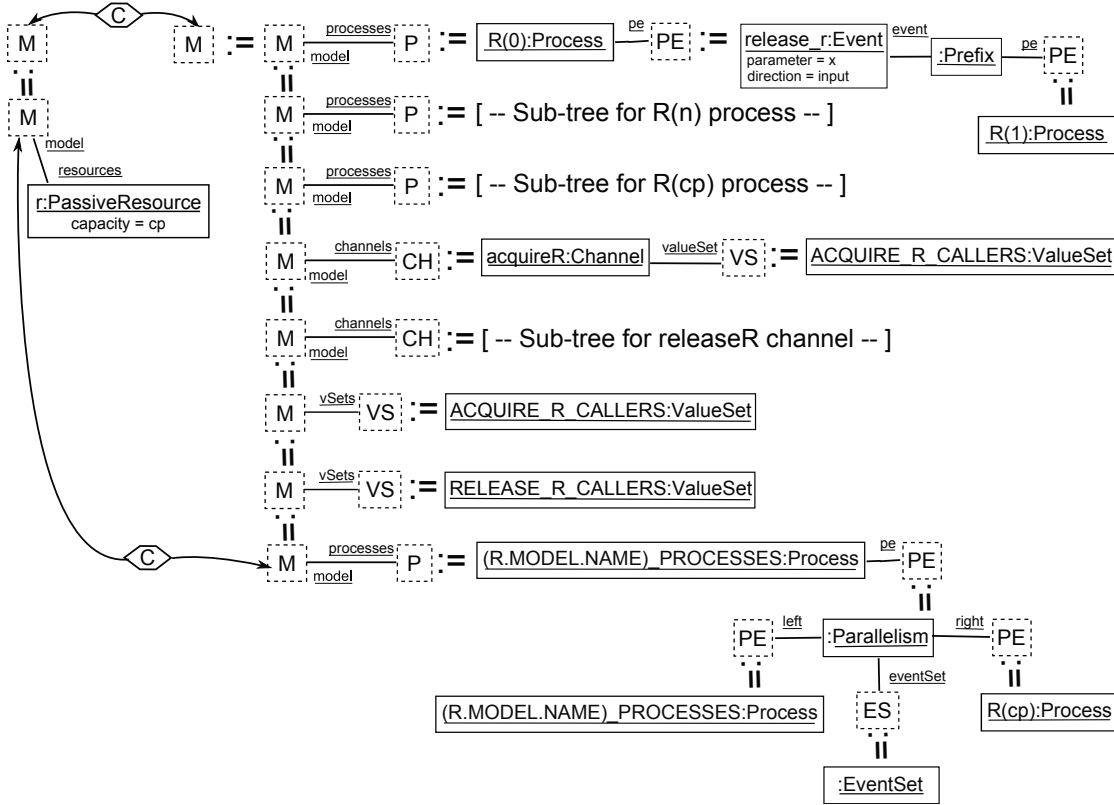


Figure 5.12: GBMT rule: Passive resource to a set of processes, channels, and value sets

derivation sub-trees. However, in some cases, the transformation under development requires some additional non-local behavior. This means that a transformation rule has to create additional target language structures somewhere else in the output target model.

In our ADR-to-CSP transformation requirements in Table 3.1, we can see the following examples of this non-local behavior: an activity transformation (requirement 2, 5); a call action, an internal action, an acquire/release action transformation (requirements 6 – 9); and a loop structure transformation (requirement 13). They require the transformation to create the following multiple structures: usually a part of some process expression and some structures, directly connected to the target model instance (e.g. channels, processes, etc.).

These are the requirements that we have not addressed yet by the previously considered GBMT rules. In order to do this, we introduce the second extension of our GBMT development approach – so called *non-local* GBMT rules. In terms of GBMT rules, such non-local behavior would mean connecting a sub-tree, that creates the additional structures, to some already constructed part (a non-terminal hyperedge) of the

target derivation tree *other* than the one, where the main target sub-tree of the rule is applied.

The non-local type of GBMT rules allows the use of an *additional correspondence type* between the source and the target non-terminals as a pre-condition, and uses its target non-terminal as the root of an *additional derivation sub-tree*, combining one or more productions. During the transformation execution, when a correspondence of the defined additional type is found and the rule can be applied, the defined additional sub-tree is applied to the target of this correspondence, thereby, replacing it.

Like in the previous extension, in the case of non-local GBMT rules, we need to impose new and adjust existing conditions on our GBMT, in order for it to have the desired properties (see Section 5.4). Here, we only provide an idea of some required adjustments and give their full definition and explanation in Section 5.3. First, the type of the main pre-condition correspondence can not be the same (same combination of its type and non-terminal types) as the type of the additional correspondence in a non-local GBMT rule. Furthermore, a non-local rule should propagate the additional correspondence that it requires for the application. This is done through a leaf hyperedge in the additional target sub-tree of a rule, that has to be identified with its root via an identity mapping. Here, in this Section, we only show (via dashed lines) this identity mapping in our example rules, but do not discuss it in detail.

To guarantee determinism in GBMTs with non-local rules, for each non-local rule, exactly one correspondence of the specified additional type must exist when the rest of the rule is applicable. Plus, the same correspondence condition as for the main target sub-tree must hold for the non-terminal leaf hyperedges of the additional target sub-tree (apart from the one with the identity mapping).

Next, we consider a non-local extended GBMT rule for requirement 13. We start with it due to its relative simplicity. This rule, shown in Figure 5.13, combines both of the extensions that we have introduced. It relates the source production for a loop structure to the target production for a referencing process $L(c)$, and uses an additional correspondence between two model non-terminals (M) to create the referenced recursive process $L(x)$. This process consists of an *if-then-else* expression with the first non-terminal hyperedge of type PE in the *then* branch linked to the body of the loop, represented by the hyperedge of type B , followed by a recursive call to itself, which terminates when the condition x is false. The new process $L(x)$ is connected directly to the target model hyperedge M , which makes the result of this GBMT rule *non-local* and different from the previous examples.

Next, we consider a non-local extended GBMT rule for the transformation of one of the action node types – internal action (requirement 7) – shown in Figure 5.14. It relates the production of an internal action to the production of an event prefix with an event with the same name followed by a *SKIP* process. Additionally, this rule specifies a non-local effect. It uses an additional correspondence between two model non-terminals (M)

12. (Req 13)

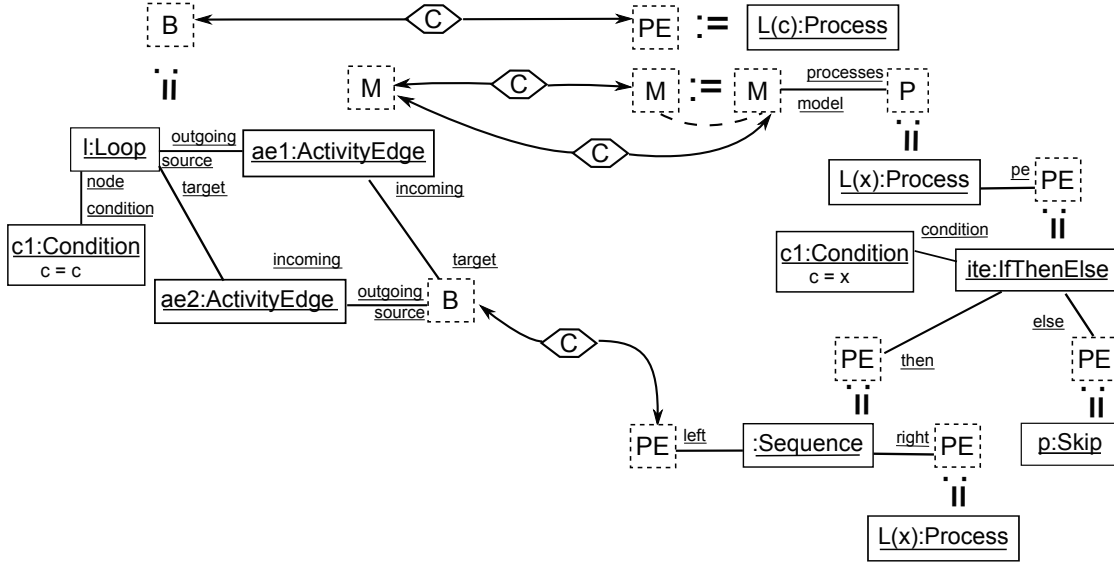


Figure 5.13: GBMT rule: Loop structure to an equivalent recursive process and a referencing process expression

13. Req 7

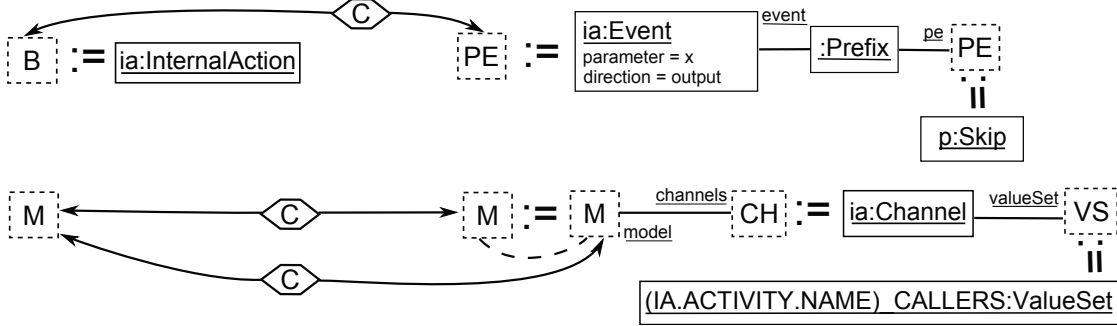


Figure 5.14: GBMT rule: Internal action to an event prefix and a channel

to create a channel according to the requirement. It accesses the name of the activity of the given internal action through its reference (*ia.activity.name*).

Since the GBMT rules for other types of action node – call, acquire, and release – are very similar to the one for an internal action and only produce some additional processes, channels, events, and sets in their main and/or additional target sub-trees, we do not consider them here. Their structure can be derived from Table 3.1.

The last non-local GBMT rules left to consider, are the ones for requirements 2 and 5 and the remaining parts of requirements 3 and 4. These rules (5 and 6) are shown in Figure 5.15 in the following compact form: we only show the left-hand sides of the first productions of the rules and the correspondence between them once, as they are the same

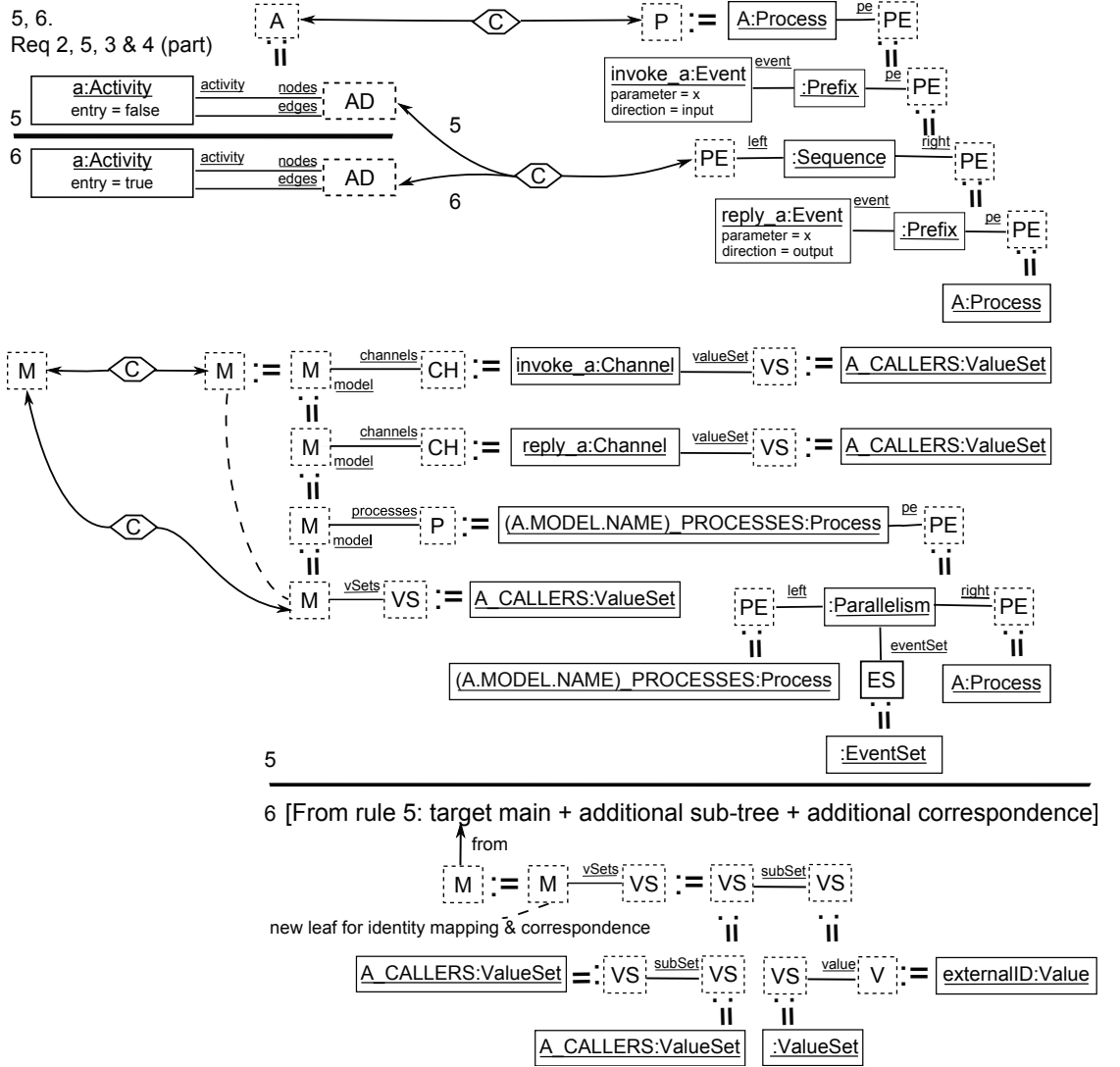


Figure 5.15: GBMT rules: Activity (entry) to a channel, process, and a value set (with additional value)

for both rules, and divide the source/target sub-trees of the rules via the horizontal lines. This compact representation is chosen, since these rules only differ in one attribute value in their source sub-tree's (*entry* attribute) and an extra part in the additional target sub-tree of rule 6.

Rule 5 is applicable when an activity is not an entry and rule 6 is applicable otherwise. Rule 5 relates the production of an activity containing a description, represented by a hyperedge of type *AD*, to the target sub-tree that combines productions to create the process with the name of the activity and its definition. This process contains two event prefixes with the *invoke* and *reply* events with the required parameters. The sub-process

expression for the first event prefix is represented by a non-terminal hyperedge of type *PE*. This produced non-terminal edge is linked via a correspondence of type *C* (labeled by 5) to the description non-terminal hyperedge of type *AD*. The sub-process expression for the second event prefix is the process itself.

Rule 5 also has a non-local part – it uses an additional correspondence between two model non-terminals (*M*) to create two channels, a process, and a value set according to requirement 2. Rule 6 has the same main target sub-tree and the same additional correspondence type (not shown) as rule 5, and its additional target sub-tree extends the additional target sub-tree of rule 5 by the productions to add an *externalID* value to the respective value set (see requirement 5).

We have described an idea for the definition of basic GBMT rules and discussed the following two extensions: extended rules and non-local rules, which can be used in combination. We have also provided and explained most of the GBMT rules for our ADR-to-CSP transformation. Next, we show how a GBMT is executed.

5.2.2 Transformation Execution

A GBMT is executed on an input source model in two steps. First, the input model is parsed to get its *leftmost derivation tree* – a derivation tree that represents its leftmost derivation. The concept of a leftmost derivation in HR grammars is analogous to the one for string grammars (see Definition 5.3.2.2 for the full formalization). Then, the GBMT rules are applied on this tree to construct the target derivation tree and, thereby, the target model.

During the execution of a GBMT, its transformation rules are used to identify which target production(s) are triggered for which source production(s) and correspondences. Given a derivation tree for the source model, we incrementally construct a derivation tree for the target model by applying the related target productions. Thereby, the correspondences between the non-terminal hyperedges in the source and target derivation trees help to keep track of the related model structures, represented by the non-terminal hyperedges.

In the first step of an execution, an input model is parsed with respect to the source HR grammar of the transformation. As an HR grammar based parsing is decidable [Roz97], for each input model we either get its leftmost derivation tree or a message that it is not parsable. If a model can not be parsed, it is not in the source language, and, hence, will be rejected by our GBMT returning *non-transformable*.

In the second step, we build a target derivation tree based on the obtained source model derivation tree and the *initial correspondence*. The initial correspondence is created between the source start graph with a single hyperedge of type S_s (start symbol of the source grammar G_s) from the source derivation tree and the target start graph with a single hyperedge of type S_t (start symbol of the target grammar G_t), which becomes

the first element of the target derivation tree. This correspondence is used as a basis for the transformation execution.

Next, we iteratively construct the target derivation tree for the given source model in the following way: we traverse the source derivation tree to find the next non-terminal hyperedge e_s and the source production p_s that rewrites it; then, we consider each correspondence c of this hyperedge e_s and find a GBMT rule tr , that relates some p_t with the source production p_s and where the left-hand side non-terminals have the types of the hyperedges linked by c and their correspondence has the type of c ; finally, if the found transformation rule tr has a basic form (no extensions, see Section 5.2.1), we apply it by rewriting the target non-terminal hyperedge, linked to e_s through c by the target production p_t of tr , and creating correspondences between the produced target non-terminal hyperedges and the non-terminal hyperedges from the source tree, according to the definition of tr .

If the found transformation rule tr is an extension of the basic rule form, its application contains several additional steps and/or pre-conditions. If tr relates derivation sub-trees, the source sub-tree with the root attached to e_s would need to be found in the source derivation tree, and the target sub-tree would then be applied to e_t . If tr is non-local, we need to ensure that a correspondence of its additional correspondence type has already been created during the previous execution steps. We address these requirements and the application differences among the types of rules in a formal way in Section 5.3.2.

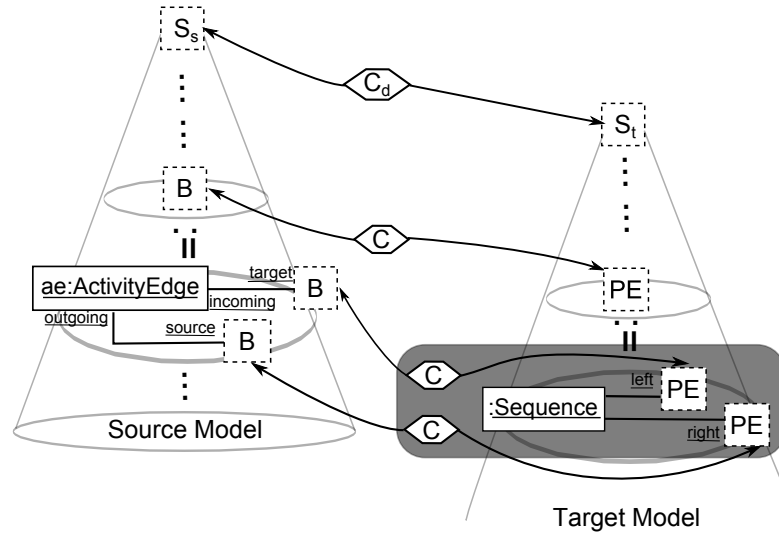


Figure 5.16: GBMT execution sketch

The transformation terminates once the complete source derivation tree has been traversed and all correspondences of the source non-terminal hyperedges have been considered. Due to the context-freeness of HR grammars [Roz97], we can use any traversal method.

Figure 5.16 sketches a GBMT execution. It highlights a single production in the source derivation tree for our example model that rewrites a hyperedge of type B . This hyperedge has a correspondence of type C with a hyperedge of type PE from the already generated fragment of the target derivation tree. The dark gray rectangle frames the result of applying the suitable GBMT rule 9 (see Figure 5.7) to the corresponding target hyperedge of type PE .

We have described an idea of model transformation definition and execution based on HR graph grammars. Next, we look into the details of the formalization of our approach and of all of the concepts present there.

5.3 Formalization of the Approach

In order to give a detailed description of all aspects of our GBMT development approach and, later, be able to prove the qualities of the transformations developed with its help, we provide a formalization of the approach. We start by recapitulating formal definitions of the main notions of HR (hyperedge replacement) grammars, that we use to specify the source and target languages of the developed model transformations (see Section 5.1.2).

The central notions in HR grammars are a *hyperedge*, a *hypergraph*, a *derivation*, an *HR grammar* itself, and its language – *hypergraph language*. We now give their formal definitions and show their inter-relations according to [Roz97].

Definition 5.3.0.1 (Hyperedge). *A hyperedge e is an atomic item with a fixed number of tentacles (attachment points), called the type of the hyperedge and denoted by $\text{type}(e)$. It can be attached to a structure with a set of nodes by attaching each of its tentacles to an attachment node.*

Additionally, there exists a fixed set of labels LB , a *labeling function* for hyperedges – $\text{lab} : E \rightarrow LB$, and a *typing function* for labels – $\text{type} : LB \rightarrow \mathbb{N}$. The typing function maps labels to non-negative integers in such a way that, for each hyperedge e , the type of its label and the number of its *tentacles* coincide – $\text{type}(\text{lab}(e)) = \text{type}(e)$.

During an application of an HR grammar production, a hyperedge is replaced by a *hypergraph*. A hypergraph, in turn, is defined over an arbitrary finite number of nodes and hyperedges as follows.

Definition 5.3.0.2 (Hypergraph). *A hypergraph H over a set of labels LB is a tuple $H = (V_H, E_H, \text{att}_H, \text{lab}_H, \text{ext}_H)$ where:*

- V_H – a finite set of nodes,
- E_H – a finite set of hyperedges,

- $att_H : E_H \rightarrow V_H^*$ – a mapping assigning a sequence of pairwise distinct attachment nodes $att_H(e)$ to each hyperedge $e \in E_H$,
- $lab_H : E_H \rightarrow LB$ – a mapping that labels each hyperedge in H such that $type(lab_H(e)) = |att_H(e)|$,
- and $ext_H \in V_H^*$ – a sequence of pairwise distinct external nodes.

For simplification, the above definition assumes that attachment nodes of each hyperedge as well as external nodes of each hypergraph are pairwise distinct. As pointed-out in [Roz97], these restrictions are non-vital. They simplify the definition and lead to the equivalence of $type(e)$ and $|att_H(e)|$, that implies the equivalence $type(lab_H(e)) = |att_H(e)|$ used in Definition 5.3.0.2.

To keep later definitions concise, we introduce the following notations. Given a set $X \subseteq LB$ of labels, we denote by E_H^X the set $\{e \in E_H \mid lab_H(e) \in X\}$ of hyperedges of H with the labels in X . The class of all hypergraphs over LB is denoted by $\mathcal{H}_{LB}$. Now we can formalize the notion of an HR grammar using these notations.

Definition 5.3.0.3 (Hyperedge Replacement Grammar). *A hyperedge replacement (HR) grammar $G = (N, T, P, S)$ consists of a set of non-terminals $N \subseteq LB$, a set of terminals $T \subseteq LB \setminus N$, a finite set of productions P , and a designated start symbol $S \in N$. Each production $p \in P$ is of the form $p = (A, R)$, with $A \in N$, $R \in \mathcal{H}_{LB}$, and $type(A) = type(R)$ where $type(R) = |ext_R|$.*

The last condition in the above definition $type(A) = type(R)$ together with the simplifications we have previously mentioned, states that the type of A and the number of attachment nodes of a hyperedge labeled by A should be equal to the number of external nodes of R .

By using an HR grammar G , one hypergraph can be obtained from another by applying a production $p \in P_G$ to the latter and replacing a hyperedge labeled by A with the hypergraph R . This step is called *direct derivation* and a sequence of such steps is called *derivation*. Both notions are formally defined as follows.

Definition 5.3.0.4 (Direct Derivation). *A direct derivation, written $H \Rightarrow_p H[e/R]$, is an application of production $p = (A, R) \in P$ to a graph $H \in \mathcal{H}_{LB}$ by choosing an edge $e \in E_H^N$ such that $lab_H(e) = A$, that is then cut out and replaced by R by gluing it to the rest of H . Thereby, each i -th external node of R is fused with the i -th attachment node of e .*

Definition 5.3.0.5 (Derivation). *A sequence of direct derivations $d = H_0 \Rightarrow \dots \Rightarrow H_k$ is called a derivation of length $k \geq 0$ from H_0 to H_k and is denoted by $H_0 \Rightarrow_P^* H_k$ or $H_0 \Rightarrow^* H_k$.*

If the description of an HR grammar, besides the set of productions and the start symbol, contains specification of terminal symbols ($T \neq \emptyset$), it can generate a language – a set of terminal hypergraphs $\{H \in \mathcal{H}_T\}$ derivable from the start hypergraph. The start hypergraph is a so-called *handle graph induced* by the start symbol S of the grammar G . It is defined as follows.

Definition 5.3.0.6 (Handle). *An hypergraph H with m nodes, $|ext_H| = m$, and a single hyperedge $e \in E_H^{LB}$ is a handle if $att_H(e) = ext_H$. If $lab_H(e) = A$, then H is said to be the handle induced by label A denoted by $A^\bullet$.*

The hypergraph language generated by an HR grammar can be formalized as follows.

Definition 5.3.0.7 (Hypergraph Language). *The hypergraph language $L(G)$ generated by an HR grammar G is $L_S(G)$, where for all $A \in N$, $L_A(G)$ consists of all hypergraphs in H_T derivable from $A^\bullet$ by applying productions of P :*

$$L_A(G) = \{H \in \mathcal{H}_T \mid A^\bullet \Rightarrow_P^* H\}.$$

Now that we have formally defined HR grammars and languages they describe, we can describe and formalize the definition (Section 5.3.1) and the execution (Section 5.3.2) of GBMTs between such languages.

5.3.1 Formalization of Transformation Definition

We consider a transformation between a source and a target language defined by finite HR grammars G_s and G_t , respectively. Our GBMTs consist of a number of rules relating productions from the source and the target grammar in a specific way. In many cases, simple 1-to-1 relations between the productions are sufficient. Thus, we consider this basic type of rules first, and describe an extension to n-to-m production relations later.

To define a GBMT rule, we choose two productions $p_s = (A_s, R_s)$ and $p_t = (A_t, R_t)$ from P_s and P_t , respectively, and relate them. Furthermore, we relate the left-hand side non-terminals A_s and A_t of productions building a correspondence type relation, and we map some hyperedges from R_s and R_t onto each other via correspondence mappings.

To describe the types of correspondences we use a finite set of labels CT . It contains at least one label – the so-called *default type* label denoted by $t_c^{default}$ – that is always used to type the *initial correspondence mapping* c^d created during the execution. This initial correspondence mapping was introduced in Section 5.3.2 on the execution of GBMTs. Next, we formalize the definition of a GBMT rule as follows.

Definition 5.3.1.1 (Grammar Based Model Transformation Rule). *A grammar based model transformation (GBMT) rule $tr = (p_s, p_t, c_{pre}, C_{post})$ between the languages defined by HR grammars G_s and G_t consists of productions $p_s = (A_s, R_s) \in P_s$ and $p_t = (A_t, R_t) \in P_t$, a correspondence type pre-condition relation $c_{pre} = (A_s, t_{st}, A_t)$*

between A_s and A_t non-terminals, where $t_{st} \in CT$, and a correspondence function $C_{post} : E_{R_t}^{N_t} \rightarrow CT \times E_{R_s}^{N_s}$ for non-terminal hyperedges of R_t mapping them to the non-terminal hyperedges of R_s and types in CT . Furthermore, the following conditions hold:

1. $A_s = S_s \Leftrightarrow A_t = S_t$, and then $t_{st} = t_c^{default}$,
2. C_{post} is total.

The relation c_{pre} is called *correspondence type pre-condition* due to its role during the transformation execution – a correspondence mapping of its *type* has to exist between some hyperedges of the source and the target derivation tree, *before* a transformation rule with such pre-condition can be applied. Formally, this *type of* relation is defined as follows.

Definition 5.3.1.2 (Correspondence Mapping Type). *A correspondence mapping $c = (e_t, (t_{ts}, e_s))$ is of the type defined by the correspondence type relation $c_{pre} = (A_s, t_{st}, A_t)$, when $lab(e_s) = A_s$, $lab(e_t) = A_t$, and $t_{ts} = t_{st}$.*

Furthermore, by requiring the C_{post} relation to be a function in Definition 5.3.1.1, we forbid 1-to-n correspondence mappings of non-terminal hyperedges in R_t to non-terminal hyperedges in R_s and, thereby, prevent the defined transformation from being non-deterministic. The exact role of this condition will become clear later when we prove the properties of the developed transformations in Section 5.4.

Now that we have defined a GBMT rule in its basic form, we can describe a complete basic GBMT. Such a transformation consists of an arbitrary finite number of basic GBMT rules and is formally defined as follows.

Definition 5.3.1.3 (Grammar Based Model Transformation). *A grammar based model transformation (GBMT) $T = \{tr_1, \dots, tr_n\}$ is a finite set of GBMT rules $tr_i, 1 \leq i \leq n$, fulfilling the following conditions:*

1. *There are no $tr^1, tr^2 \in T$ such that $c_{pre}^1 = c_{pre}^2$, $p_s^1 = p_s^2$, and $p_t^1 \neq p_t^2$.*

There are no two rules in T with the same correspondence type pre-condition relations and source productions, and different related target productions.

2. *For each production $p_s = (A_s, R_s) \in P_s$ and for each correspondence mapping $c = (e_t, (t_{ts}, e_s)) \in \bigcup_{i=1}^n C_{post}^i \cup \{c^d = (e_t^d, (t_c^{default}, e_s^d)) \mid lab(e_t^d) = S_t, lab(e_s^d) = S_s\}$, where $lab(e_s) = A_s$ and $lab(e_t) = A_t$ for some $A_t \in N_{G_t}$, there exists a rule $tr = (p_s, p_t, c_{pre}, C_{post}) \in T$ such that $c_{pre} = (A_s, t_{ts}, A_t)$.*

There is a transformation rule in T for each combination of a source production and a correspondence mapping $c = (e_t, (t_{ts}, e_s))$ defined by a function C_{post}^i ($1 \leq i \leq n$) or as the initial mapping – $c^d = (e_t^d, (t_c^{default}, e_s^d))$ – such that its last value e_s is labeled with A_s . Such a rule then has a correspondence type pre-condition relation $c_{pre} = (A_s, t_{ts}, A_t)$ that equals the type of c .

The conditions stated in the above definition ensure that the defined transformation will have the desired correctness and determinism properties proven later. The first condition prohibits multiple GBMT rules with the same source production and correspondence type pre-condition, and different target productions, since they are applicable at the same time. This would make the transformation execution non-deterministic.

The second condition ensures that for each source production and each type of correspondence mapping a hyperedge labeled with its left-hand side non-terminal A_s can be involved in, there is a transformation rule in T with these source production and correspondence type pre-condition. The types of the correspondence mappings a hyperedge can be involved in during the transformation execution are defined by C_{post} functions of all transformation rules in T and by the initial mapping $c^d = (e_t^d, (t_c^{default}, e_s^d))$ created during the execution. These C_{post} functions and the c^d mapping define the actual mappings and, thereby, also their type that is fully determined by the combination of the labels of e_s and e_t in LB , and the correspondence type label t_{ts} in CT .

As we have seen on our ADR-to-CSP transformation example in Section 5.2.1, sometimes a 1-to-1 relation of productions is not enough to describe the desired transformation logic. Now, we formalize the *extended* type of GBMT rules, we have previously introduced for such transformation cases. This type of rule allows n-to-m relations between the productions of the source and the target HR grammar.

To specify extended GBMT rules, we choose the derivations we want to relate – a derivation of the length n in the source grammar and a derivation of the length m in the target grammar, where $n, m \geq 1$. Due to the context-freeness of HR grammars, we can and do use *derivation trees* to represent the derivations in our rules. Thus, we now provide a definition of a derivation tree, taken from [Roz97] and extended by introducing a set of non-terminal hyperedges E_t^N for a tree t (analogous to those of a hypergraph).

Definition 5.3.1.4 (Derivation Tree). *The set $TREE(P)$ of derivation trees over productions P in a grammar G is recursively defined as follows:*

- $A \in TREE(P)$ with $root(A) = A$, $result(A) = A^\bullet$, and $E_A^N = E_{A^\bullet}^N$ for all $A \in N$.
- For every production $p = (A, R) \in P$ and every mapping $branch : E_R^N \rightarrow TREE(P)$ such that $lab(e) = lab(root(branch(e)))$ for all $e \in E_R^N$, the triple $t = (A, R, branch) \in TREE(P)$.

Furthermore, we let $root(t) = A$, $result(t) = R[e_1/result(branch(e_1))] \cdots [e_{|E_R^N|}/result(branch(e_{|E_R^N|}))]$, where $e_i \in E_R^N$, and $E_t^N = E_R^N \cup \bigcup_{e \in E_R^N} E_{branch(e)}^N$.

This definition distinguishes the following two types of derivation trees: a *non-terminal tree* A in N and a *production based tree* for a production $p = (A, R)$. The second type of tree is then recursively constructed through sub-trees, connected via *branch* mappings to the non-terminal hyperedges e in R . Furthermore, one derivation tree t_p can be a *prefix* of the other derivation tree t , when t can be constructed from t_p by mapping some

hyperedges e in $E_{t_p}^N$ with non-terminal sub-trees $branch(e) \in N$ to the new production based sub-trees such that $branch(e) \in TREE(P) \setminus N$. This notion of prefix is used later for the application of GBMT rules during the execution.

The fact that we can use trees to represent derivations that we want to relate is guaranteed by the following Proposition 1 proven in [Roz97]. It states the required correspondence between derivations and derivation trees.

Proposition 1. *Let P_G be a set of productions of grammar G , let $A \in N_G$ and $H \in \mathcal{H}_{\mathcal{T}_G}$. Then there is a derivation $A^\bullet \Rightarrow^* H$ iff there is a derivation tree $t \in TREE(P_G)$ with $root(t) = A$ and $result(t) = H$.*

Now, we can use the defined derivation trees to formalize the notion of an extended GBMT rule. Instead of single productions, such a rule relates derivation sub-trees over P_s and P_t and, like in the previous basic rule type, it also specifies the correspondence mappings, but in an extended way. It requires each target *leaf* – non-terminal hyperedge which is produced and not consumed (rewritten) by the target derivation sub-tree – to have exactly one corresponding *leaf* non-terminal hyperedge in the related source derivation sub-tree. This condition is formalized as part of the following definition.

Definition 5.3.1.5 (Extended Grammar Based Model Transformation Rule). *An extended GBMT rule $tr = (t_s, t_t, c_{pre}, C_{post})$ is constructed from a GBMT rule (Definition 5.3.1.1) by extending its source and target productions p_s and p_t to finite derivation sub-trees $t_s \in TREE(P_s) \setminus N_s$ and $t_t \in TREE(P_t) \setminus N_t$, respectively, and by extending the domain and co-domain of its correspondence function such that $C_{post} : E_{result(t_t)}^{N_t} \rightarrow CT \times E_{result(t_s)}^{N_s}$.*

Since a single production $p = (A, R)$ can be represented as a *one-production derivation tree* $t = (A, R, \{branch(e) = lab(e) \mid \forall e \in E_R^N\})$, Definition 5.3.1.1 is a special case of Definition 5.3.1.5.

Now, our transformation can contain extended GBMT rules. Thus, we need to refine its definition, and re-visit and adjust the conditions stated in Definition 5.3.1.3 to ensure the desired transformation quality properties.

The possibility to use non-trivial source derivation sub-trees with potentially multiple production based sub-trees in extended GBMT rules can compromise completeness of the defined transformation. For instance, when only a GBMT rule for one possible sub-derivation in source models has been defined. To avoid this, we require a *well-formed rule set* for each combination of a source production $p_s = (A_s, R_s)$ and a correspondence type pre-condition relation c_{pre} . Such *rule set* for each p_s and c_{pre} contains all extended GBMT rules tr with c_{pre} whose source derivation sub-trees start with p_s such that $t_s = (A_s, R_s, branch_s)$. A *well-formed rule set* then either uses *rule priorities* and contains a *special rule* with the lowest priority (1), or, alternatively, fulfills the *coverage criteria* we formulate (2). We describe both of these cases next.

In case 1, such a set must contain one special rule with a one-production source derivation sub-tree. This rule must have the lowest priority in the set to guarantee that it is only used when no other GBMT rules can be applied. The presence of this special rule gives us the guarantee that the defined transformation will transform each application of production p_s to a non-terminal hyperedge with the correspondence mapping of the type c_{pre} , regardless of the productions applied later to the hyperedges in R_s .

If no such special rule has been defined for a p_s and c_{pre} combination, to, for example, avoid putting additional unused logic in the transformation definition, then, we have case 2, where the rules in the set for p_s and c_{pre} must fulfill the *coverage criteria* together. This coverage criteria requires the rules of the set together to describe all *replacement alternatives* available in the grammar G_s for their source derivation sub-trees. For one rule tr in a rule set $RS_{p_s, c_{pre}}$ such replacement alternatives for its source sub-tree t_s are defined as follows.

Definition 5.3.1.6 (Replacement Alternative). *A derivation sub-tree t_{alt} is a replacement alternative for sub-tree $t = (A, R, branch_t)$ in an HR grammar G when:*

1. $t, t_{alt} \in TREE(P)$ and $t_{alt} = (A, R, branch_{alt})$;
2. For all hyperedges $e_{alt} \in E_{t_{alt}}^N$: if there is a hyperedge $e \in E_t^N$ in t at the same position as e_{alt} in t_{alt} then $branch_{t_{alt}}(e_{alt}) \in N$ iff $branch_t(e) \in N$; otherwise, $branch_{t_{alt}}(e_{alt}) \in N$;
3. There exists a hyperedge $e'_{alt} \in E_{t_{alt}}^N$ with $branch_{t_{alt}}(e'_{alt}) \notin N$ at the same position as some hyperedge $e' \in E_t^N$ such that $branch_{t_{alt}}(e'_{alt}) \neq branch_t(e')$.

The first condition of this definition requires a replacement alternative derivation sub-tree t_{alt} to start with the same production as t , and to be from the tree set $TREE(P)$.

The second condition requires that all non-terminal hyperedges from t and t_{alt} , which appear at the same position in both trees, are mapped to the same (non-terminal or production based) type of sub-tree by the corresponding *branch* mappings. Thereby, two non-terminal hyperedges are considered to appear at the same position in their trees, when the roots of the trees are the same and the *paths* to the hyperedges in these trees are the same, i.e. contain the same⁴ hyperedges in the same order. Formally, a *path* is defined as follows.

Definition 5.3.1.7 (Hyperedge Path). *A path from a root A to a hyperedge e in a tree $t = (A, R, branch)$ is a sequence $e_1 \cdots e_n \in E_t^{N*}$, where $e_1 \in E_R^N$, $e_n = e$, $e_{i+1} \in E_{R_{branch(e_i)}}^N$ for $1 \leq i < n$, and its length equals n .*

The second condition of Definition 5.3.1.6 means, that one hyperedge in t_{alt} has the non-terminal $A \in N$ as a sub-tree, if and only if, the other one at the same position in

⁴Every hyperedge is assumed to have a unique identification label (id) in a hypergraph. Hyperedges in the same hypergraph and with the same id are considered equivalent.

t has it as well. If a hyperedge only appears in t_{alt} , its sub-tree must be assigned to a non-terminal. This keeps the number of alternative replacement sub-trees finite.

Finally, the last condition of that definition requires the alternative sub-tree t_{alt} to differ from t in at least one *branch* mapping for a non-terminal hyperedge, that has an equivalent in t and whose mapped sub-tree is not a non-terminal.

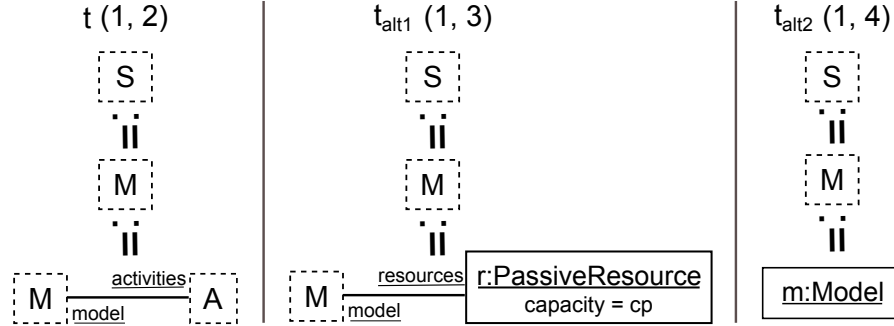


Figure 5.17: Replacement alternatives for sub-tree t in the ADR HR grammar

To clarify the above definition for replacement alternatives, we consider an example source sub-tree in our HR grammar for ADR language (see Figure 5.3). Figure 5.17 shows an example sub-tree t (left) and its two replacement alternatives t_{alt1} (middle) and t_{alt2} (right) in our ADR HR grammar. Next to each of these three sub-trees we see the numbers of the productions used in them (in brackets).

These two sub-trees t_{alt1} and t_{alt2} are replacement alternatives of t since they fulfill the conditions stated in Definition 5.3.1.6. First, they are both in $TREE(P_{ADR})$ (see Definition 5.3.1.4) and their first productions ($S := M$) are equal to the first one of the sub-tree t . Thus, the condition 1 of Definition 5.3.1.6 is fulfilled.

Then, all the non-terminal hyperedges of t_{alt1} and t_{alt2} that have hyperedges at the same position in t – the hyperedges of type M produced by production 1 – have production based sub-trees, if and only if, this is also the case in t (for the equally positioned). The hyperedge of type M produced by production 1 in sub-tree t has a production based sub-tree (production 2) and so do the equally positioned hyperedges in t_{alt1} and t_{alt2} with production 3 and production 4 based sub-trees, respectively. All non-terminal hyperedges of t_{alt1} and t_{alt2} that do not have equally positioned ones in t – the hyperedge of type M produced by production 3 – has a non-terminal sub-tree ($M \in TREE(P_{ADR})$ sub-tree). Thus, the condition 2 of Definition 5.3.1.6 is also fulfilled.

Finally, in t_{alt1} and t_{alt2} there are non-terminal hyperedges – the hyperedges of type M produced by production 1 – that have an equally positioned one in t , have production sub-trees in t_{alt1} (production 3) and t_{alt2} (production 4) that differ from the one of the equally positioned hyperedge in t (production 2). Thus, the last condition 3 of Definition 5.3.1.6 is also fulfilled.

One question is left to clarify. Does Figure 5.17 shows all possible replacement alternatives of t in our ADR HR grammar? Could one, for instance, add a sub-tree to the bottom hyperedge of type M in the t_{alt1} sub-tree to create a new possible replacement alternative for t ? We address this question in Proposition 2 next.

Based on Definition 5.3.1.6 and the conditions it states on a replacement alternative, we can prove the following two propositions. The first one is concerned with the *length of a replacement alternative tree*, where the *length of a tree* is defined as the length of the path (see Definition 5.3.1.7) with the most hyperedges to a hyperedge in that tree.

Proposition 2 (Length of Replacement Alternative). *Given a derivation tree $t = (A, R, \text{branch})$ and its replacement alternative $t_{alt} = (A, R, \text{branch}_{alt})$, the length of t_{alt} is less or equal to the length of t .*

Proof. Assume the opposite is true. There exists a path $ph = e_1 \cdots e_n$ in t_{alt} such that the length of ph is *greater* than the length of any path in t . When $n = 1$, by Definition 5.3.1.7, $e_n \in E_R^N \Rightarrow$ exists path in t to e_n with the length of ph ($\nless$).

Consider $n > 1$. By Definition 5.3.1.7, $e_n \in E_{R_{\text{branch}_{alt}(e_{n-1})}}^N \Rightarrow \text{branch}_{alt}(e_{n-1}) \notin N \Rightarrow$ by Definition 5.3.1.6, exists $e' \in E_t^N$ at the same position in t as e_{n-1} in t_{alt} and with $\text{branch}(e') \notin N \Rightarrow e'$ has the same path ph' in t as e_{n-1} in $t_{alt} \Rightarrow$ the path ph' is shorter than ph by 1. As $\text{branch}(e') \notin N$, exists $e'_n \in E_{\text{branch}(e')}^N$ such that its path in t has the length $ph' + 1$ and its length equals ph ($\nless$).

Thus, the length of a replacement alternative of a tree is always equal or shorter than the length of the tree itself. $\square$

We now know, that each replacement alternative of a tree is at most as long as this tree. Thus, our set of replacement alternatives for one tree stays finite.

For our example in Figure 5.17 this result means that the replacement alternatives of t can not be extended to (be a prefix of) new ones. They already represent all possible alternative replacements of the hyperedge created by production 1 in our ADR HR grammar. Thus, the only possible additional replacement alternative our example sub-tree t can have is the tree with the length smaller than the one of t . Such a tree, by Definition 5.3.1.6, would start with the same production as t and would have a non-terminal sub-tree M for the produced hyperedge of the same type. This violates condition 2 of Definition 5.3.1.6 and, thus, Figure 5.17 shows all replacement alternatives of our tree t .

The second proposition is about the relation between two replacement alternatives, i.e. whether one alternative can be a prefix of the other one. It considers the properties of the shorter replacement alternatives in detail.

Proposition 3 (Replacement Alternative Distinction). *Given a sub-tree $t = (A, R, \text{branch})$ and its replacement alternative $t_{alt} = (A, R, \text{branch}_{alt})$, t_{alt} is not equal, nor an extension or prefix of t .*

Proof. Assume the opposite is true. The tree t_{alt} is equal to $t \Rightarrow$ contradiction with Definition 5.3.1.6 (3) ($\nexists$).

The tree t_{alt} is a prefix of $t \Rightarrow$ there is no $e_{alt} \in E_{t_{alt}}^N$ such that $branch_{alt}(e_{alt}) \notin N$ and for $e \in E_t^N$ at the same position in t as e_{alt} in t_{alt} $branch(e) \neq branch_{alt}(e_{alt}) \Rightarrow$ contradiction with Definition 5.3.1.6 (3) ($\nexists$).

The tree t_{alt} is an extension of $t \Rightarrow$ exists $e_{alt} \in E_{t_{alt}}^N$ such that $branch_{alt}(e_{alt}) \notin N$ and for $e \in E_t^N$ at the same position in t as e_{alt} in t_{alt} $branch(e) \in N \Rightarrow$ contradiction with Definition 5.3.1.6 (2) ($\nexists$).

Thus, a replacement alternative of a tree is never equal, nor an extension or a prefix of it. $\square$

We have shown how a replacement alternative relates to the tree it replaces and which length it can have. Next, we consider which role these alternatives play during the definition of a GBMT.

Informally, if a rule tr with the first production p_s in its source derivation sub-tree t_s and a pre-condition correspondence type c_{pre} has n non-terminal hyperedges e in t_s which $branch(e) \notin N$, we consider each such hyperedge e . The tree t_s describes one possible replacement for e which is defined in $branch(e)$. If there are productions in G_s allowing alternative replacements of e , then there must be GBMT rules describing all these replacement alternatives in their source derivation sub-trees. If there are multiple such hyperedges e in $E_{t_s}^N$ ($n \geq 1$), then the GBMT must have rules describing all different combinations of these alternative replacements for each e in their source sub-trees.

Another solution to cover all possible alternative derivations is to have a *special rule* for each combination of a source production p_s and a correspondence type pre-condition c_{pre} such that, its source derivation sub-tree t_s contains only one production – p_s . By assigning this rule a *priority* which is lower than those of all other rules for this p_s and c_{pre} pair, we only use it, when none of the other GBMT rules are applicable. When this method is used, a non-conflicting priority mapping $priority : T \rightarrow \mathbb{N}$ has to be defined on all rules of the developed GBMT T .

In both cases, only a single combination of a source production p_s and a correspondence type pre-condition c_{pre} have to be determined to build a so-called *well-formed rule set* for p_s and c_{pre} . This allows us to choose between the methods described above for the different combinations. For the completeness of the whole GBMT, one of these methods has to be applied to each such combination existing in the transformation definition. We state this in the following definition of a *well-formed rule set*.

Definition 5.3.1.8 (Well-Formed Rule Set). *A rule set $RS_{p_s, c_{pre}} \subseteq T$ for a production $p_s = (A_s, R_s) \in P_s$ and a correspondence type pre-condition relation $c_{pre} = (A_s, t_{st}, A_t)$ in a GBMT T consists of the rules with $t_s = (A_s, R_s, branch_s)$ and with the correspondence type pre-condition c_{pre} . $RS_{p_s, c_{pre}}$ is well-formed iff it is not empty and one of the following holds:*

1. *[Special Rule]* There exists exactly one GBMT rule $tr \in RS_{p_s, c_{pre}}$ such that $t_s = (A_s, R_s, \{\text{branch}_s(e) = \text{lab}(e) \mid \forall e \in E_{R_s}^{N_s}\})$ is a one-production derivation sub-tree and its priority is the lowest in $RS_{p_s, c_{pre}}$.
2. *[Coverage Criteria]* For each GBMT rule $tr \in RS_{p_s, c_{pre}}$ and each replacement alternative t_{alt} for t_s available in G_s (see Definition 5.3.1.6), there exists a GBMT rule $tr' \in RS_{p_s, c_{pre}}$ such that $t'_s = t_{alt}$. Furthermore, for all distinct $tr, tr' \in RS_{p_s, c_{pre}}$, t'_s is a replacement alternative of t_s .

The first condition of this definition states the requirements for the case with one *special rule* of lower priority than the rest of the GBMT rules in the set. The second condition states the requirements for the case with the coverage criteria that we have introduced. It demands that for each GBMT rule in the set, the set also contains the GBMT rules with the source derivation sub-trees equal to the replacement alternatives of the source derivation sub-tree of that rule. This way, we cover all alternative derivation paths starting with an application of p_s . Plus in the second condition, we also demand that the source derivation sub-trees of all rules in the set are replacement alternatives of each other. This is done to make use of Proposition 3, and avoid having multiple GBMT rules applicable at one point during an execution.

Combination of these two cases is also possible within a production/correspondence type pre-condition relation combination, for example, when the special rule has to have a source derivation sub-tree involving more than one production. However, currently, we do not consider this option.

The description above requires a new definition of a GBMT with extended transformation rules, which we give next.

Definition 5.3.1.9 (Extended Grammar Based Model Transformation). *A GBMT $T = \{tr_1, \dots, tr_n\}$ is extended iff it contains extended⁵ GBMT rule(s) and fulfills the following additional conditions:*

1. *There are no $tr^1, tr^2 \in T$ with the same priority value such that $c_{pre}^1 = c_{pre}^2$ and t_s^1 is a prefix of (or equals) t_s^2 .*

There are no two rules in T with the same priority, same correspondence type pre-condition relation, and one source derivation sub-tree being a prefix of (or equals) the other one.

2. *For each production $p_s = (A_s, R_s) \in P_s$ and each correspondence mapping $c = (e_t, (t_s, e_s)) \in \bigcup_{i=1}^n C_{post}^i \cup \{c^d\}$ such that $\text{lab}(e_s) = A_s$, there exists exactly one well-formed rule set $RS_{p_s, c_{pre}} \subseteq T$ (Definition 5.3.1.8) such that c_{pre} is the type of c (Definition 5.3.1.2).*

There is exactly one well-formed rule set $RS_{p_s, c_{pre}} \subseteq T$ for each combination of a source production p_s and a correspondence mapping c whose type equals c_{pre} . This

⁵Basic GBMT rule type defined in Definition 5.3.1.1 is a special case of an extended GBMT rule.

correspondence mapping c is either produced by a function C_{post} of some GBMT rule in T or given as the initial mapping c^d .

The above definition requires T to be finite, yet it should contain a well-formed rule set RS for each production and correspondence type combination. When a rule set contains a special rule, it automatically fulfills the well-formedness criteria 1 of Definition 5.3.1.8, and, hence, can be finite. In the case when a rule set fulfills the well-formedness criteria 2 of Definition 5.3.1.8, we need to show that this rule set is finite, and, therefore, it is possible to build a finite T containing such type of rule sets. We show this in the following proposition.

Proposition 4 (Finite Well-Formed Rule Set). *Given a well-formed rule set $RS_{p_s, c_{pre}} \subseteq T$ for a production $p_s = (A_s, R_s)$ and a correspondence type c_{pre} that fulfills the coverage criteria (Definition 5.3.1.8 (2)) and a finite⁶ source HR grammar G_s of T , then $RS_{p_s, c_{pre}}$ is always finite.*

Proof. The rule set $RS_{p_s, c_{pre}}$ fulfills Definition 5.3.1.8 (2) $\Rightarrow$ for all distinct $tr, tr' \in RS_{p_s, c_{pre}}$, t'_s is a replacement alternative of $t_s \Rightarrow$ by Proposition 2, the lengths of t_s of the rules in $RS_{p_s, c_{pre}}$ are comparable $\Rightarrow$ there exists $tr \in RS_{p_s, c_{pre}}$ with t_s whose length n is greater than or equals the lengths of t_s of all other rules in $RS_{p_s, c_{pre}}$. By Definition 5.3.1.5, t_s of this tr is finite $\Rightarrow n$ is finite.

As G_s is finite, there is a finite number of distinct derivation trees $t = (A_s, R_s, branch_s) \in TREE(P_s)$ with the length of up to a finite $n \Rightarrow RS_{p_s, c_{pre}}$ is always finite. $\square$

Now, with the above definition of an extended GBMT, we can relate source and target derivation sub-trees in our GBMT rules. However, in some cases, the transformation under development requires some non-local behavior specification, for instance, when a transformation rule has to create additional target language structures somewhere else in the target model. We have seen examples of such non-local behavior in our ADR-to-CSP transformation requirements in Table 3.1 (e.g., requirements 7, 13). In Section 5.2.1, we have introduced so-called non-local GBMT rules to address such requirements and shown their examples for our ADR-to-CSP transformation (e.g., Figure 5.13 and 5.14). Now, we formalize this type of GBMT rules.

Formally, such non-local behavior would mean connecting a sub-tree, that creates additional structures, via a $branch_t$ mapping to some already constructed part (non-terminal hyperedge) of the target derivation tree *other* than the one, where the main target sub-tree of the rule is mapped to. The following definition of a *non-local GBMT rule* describes this formally.

Definition 5.3.1.10 (Non-Local Grammar Based Model Transformation Rule). *A non-local GBMT rule $tr = (t_s, t_t, t_t^a, c_{pre}, c_{pre}^a, C_{post})$ is an extended GBMT rule with an additional correspondence type pre-condition relation $c_{pre}^a = (A_s^a, t_{st}^a, A_t^a)$, an additional*

⁶We generally only consider finite HR grammars for language definition, yet state it here explicitly.

finite target derivation sub-tree $t_t^a = (A_t^a, R_t^a, \text{branch}_t^a)$, and a correspondence function with an extended domain and co-domain such that $C_{\text{post}} : \left\{ E_{\text{result}(t_t)}^{N_t} \cup E_{\text{result}(t_t^a)}^{N_t} \right\} \rightarrow CT \times \left\{ E_{\text{result}(t_s)}^{N_s} \cup \{e_s^a\} \right\}$ with $e_s^a \in E_{A_s^a}^{N_s}$, where the following additional conditions hold:

1. $c_{\text{pre}}^a \neq c_{\text{pre}}$.
2. There exists exactly one $c^a = (e_t^a, (t_{st}^a, e_s^a)) \in C_{\text{post}}$ of type c_{pre}^a such that $e_t^a \in E_{\text{result}(t_t^a)}^{N_t}$ and is identified with the $\text{root}(t_t^a)$ via a hyperedge identity mapping. This mapping states that, when the top production (A_t^a, R_t^a) of t_t^a is applied to a hyperedge with label A_t^a , this hyperedge is kept as e_t^a in the replacing R_t^a .

The first condition of the above definition forbids a non-local rule to use the same main and additional correspondence type. This way, we avoid concurrent *branch* mapping assignments for the same hyperedge, which could lead to non-deterministic results.

The second condition ensures that each non-local rule propagates the additional correspondence mapping, that it requires for the application. This is done through the leaf hyperedge e_t^a of t_t^a , that is identified with the $\text{root}(t_t^a)$ via a hyperedge identity mapping, and its correspondence mapping c^a to some e_s^a . When such non-local rule is applied, e_s^a and e_t^a (through identity mapping) will become the hyperedges of the new suitable additional correspondence mapping of type c_{pre}^a . This way, we ensure that, after an application of a non-local GBMT rule, the additional correspondence mapping it has used, still exists and allows the application of other rules.

Now, we can use non-local rules in our GBMTs. Though, to be able to guarantee the desired properties, we need to re-visit and amend the conditions on the complete transformations with non-local GBMT rules. The amended conditions must ensure for each non-local rule of the transformation that, when it is applied, there exists exactly one additional correspondence mapping of type c_{pre}^a to be used for the rule application. Now, we formalize the conditions sufficient to ensure this in the definition of a GBMT with non-local rules.

Definition 5.3.1.11 (Non-Local Grammar Based Model Transformation). *An extended GBMT $T = \{tr_1, \dots, tr_n\}$ is non-local iff it contains non-local GBMT rule(s) and fulfills the following additional conditions:*

1. For each non-local GBMT rule $tr = (t_s, t_t, t_t^a, c_{\text{pre}}, c_{\text{pre}}^a, C_{\text{post}}) \in T$, there is a local GBMT rule $tr' \in T$ with $c'_{\text{pre}} \neq c_{\text{pre}}^a$ and $c' \in C'_{\text{post}}$ of type c_{pre}^a ;
2. For each non-local GBMT rule $tr \in T$, there is no rule $tr' \in T$ such that there exist distinct $c'_1, c'_2 \in C'_{\text{post}}$ of type c_{pre}^a .

Condition 1 of the above definition ensures that a non-local GBMT T contains a (not non-local) rule tr' , that creates (and does not require as c'_{pre}) an additional correspondence c^a_{pre} for each non-local rule tr . Condition 2 ensures that c^a_{pre} for a non-local rule is created at most once in a GBMT rule application.

Transformation T can contain multiple rules fulfilling condition 1 and creating a correspondence of type c^a_{pre} for some tr in T . To guarantee that a transformation rule that requires this type of additional correspondence (i.e. tr) is deterministically applicable, we need to state the following extra condition on a GBMT for each of its non-local rules. This condition is not always trivial to show.

Definition 5.3.1.12 (Additional Correspondence Creation Condition (ACCC)). *Given a non-local GBMT T , for each non-local rule $tr \in T$: only one rule $tr' \in T$ fulfilling condition 1 of Definition 5.3.1.11 for tr is always applied exactly once and before tr during an execution⁷ of T on an input source model $M_s \in L(G_s)$.*

This additional condition considers all possible executions of the transformation T on all possible input models. In general, it might not be possible to show this condition due to an infinite number of models in the source language. It depends both on the way the transformation rules are defined as well as on the productions available in the source and target grammars G_s and G_t for their definition. Furthermore, depending on the chosen additional correspondence type and the productions in G_s and G_t , that create the elements involved in this type of mapping, a definition of the desired non-local GBMT fulfilling the above condition may not be possible.

For our example ADR-to-CSP transformation and the GBMT we have defined for it in Section 5.2.1, this additional ACCC condition holds, which we show here. All non-local GBMT rules for our ADR-to-CSP example use an M -to- M correspondence as an additional correspondence type pre-condition (see e.g., Figure 5.13, 5.14, and 5.15). A correspondence of this type is created the first time by GBMT rule 1 (see Figure 5.8), that fulfills condition 1 of Definition 5.3.1.11. This rule tr' is a basic GBMT rule with a correspondence type pre-condition S -to- S which is different from M -to- M ($c'_{pre} \neq c^a_{pre}$), and it creates a correspondence mapping of type M -to- M ($c' \in C'_{post}$ of type c^a_{pre}).

Now, we just need to show that this GBMT rule is (1) the only such rule, that is (2) always applied exactly once and (3) before each of our non-local GBMT rules, during an execution of the ADR-to-CSP GBMT on an input source model. Point 1 is always true, since this is the only rule in our ADR-to-CSP GBMT that fulfills condition 1 of Definition 5.3.1.11. Point 2 is also true, since this rule is defined for the source production $S := M$, that consumes a hyperedge of the start graph of G_s , that is never re-created by any other production of our ADR HR grammar (see Section 5.1.2, Figure 5.3). Thus, this rule can only be applied once and it is always applied, since each source model is derived using this production from the start graph and its hyperedge always has the correspondence mapping of type S -to- S (the initial correspondence). Finally, Point 3 is

⁷Formalization of the GBMT rule applicability and of the GBMT execution are given in Section 5.3.2.

also true, since this rule is applied before any other rule of the ADR-to-CSP GBMT can become applicable (a mapping of its pre-condition type has to be created).

Combined, both conditions of Definition 5.3.1.11 with the additional ACCC condition (Definition 5.3.1.12) ensure that, for each non-local GBMT rule $tr \in T$, when it is applicable to a hyperedge e_s with a correspondence mapping $c = (e_t, (t_{ts}, e_s))$ during the transformation execution, there exists exactly one *suitable additional correspondence mapping* c_a of type c_{pre}^a in the structure created during the execution of T before this rule is applied. An additional correspondence mapping $c_a = (e_t^a, (t_{ts}^a, e_s^a))$ is *suitable*, when no rule has yet been applied to e_s^a with this mapping – its target hyperedge e_t^a is mapped to a non-terminal sub-tree ($branch_t'(e_t^a) \in N_t$), or the rule tr' , that has been applied to e_s^a in c^a , does not keep (re-produce) an additional correspondence mapping of type $c_{pre}^a - \{ c' \in C'_{post} \mid c' \text{ is of type } c_{pre}^a \} = \emptyset$. A formal description of a suitable additional correspondence mapping and of its role in the execution are given later.

Compliance of Our ADR-to-CSP GBMT Definition Up to now, we have formally defined different types of GBMT rules, starting with the basic ones relating single productions and extending them to relate derivation trees (extended rules) and to support additional non-local behavior on target derivation trees (non-local rules). We have defined GBMT rules of all of these types for our example ADR-to-CSP transformation in Section 5.2.1. It is only left to ensure that these GBMT rules fulfill the conditions stated in the provided formal definitions for their types.

We start with the basic GBMT rules and move later to the extended and, finally, to the non-local rules. We discuss in details why one selected rule from each type is conform with the respective formal definitions and the conditions stated above. For the other GBMT rules of the type, we only provide a short summary.

For the basic GBMT rule type we consider rule 9 shown in Figure 5.7. This rule has to comply with Definition 5.3.1.1 – it has to consist of a source and a target HR grammar production, a correspondence type pre-condition relation between their left-hand sides, and a correspondence function with certain properties. Our example rule 9 contains all of the four parts. We just need to ensure that its correspondence function has the desired properties and maps each non-terminal hyperedge of the right-hand side of the target production onto a pair of a correspondence type and a non-terminal hyperedge of the right-hand side of the source production. This last requirement is met by our rule 9 – it maps both produced non-terminal hyperedges of type PE onto the non-terminal hyperedges of type B and a correspondence type C . Also, since the left-hand sides of the productions in rule 9 are not the start symbols of the source and target HR grammars, the first condition of the definition is automatically fulfilled. Finally, the correspondence function defined in rule 9 is a total function, since it relates *each* target hyperedge of type PE to a correspondence type C and a hyperedge of type B , and none of them are mapped onto more than one such pair. Thus, the considered GBMT rule 9 complies with Definition 5.3.1.1.

For all other basic rules of our ADR-to-CSP GBMT the compliance with Definition 5.3.1.1 can be shown in the same way. They all have the four parts described above and their correspondence functions have the desired properties, which can be easily checked. For rule 7, the correspondence function is not visualized, since its domain and co-domain are empty.

Now, we consider the extended type of GBMT rules and rule 8 from our ADR-to-CSP GBMT definition, that is shown in Figure 5.10, as an example. This rule has to comply with Definition 5.3.1.5 – it has to consist of two finite production based sub-trees ($\notin N$), one in the source and one in the target HR grammar, a correspondence type pre-condition relation between their roots (left-hand sides of the first productions), and a correspondence function with certain properties inherited from Definition 5.3.1.1.

Our example rule 8 contains all of the four parts. We just need to ensure that its correspondence function has the desired properties and maps each non-terminal leaf hyperedge of the target derivation sub-tree ($E_{result(t_t)}^{N_t}$) onto a pair of a correspondence type and a non-terminal leaf hyperedge of the source derivation sub-tree ($E_{result(t_s)}^{N_s}$). This requirement is met by our rule 8 – it maps the only leaf non-terminal hyperedge of type PE onto a leaf non-terminal hyperedge of type B and a correspondence type C .

The roots of the related sub-trees in rule 8 are not the start symbols of the source and target HR grammars and, thus, the first condition of Definition 5.3.1.1 on the correspondence function is automatically fulfilled. Furthermore, the correspondence function defined in rule 8 is total – it relates *each* target leaf hyperedge of type PE to a correspondence type C and a leaf hyperedge of type B , and none of them are mapped onto more than one such pair. Thus, the considered extended GBMT rule 8 complies with Definition 5.3.1.5.

For all other extended rules of our ADR-to-CSP GBMT the compliance with Definition 5.3.1.5 can be shown in the same way. They all have the four parts described above and their correspondence functions have the desired properties, which can easily be checked. For rule 4, the correspondence function is not visualized, since its domain and co-domain are empty.

Finally, we now consider the non-local type of GBMT rules and rule 12 from our ADR-to-CSP GBMT definition, shown in Figure 5.13, as an example. This rule has to comply with Definition 5.3.1.10 – like an extended GBMT rule, it has to consist of two finite derivation production based sub-trees ($\notin N$) and a correspondence type pre-condition relation between their roots. It also has to have an additional correspondence type pre-condition, an additional finite production based target derivation sub-tree ($\notin N$), and an extended correspondence function with certain properties, both those inherited from Definition 5.3.1.5 and the new ones.

Our example non-local GBMT rule 12 contains all of the following six parts: a finite production based source sub-tree for a loop with a body represented by a block hyperedge of type B , a finite production based target sub-tree for a process $L(c)$, a

correspondence type C pre-condition relation $B\text{-to-}PE$ between their roots, an additional correspondence type C pre-condition relation $M\text{-to-}M$ linked to the root of an additional finite production based target sub-tree for a recursive process $L(x)$, and a correspondence function that relates the leaf hyperedge of type PE of the additional target sub-tree to the leaf hyperedge of type B of the source sub-tree. We just need to ensure that this correspondence function has the desired properties and maps each non-terminal leaf hyperedge of both target derivation sub-trees ($E_{result(t_t)}^{N_t} \cup E_{result(t_t^a)}^{N_t}$) onto a pair of a correspondence type and a non-terminal leaf hyperedge of the source derivation sub-tree ($E_{result(t_s)}^{N_s}$) or the hyperedge (e_s^a) of the handle ($A_s^{a\bullet}$), induced by the source non-terminal (A_s^a) of the additional correspondence type pre-condition.

This last mapping requirement is met by our rule 12 – it maps the leaf non-terminal hyperedge of type PE of the additional target sub-tree onto the leaf non-terminal hyperedge of type B and a correspondence type C . It maps the leaf non-terminal hyperedge of type M with the identity mapping from the same sub-tree onto the hyperedge (e_s^a) of the handle graph ($M^\bullet$), induced by the source non-terminal M and a correspondence type C , and none of these leafs is mapped onto more than one such pair. Thus, the correspondence function of rule 12 is total. Furthermore, the roots of the main sub-trees in rule 12 are not the start symbols of the involved HR grammars and, thus, the first condition on the correspondence function inherited from Definition 5.3.1.1 is fulfilled.

As for the additional conditions stated in Definition 5.3.1.10, the main correspondence type pre-condition $B\text{-to-}PE$ of type C is different from the additional $M\text{-to-}M$ of one type C , and there is only one mapping of the leaf hyperedge of type M of the additional target sub-tree with the type described in condition 2 of Definition 5.3.1.10. Thus, the considered non-local GBMT rule 12 complies with Definition 5.3.1.10. For all other non-local rules of our ADR-to-CSP GBMT the compliance with Definition 5.3.1.10 can be shown in the same way. They all have the six parts described above and their correspondence functions have the desired properties.

Additionally to the individual GBMT rule types, we have also formalized the definitions of GBMTs containing them. Now, we need to show that our complete ADR-to-CSP GBMT, that contains all three types of rules, is conform with these definitions. Since the definitions are nested and see the basic and extended rules as special cases of non-local rules, we can consider Definition 5.3.1.11 for GBMTs with non-local rules directly.

Definition 5.3.1.11 describes a GBMT as a finite set of extended and non-local rules that fulfills its conditions 1 and 2. Our ADR-to-CSP GBMT fulfills condition 1, since all of its non-local rules (tr) use the additional correspondence type pre-condition $M\text{-to-}M$ of type C (c_{pre}^a), that is first created by basic rule 1 (tr' with $c'_{pre} \neq c_{pre}^a$ and $c' \in C'_{post}$ of type c_{pre}^a) of the transformation. It also fulfills condition 2, since there is no rule (tr') in our ADR-to-CSP GBMT that produces two or more correspondence mappings of type $M\text{-to-}M$ with the correspondence type C (no distinct $c'_1, c'_2 \in C'_{post}$ of type c_{pre}^a).

Now we need to consider the conditions of Definition 5.3.1.9 for extended GBMTs, that our ADR-to-CSP GBMT also has to fulfill by Definition 5.3.1.11. Definition 5.3.1.9

also states two conditions on a GBMT. Our ADR-to-CSP GBMT fulfills condition 1, since all its rules have the same priority, and their source sub-trees are distinct and none of them is a prefix of the other (see Section 5.2.1). Its second condition 2 is more complex and requires a more detailed consideration.

This condition requires our ADR-to-CSP GBMT to contain exactly one well-formed rule set $(RS_{p_s, c_{pre}})$ for each combination of a source production $(p_s = (A_s, R_s))$ and a type of correspondence mapping (c_{pre}) , that a hyperedge of the type of its left-hand side (A_s) has in that GBMT definition. Since the rules of our GBMT only have simple source sub-trees with one production, by Definition 5.3.1.8, such rule sets in our example only contain single and, thus, special rules. We do not have to consider any replacement alternatives here. We just need to show that there is exactly one of such special rules for each combination (p_s, c_{pre}) in the GBMT.

In our ADR-to-CSP GBMT there are the following types of correspondence mappings: the initial mapping type *S-to-S* with the correspondence type C_d ; the mapping types *M-to-M*, *A-to-P*, *AD-to-PE*, and *B-to-PE* all with the correspondence type C . There is exactly one rule for each combination of these types with each source production applicable to its source non-terminal (see Section 5.2.1). Thus, our ADR-to-CSP GBMT fulfills condition 2 of Definition 5.3.1.9.

Since the basic GBMT rule type is a special case of the extended one, we have shown all the conditions required by Definition 5.3.1.11 for non-local GBMTs of our ADR-to-CSP GBMT and we have also shown the ACCC condition (Definition 5.3.1.12) for it. Later, in Section 5.4, we prove, amongst others, that non-local GBMTs with these characteristics have the desired quality properties by construction. First, however, we return to the formalization and discuss the execution of a GBMT containing all different types of rules.

5.3.2 Formalization of Transformation Execution

Now, we discuss the execution steps of a GBMT defined between the languages of two HR grammars G_s and G_t . It performs a transformation of an input source model M_s on its derivation tree over productions from P_s of G_s . Therefore, before such a GBMT can be executed, we need to obtain this source derivation tree $t_{M_s} \in TREE(P_s)$ by parsing M_s . Since an input source model can potentially have more than one derivation tree in G_s , we need to choose the one to use in the execution in a way, that lets us avoid non-deterministic results. Thus, we use so-called *leftmost derivation* of a source model M_s and its tree representation as an input for our GBMTs.

The notion of the leftmost derivation for hypergraphs used here is analogous to the one for strings (see Section 5.1.1). Since hyperedges of a hypergraph have no natural ordering such as that of symbols in a string, we impose such an ordering on hypergraphs as suggested in [Pra71]. For a hypergraph H we define an *ordering* $O_H = (e_1, \dots, e_n)$ such that e_i are distinct hyperedges in E_H^N and $n = |E_H^N|$. For an HR grammar G , for

each production $p = (A, R)$ in P we define an ordering O_R for its hypergraph R . Now, we can formally define *leftmost direct derivation* and *leftmost derivation sequence* for an HR grammar.

Definition 5.3.2.1 (Leftmost Direct Derivation). *Given a hypergraph H with an ordering O_H for E_H^N and a hypergraph K with an ordering O_K for E_K^N . Then K is a leftmost direct derived from H iff:*

1. $H \Rightarrow_p K$ by rewriting the leftmost non-terminal hyperedge $e_1 \in E_H^N$ in the ordering O_H ;
2. O_K is derived from O_H by substituting for $e_1 \in O_H$ the sequence O_R of the hyperedges E_R^N introduced into K by the rule $p \in P$.

To distinguish from a normal direct derivation, we denote a leftmost direct derivation by $\xRightarrow{l}$. Now, we can define a leftmost derivation sequence analogous to the definition of a normal derivation (Definition 5.3.0.5),

Definition 5.3.2.2 (Leftmost Derivation). *A sequence of leftmost direct derivations $d = H_0 \xRightarrow{l} \dots \xRightarrow{l} H_k$ is called a leftmost derivation of length $k \geq 0$ from H_0 to H_k and denoted by $H_0 \xRightarrow{l^*}_P H_k$ or $H_0 \xRightarrow{l^*} H_k$.*

For the transformation execution, we use the tree-based representation of such a leftmost derivation (*leftmost derivation tree*) of the input source model to be transformed. Source HR grammars, where an input model can have more than one leftmost derivation, would potentially compromise determinism of the developed transformation through this ambiguity. Thus, we only consider so-called *unambiguous* HR grammars as source grammars for our GBMT development method. Formally, this grammar type is defined as follows.

Definition 5.3.2.3 (Unambiguous Graph Grammar). *A graph grammar G is unambiguous if for each hypergraph (model) $M \in L(G)$ there exists exactly one leftmost derivation $S^\bullet \xRightarrow{l^*}_P M$.*

We prepare for the GBMT execution by parsing the input source model M_s using the given unambiguous source grammar G_s . If the parsing was not successful, then M_s is not in the language of G_s and there is no leftmost derivation for M_s in P_s . Then, the process terminates and returns *non-transformable* for this input source model indicating that it is out of scope and will not be transformed. This way, we ensure that only models from the language defined by our source grammar G_s are transformed.

When the parsing was successful, we obtain the leftmost derivation tree t_{M_s} for our M_s to use during the transformation execution. This tree is related to the non-terminal hyperedge e_s of the source start hypergraph $S_s^\bullet$. To make this connection explicit, we add a

production p_s^ϵ , that produces $S_s^\bullet$ such that $p_s^\epsilon = (\epsilon, S_s^\bullet)$, to the source productions creating $P'_s = P_s \cup \{p_s^\epsilon\}$. Here, ϵ denotes an empty hypergraph. Then, we build the *extended source derivation tree* $t'_{M_s} = (\epsilon, S_s^\bullet, \text{branch}'_s)$ such that $\text{branch}'_s(e_s) = t_{M_s}$. This way, e_s is explicitly related with t_{M_s} via its *branch* mapping like all non-terminal hyperedges in the tree $(E_{t_{M_s}}^{N_s})$. This extension allows us to describe the GBMT execution in a uniform way while keeping the original definition of a derivation tree (see Definition 5.3.1.4).

On the target side, we also extend the target productions P_t with an analogous p_t^ϵ production creating P'_t , and build the initial *extended target derivation tree* $t'_{M_t} = (\epsilon, S_t^\bullet, \emptyset)$, that produces $S_t^\bullet$ with a hyperedge e_t .

Finally, we create an *initial correspondence* mapping c^d , we have previously discussed, connecting hyperedges e_s and e_t of the start hypergraphs such that $c^d = (e_t, (t_c^{\text{default}}, e_s))$. This correspondence together with both derivation trees t'_{M_s} and t'_{M_t} forms the initial state of the *tree correspondence structure (TCS)*, that our transformation operates on. With each GBMT rule application, it creates new sub-tree(s) in the extended target derivation tree and adds correspondence mappings between non-terminal hyperedges of the derivation trees to the TCS structure. Formally, this structure is defined as follows.

Definition 5.3.2.4 (Tree Correspondence Structure). *A tree correspondence structure (TCS) $tcs(M_s) = (t'_{M_s}, C, t'_{M_t})$ created during an execution of a GBMT T on an input source model M_s ⁸ consists of an extended source derivation tree $t'_{M_s} \in \text{TREE}(P'_s)$ for M_s , an extended target derivation tree $t'_{M_t} \in \text{TREE}(P'_t)$ for the generated output target model M_t , and a set of correspondence mappings $C \subseteq (E_{t'_{M_t}}^{N_t} \times (CT \times E_{t'_{M_s}}^{N_s}))^*$ between the non-terminal hyperedges of t'_{M_t} and t'_{M_s} .*

To apply GBMT rules, we traverse the t'_{M_s} part of $TCS(M_s)$ to find the next non-terminal hyperedge e_s , that is in one or more correspondence mapping(s) $c = (e_t, (t_s, e_s)) \in C$ with some hyperedges e_t from $E_{t'_{M_t}}^{N_t}$. Due to the context-freeness of HR grammars, this traversal can be done in any order. Next, we consider each of these correspondence mappings c separately and find a set of transformation rules $RS_{p_s, c_{pre}}$ such that its correspondence type pre-condition c_{pre} is equal to the type of c and p_s is the first production in the sub-tree assigned to e_s through $\text{branch}'_s(e_s)$. Finally, we select a transformation rule tr with the highest priority from the set $RS_{p_s, c_{pre}}$ such that its source derivation sub-tree t_s forms a prefix of (or equals) the sub-tree assigned to e_s in t'_{M_s} via branch'_s . This rule is said to be *applicable* to the combination of a hyperedge e_s (and its sub-tree in t'_{M_s}) and a correspondence mapping c in C . Formally, the GBMT rule applicability is defined as follows.

Definition 5.3.2.5 (GBMT Rule Applicability). *A GBMT⁹ rule tr is applicable in TCS $tcs_T(M_s) = (t'_{M_s}, C, t'_{M_t})$ to a combination of a hyperedge e_s such that $e_s \in E_{t'_{M_s}}^{N_s}$*

⁸From now on, we shorten 'execution on leftmost derivation of M_s ' to 'execution on M_s ' for simplification.

⁹From now on, for brevity, we refer to a (non-local) extended GBMT simply as a GBMT.

and a correspondence mapping $c \in C$ such that $c = (e_t, (t_{ts}, e_s))$, if its c_{pre} is of the type of c (see Definition 5.3.1.2) and its t_s forms a prefix of (or equals) the sub-tree assigned to e_s via the mapping $branch'_s(e_s)$ in t'_{M_s} . If tr is non-local, there also exists a correspondence mapping $c^a = (e_t^a, (t_{ts}^a, e_s^a)) \in C$ of type c_{pre}^a with either:

- a) $branch'_t(e_t^a) \in N_t$ or
- b) for all $e'_t \in E_{R_{branch'_t(e_t^a)}}^{N_t}$ set $\{c' = (e'_t, (t'_{ts}, e'_s)) \in C \mid c' \text{ of type } c_{pre}^a\} = \emptyset$ in $tcs_T(M_s)$.

An application of a GBMT rule tr is then performed by assigning its target derivation sub-tree t_t via $branch'_t$ mapping of t'_{M_t} to the hyperedge e_t of the correspondence mapping c . This creates a new sub-tree at $branch'_t(e_t) = t_t$ in the extended target derivation tree t'_{M_t} . Additionally, all correspondence mappings defined in tr by its function C_{post} are created between the corresponding hyperedges of the newly added $branch'_t(e_t)$ and the hyperedges of the sub-tree $branch'_s(e_s)$, which are at the same positions as hyperedges in t_s used in C_{post} . If tr is non-local, an additional sub-tree t_t^a is also added at e_t^a of the additional correspondence mapping c^a via $branch'_t$. Depending on the case from Definition 5.3.2.5, e_t^a is either directly mapped to t_t^a via $branch'_t$ (a) or the already existing $branch'_t(e_t^a)$ sub-tree is restructured to add t_t^a (b). Both alternatives are described in detail below as the steps of the execution process.

A GBMT execution terminates when each combination of a hyperedge from t'_{M_s} and a correspondence mapping $c \in C$ involving it in the TCS of the transformation has been considered, and the suitable rule has been applied. The correspondence mappings for each non-terminal hyperedge e_s are considered in a specific order defined based on the orderings of the individual elements in their hypergraphs.

A mapping $c = (e_t, (t_{ts}, e_s)) \in C$ is placed *before* $c' = (e'_t, (t'_{ts}, e'_s)) \in C$ in the correspondence ordering relation $O^c - O^c(c) < O^c(c') -$ when $O(e_t) < O(e'_t)$, or when $O(e_t) = O(e'_t)$ and $O^t(t_{ts}) < O^t(t'_{ts})$. Where the ordering O of e_t and e'_t is defined based on the ordering O transferred from P_t to $result(t'_{M_t})$ in t'_{M_t} created up to this application step, and the ordering O^t of $t_c \in CT$ is an arbitrary fixed ordering defined on CT of T . Formally, the complete GBMT execution process is defined as follows.

Definition 5.3.2.6 (Grammar Based Model Transformation Execution). *An execution of a GBMT T between $L(G_s)$ and $L(G_t)$ on a finite input source model $M_s \in L(G_s)$ that produces a target derivation tree t_{M_t} and, thereby, an output target model $M_t = result(t_{M_t}) \in L(G_t)$ consists of the following steps:*

1. Extend t_{M_s} to $t'_{M_s} = (\epsilon, S_s^\bullet, branch'_s) \in TREE(P_s \cup \{p_s^\epsilon = (\epsilon, S_s^\bullet)\})$ with $e_s^d \in E_{S_s^\bullet}^{N_s}$ and $branch'_s(e_s^d) = t_{M_s}$. Set $t'_{M_t} = (\epsilon, S_t^\bullet, branch'_t) \in TREE(P_t \cup \{p_t^\epsilon = (\epsilon, S_t^\bullet)\})$ with $e_t^d \in E_{S_t^\bullet}^{N_t}$ and $branch'_t(e_t^d) = lab(e_t^d)$, and initialize the set of correspondence mappings $C = \{c^d = (e_t^d, (t_c^{default}, e_s^d))\}$ with $t_c^{default} \in CT$.

Construct TCS of T such that $tcs_T(M_s) = (t'_{M_s}, C, t'_{M_t})$.

2. Traverse t'_{M_s} in depth-first (alternatively breadth-first) manner visiting its non-terminal hyperedges $e_s \in E_{t'_{M_s}}^{N_s}$.

- a) Consider all correspondence mappings $c = (e_t, (t_{ts}, e_s))$ in C for e_s in the ordering O^c . Find a rule $tr \in T$ applicable (Definition 5.3.2.5) to a e_s, c combination. If tr is non-local (Definition 5.3.1.10), find a correspondence mapping $c^a = (e_t^a, (t_{ts}^a, e_s^a))$ in C of type c_{pre}^a fulfilling a or b of Definition 5.3.2.5.

When tr and, if applicable, c^a were found:

b) Apply tr :

- i. Assign $branch'_t(e_t) = t_t$ in t'_{M_t} .

- ii. For each correspondence mapping $c_p = (e_t^p, (t_{ts}^p, e_s^p))$ in C_{post} such that $e_t^p \in E_{t_t}^{N_t}$, add a correspondence mapping $c'_p = (e_t^p, (t_{ts}^p, e_s'^p))$ to $C = C \cup \{c'_p\}$ with $e_t^p \in E_{branch'_t(e_t)}^{N_t}$ added to t'_{M_t} within the sub-tree t_t in step 2(b)i, and hyperedge $e_s'^p \in E_{t'_{M_s}}^{N_s}$ is at the same position in $branch'_s(e_s)$ as e_s^p in the t_s of rule tr .

c) If tr is non-local:

- i. If $branch'_t(e_t^a) \notin N_t$ (b), find $e \in E_{t_t^a}^{N_t}$ with hyperedge identity mapping to $root(t_t^a)$, and assign $branch_t(e) = branch'_t(e_t^a)$ in t_t^a .

- ii. Assign $branch'_t(e_t^a) = t_t^a$.

- iii. For each correspondence mapping $c_p = (e_t^p, (t_{ts}^p, e_s^p))$ in C_{post} such that $e_t^p \in E_{t_t^a}^{N_t}$ add a correspondence mapping $c'_p = (e_t^p, (t_{ts}^p, e_s'^p))$ to $C = C \cup \{c'_p\}$, where $e_t^p \in E_{branch'_t(e_t^a)}^{N_t}$ is added to t'_{M_t} within the sub-tree t_t^a in step 2(c)ii and $e_s'^p \in E_{t'_{M_s}}^{N_s}$ is at the same position in $branch'_s(e_s)$ as e_s^p in t_s of rule tr , or $e_s'^p$ equals e_s^a .

3. If all e_s have been visited, return $branch'_t(e)$ sub-tree of t'_{M_t} where $e \in E_{S_t}^{N_t}$ and terminate.

The returned target derivation tree $t_{M_t} = branch'_t(e)$ contains the sought target model M_t , that can be extracted as the final output through a simple post-processing step flattening the derivation tree. Later, we also refer to this returned result as $T(M_s)$.

To increase the understandability of the execution process we have described, we now provide one possible implementation of it in pseudo-code (see Algorithm 1) without repeating the full description. This implementation consists of a main part and a recursively called procedure. It traverses the extended leftmost source derivation tree of an input source model in the depth-first manner to apply GBMT rules. The basic steps of the algorithm like rule finding and application are the same for any implementation.

Algorithm 1: GBMT execution in pseudo-code

Input: T – GBMT,
 t_{M_s} – leftmost derivation tree of an input source model $M_s \in L(G_s)$.
Output: t_{M_t} – derivation tree of an output target model M_t .

```

1  $t'_{M_s} := (\epsilon, S_s^\bullet, branch'_s)$ ; //create an extended source derivation tree
2  $branch'_s(e_s^d) := t_{M_s}$  for  $e_s^d \in E_{S_s^\bullet}^{N_s}$ ;
3  $t'_{M_t} := (\epsilon, S_t^\bullet, branch'_t)$ ; //create an extended target derivation tree
4  $branch'_t(e_t^d) := lab(e_t^d)$  for  $e_t^d \in E_{S_t^\bullet}^{N_t}$ ;
5 //initialize  $C$ 
6  $C := \{c^d = (e_t^d, (t_c^{default}, e_s^d))\}$  with  $t_c^{default} \in CT$ ;
7  $tcs_T(M_s) := (t'_{M_s}, C, t'_{M_t})$ ; //construct TCS of  $T$  for  $M_s$ 

8 //start the recursive depth-first tree traversal of  $t'_{M_s}$  to apply GBMT rules
9  $visitSubTrees(t'_{M_s}, tcs_T(M_s))$ ;

10 return  $branch'_t(e)$  of  $t'_{M_t}$  for  $e \in E_{S_t^\bullet}^{N_t}$ ;
```

The procedure `visitSubTrees` listed below is called from the main part to visit all non-terminal hyperedges of the extended source derivation tree t'_{M_s} starting with the hyperedge labeled by the start symbol S_s . It takes a source derivation tree t_s and a TCS $tcs_T(M_s)$ of the GBMT T on M_s , and visits all non-terminal hyperedges of this tree in a depth-first manner. Each visited hyperedge e_s is considered in combination with all of its correspondence mappings stored in the C part of the $tcs_T(M_s)$ structure. For each such combination, we find an applicable rule tr in T and apply it constructing step-by-step the target derivation tree t'_{M_t} , assigning $branch'_t$ mappings, and adding new correspondence mappings into C .

In the case of a non-local rule, we find the required additional correspondence mapping c^a and add the sub-tree t_t^a to its first argument e_t^a either directly as $branch'_t(e_t^a)$ (when a) of Definition 5.3.2.5 holds) or by re-structuring the existing $branch'_t(e_t^a)$ (when b) of Definition 5.3.2.5 holds). Further correspondence mappings from C_{post} defined for the hyperedges from t_t^a are also added to C .

Up to now, we have formalized both the description and the execution of GBMTs. Along the way, we mentioned how the conditions imposed in the formalization affect the properties of the developed GBMTs. Next, we formally define these properties for GBMTs and prove that they hold by construction based on these conditions.

Procedure visitSubTrees

Input: $t_s = (A_s, R_s, \text{branch}_s) \in \text{TREE}(P_s \cup \{p_s^\epsilon\})$, $tcs_T = (t'_{M_s}, C, t'_{M_t}) \in \text{TCS}$

```

1 foreach  $e_s \in E_{R_s}^{N_s}$  do
2   foreach  $c = (e_t, (e_{ts}, e_s)) \in C$  for  $e_s \wedge c$  is next in  $O^c$  do
3     Find  $tr \in T$  applicable (Definition 5.3.2.5) to  $e_s, c$  combination;
4     if  $tr$  is non-local (Definition 5.3.1.10) then
5       Find  $c^a = (e_t^a, (t_{ts}^a, e_s^a)) \in C$  of type  $c_{pre}^a$  : a or b of Definition 5.3.2.5
6       holds;
7       if  $tr$  is found  $\wedge c^a$  is found (when applicable) then
8          $\text{branch}'_t(e_t) := t_t$ ;
9         foreach  $c_p = (e_t^p, (t_{ts}^p, e_s^p)) \in C_{post} \wedge e_t^p \in E_{t_t}^{N_t}$  do
10           $c'_p := (e_t^p, (t_{ts}^p, e_s'^p))$  where  $e_t^p \in E_{\text{branch}'_t(e_t)}^{N_t}$  and  $e_s'^p \in E_{t'_{M_s}}^{N_s}$  is at the
11          same position in  $\text{branch}'_s(e_s)$  as  $e_s^p$  in  $t_s$ ;
12           $C := C \cup \{c'_p\}$ ;
13          if  $tr$  is non-local then
14            if  $\text{branch}'_t(e_t^a) \notin N_t$  (case b) then
15              Find  $e \in E_{t_t^a}^{N_t}$  with the hyperedge identity mapping to  $\text{root}(t_t^a)$ ;
16               $\text{branch}_t(e) := \text{branch}'_t(e_t^a)$ ;
17               $\text{branch}'_t(e_t^a) := t_t^a$ ;
18              foreach  $c_p = (e_t^p, (t_{ts}^p, e_s^p)) \in C_{post} \wedge e_t^p \in E_{t_t^a}^{N_t}$  do
19                 $c'_p := (e_t^p, (t_{ts}^p, e_s'^p))$  where  $e_t^p \in E_{\text{branch}'_t(e_t^a)}^{N_t}$  and  $e_s'^p \in E_{t'_{M_s}}^{N_s}$  is at
20                the same position in  $\text{branch}'_s(e_s)$  as  $e_s^p$  in  $t_s$  or  $e_s'^p = e_s^a$ ;
21                 $C := C \cup \{c'_p\}$ ;
22           $\text{visitSubTrees}(\text{branch}'_s(e_s), tcs_T)$ ;

```

5.4 Guaranteed Quality Properties of Transformations

Here, we address important quality properties of the GBMTs developed using our approach and compliant with the definitions and conditions described in Section 5.3.1, like our ADR-to-CSP GBMT (see end of Section 5.3.1). These properties, introduced in Section 4.2.1, are *termination*, *soundness* and *completeness*, and *determinism*. For each of these properties we first provide an informal description and a basic justification for why it holds for GBMTs, and then provide a complete formal definition and a full detailed proof. Each of the provided proofs highlights the conditions we put on the definition of GBMTs in order to guarantee these properties by construction.

As in Section 5.3, in the following text we refer to the source and target HR grammars of a GBMT as G_s and G_t , to an input source model as M_s , to an output target model as M_t , and to the GBMT itself as T . Since we only execute model transformations on finite input models that were successfully parsed in G_s , belong to $L(G_s)$, and have a leftmost derivation tree t_{M_s} in G_s , we only consider them in our proofs. We start by considering the *termination* property.

5.4.1 Termination

The *termination* property for a GBMT T means that for any M_s , T is executed in a finite number of steps and each of the steps is finite.

As described before, T is executed on the leftmost derivation tree t_{M_s} of M_s . We traverse this tree in some pre-defined order and visit each of its non-terminal hyperedges that has correspondences, including the initial one (see Section 5.2.1). A finite number of productions (finite target derivation sub-trees) of G_t is applied by T for every finite derivation sub-tree formed by productions of G_s in t_{M_s} and a correspondence c of the hyperedge mapped onto it. Since these derivation sub-trees are finite, and the set of created correspondences (shown later) and t_{M_s} are *finite*, the whole execution process is also guaranteed to have a finite number of finite steps.

Now, we formalize the discussed property and give its proof for GBMTs.

Definition 5.4.1.1 (Termination). *A GBMT T terminates iff any execution of T on any input source model $M_s \in L(G_s)$ is performed in a finite number of finite steps.*

To prove that a GBMT T terminates in any execution, we first introduce a supporting Lemma . It shows that the set C of the TCS constructed during an execution of T is always finite.

Lemma 1 (Finite Correspondence Mappings Set). *Let T be a GBMT executed on an input source model $M_s \in L(G_s)$. Then, a TCS $tcs_T(M_s)$ constructed during an execution always yields a finite set of correspondence mappings in C .*

Proof. Consider the execution steps in Definition 5.3.2.6 starting with the initialization of C . Step 1, executed once, adds one initial mapping $c^d = (e_t^d, (t_c^{default}, e_s^d))$ to the empty C , keeping it finite. We prove that C is always finite by induction. Set $C^0 = \{c^d\}$ is finite. Assume by induction C^i constructed in the i -th iteration of step 2 is finite.

Consider the $(i+1)$ -th iteration of step 2 and C^{i+1} constructed in it. This iteration considers up to $|C^{i+1}|$ combinations of e_s with $c \in C^{i+1}$ in step 2a.

Each of the iterations of step 2a searches for an applicable rule $tr \in T$ and, when it is non-local, an additional correspondence $c^a \in C^i$. The search for tr is done in a finite number of steps, since by Definition 5.3.1.9, T is finite. The search for

c^a is done in a finite number of steps, since by induction C^i is finite. Finally, the search for a hyperedge e in step 2(c)i is done in a finite number of steps, since by Definition 5.3.1.10, t_t^a is finite.

When the found rule tr is applied in an iteration of step 2a, new correspondence mappings are added to C^{i+1} in steps 2(b)ii and 2(c)iii (if tr is non-local). Otherwise, C^{i+1} stays unchanged. Together, steps 2(b)ii and 2(c)iii add $|C_{post}|$ correspondence mappings to C^{i+1} in one iteration. By Definition 5.3.1.10, for the most extended version of C_{post} , derivation trees t_s , t_t , and t_t^a are finite $\Rightarrow C_{post}$ defined over them is always finite. Therefore, an iteration of step 2a takes a finite number of steps.

By Definition 5.3.1.5 and 5.3.1.10 of GBMT rules, none of the mappings added in steps 2(b)ii and 2(c)iii can contain the current e_s . Thus, $(i+1)$ -th iteration of step 2 only considers up to $|C^i|$ combinations of e_s with $c \in C^i$. Therefore, in this iteration, maximum $|C^i|$ times $\max_{tr \in T} |C_{post}^{tr}|$ correspondence mappings are added to C^i to construct C^{i+1} .

By induction C^i is finite, and, as shown before, C_{post} is always finite $\Rightarrow$ constructed C^{i+1} is finite $\Rightarrow C$ is finite after each iteration of step 2. Overall, the construction of C takes $|E_{t'_{M_s}}^{N_s}|$ iterations of step 2 $\Rightarrow$ since t_{M_s} and, hence, t'_{M_s} is finite, $|E_{t'_{M_s}}^{N_s}|$ is finite $\Rightarrow C$ is finite. $\square$

With this Lemma we can now prove termination of a GBMT. Note, that the parsing step before the transformation execution can influence termination of the complete process. For HR grammars the membership problem is decidable [Roz97] and, thus, parsing of an input model always terminates. Now, we just need to show that the execution process also terminates.

Theorem 1 (Termination). *Let T be a GBMT. Then, for each $M_s \in L(G_s)$ – input source model – execution of T on M_s is performed in a finite number of finite steps.*

Proof. To prove that T always terminates, we show that each step in Definition 5.3.2.6 has a finite number of iterations and sub-steps, and that the minimal sub-steps are finite. Since assignment and return steps are trivially finite, we do not consider them in this proof.

Consider step 2 and its sub-steps. It traverses an extended leftmost derivation tree t'_{M_s} of M_s that is finite $\Rightarrow$ step 2 has a finite number of iterations i.e. $|E_{t'_{M_s}}^{N_s}|$. Sub-step 2a is performed in each iteration of step 2 for a non-terminal hyperedge $e_s \in E_{t'_{M_s}}^{N_s}$ and consists of a maximum of $|C|$ iterations.

By Lemma 1, C is finite and an iteration of sub-step 2a has a finite number of finite steps $\Rightarrow$ step 2a has a finite number of finite steps for each e_s . Hence, step 2 has a finite number of iterations consisting of a finite number of finite steps $\Rightarrow T$ is always executed in a finite number of finite steps $\Rightarrow T$ terminates on any M_s . $\square$

5.4.2 Soundness and Completeness

The next question we want to address is, whether a GBMT T developed using our approach *can transform* any *valid input model* M_s , and whether the transformation result $T(M_s)$ is a *valid output model*. The described properties are called *completeness* and *soundness*, respectively (see Section 4.2 for details).

In order to consider these properties, we first need to define the notions of a *valid input source* and a *valid output target* model that we use in the context of GBMTs.

Definition 5.4.2.1 (Valid Input source Model). *A model M_s is a valid input source model for a GBMT T if it belongs to the language defined by the source grammar G_s of T – $M_s \in L(G_s)$.*

Definition 5.4.2.2 (Valid Output Target Model). *A model M_t is a valid output target model for a GBMT T if it belongs to the language defined by the target grammar G_t of T – $M_t \in L(G_t)$.*

Furthermore, we define when a GBMT *can transform* an input source model as follows.

Definition 5.4.2.3 (Ability to Transform). *A GBMT T can transform an input model M_s , or M_s is transformable by T , when none of the execution process steps of T from Definition 5.3.2.6 fail on M_s .*

For completeness, we have to show that if $M_s \in \mathcal{L}(G_s)$, an execution of T on such M_s will not fail. Essentially, this means that the step 2a of Definition 5.3.2.6, where an applicable GBMT rule (tr) and an additional mapping (c^a) for it (for non-local rules) is found, has to always succeed for each combination of a source non-terminal hyperedge (e_s) and its correspondence (c).

In Section 5.3.1, we have demanded that T contains a GBMT rule for *every production* in G_s and *every correspondence type* an edge rewritten by it might have. The ACCC ensures the applicability in the case when such rules are non-local. Hence, we can transform every production application (direct derivation) in t_{M_s} into a derivation subtree in the generated target derivation tree t_{M_t} . Next, we look at the generation of the target model M_t via its derivation tree t_{M_t} .

For the completeness of a GBMT T , we need to show that the generation of t_{M_t} by T does not fail, i.e., that all target productions are applicable at the places where the transformation wants to apply them. Both G_s and G_t are *context-free*, so the existence of the non-terminal hyperedges ensured by the correspondences is all that is needed to ensure the applicability of the target productions. When considering the next source production application, we apply the related target productions to the non-terminal hyperedge corresponding to the hyperedge rewritten by the first source production. Therefore, since the source productions are applicable and T contains GBMT rules for every type of correspondence that a non-terminal hyperedge rewritten by it might have, the target

productions are applicable too. Thus, T always returns a model for a valid M_s and thus, T is complete.

As for the soundness of a GBMT T , we need to show that M_t is a valid output model – $M_t \in \mathcal{L}(G_t)$. This can be reduced to the question whether the generated derivation tree of the target model t_{M_t} is complete, meaning that no non-rewritten non-terminal hyperedges are left after the application of a GBMT T . From the requirements in Section 5.3.1, every non-terminal hyperedge produced (and not consumed) by target sub-tree(s) is *linked by a correspondence* to a non-terminal hyperedge in t_{M_s} . And since $M_s \in \mathcal{L}(G_s)$, all source non-terminal hyperedges produced are eventually rewritten. This implies, since T is complete, that all leaf target non-terminal hyperedges produced are also eventually rewritten by the related target productions, i.e., t_{M_t} is complete. Thus, we have $M_t \in \mathcal{L}(G_t)$ and T is sound.

Next, we formally define the notions of completeness and soundness and provide the detailed proofs of these properties for GBMTs.

Definition 5.4.2.4 (Completeness). *A GBMT T is complete iff it can transform any valid input source model M_s .*

Definition 5.4.2.5 (Soundness). *A GBMT T is sound iff its result on a valid input source model M_s – $T(M_s)$ – is a valid output target model.*

Now, we prove these properties for a GBMT T starting with the proof of completeness. More precisely, we start with a supporting Lemma for non-local GBMT rules which is used later in the proofs of completeness and determinism. This Lemma relies on the additional ACCC condition we have formulated in Definition 5.3.1.12.

Lemma 2 (Unique Additional Correspondence). *Let T be a non-local GBMT fulfilling ACCC (Definition 5.3.1.12) and M_s an input source model. Then, in each execution of T on M_s , for each non-local $tr \in T$, when tr is applied, C of $TCS_{TCS}(M_s)$ contains exactly one and the same correspondence mapping of type c_{pre}^a fulfilling condition a or b of Definition 5.3.2.5.*

Proof. As T is non-local, by Definition 5.3.1.11 (1), for each non-local GBMT rule $tr = (t_s, t_t, t_t^a, c_{pre}, c_{pre}^a, C_{post}) \in T$, exists a rule $tr' \in T$ with $c_{pre}' \neq c_{pre}^a$ and $c' \in C_{post}'$ of type c_{pre}^a . T also fulfills ACCC $\Rightarrow$ only one of such tr' is *always applied exactly once and before* tr during an execution of T on an M_s .

Hence, by Definition 5.3.2.6, tr' always adds one correspondence mapping c_p' (in step 2(b)ii or 2(c)iii) of type c_{pre}^a to C before tr is applied. By Definition 5.3.1.11 (2), there is no $tr' \in T$ that creates more than one correspondence mapping $c' \in C_{post}'$ of type $c_{pre}^a \Rightarrow tr'$ creates *only once and exactly one* such c_p' before tr is applied. Thus, other rules tr'' applied between tr' and tr can at most consume ($c_{pre}'' = c_{pre}^a$ and no $c'' \in C_{post}''$ of type c_{pre}^a) or re-create ($c_{pre}'' = c_{pre}^a$ and there exists exactly one $c'' \in C_{post}''$ of type c_{pre}^a) this type of mapping.

Consider the following alternatives for $c_p' = (e_t^p, (t_{ts}^p, e_s'^p))$ created by tr' :

1. c'_p is not used by any $tr'' \Rightarrow branch'_t(e_t^p) \in N_t \Rightarrow$ Definition 5.3.2.5 (a) holds for c'_p ;
2. c'_p is consumed by tr'' with $c''_{pre} = c_{pre}^a$ and no $c'' \in C''_{post}$ of type $c_{pre}^a \Rightarrow$ for all $e'_t \in E_{R_{branch'_t(e_t^p)}}^{N_t}$ set $\{c' = (e'_t, (t'_{ts}, e'_s)) \in C \mid c' \text{ of type } c_{pre}^a\} = \emptyset$ in $tcs_T(M_s) \Rightarrow$ Definition 5.3.2.5 (b) holds for c'_p ;
3. c'_p is re-created by tr'' with $c''_{pre} = c_{pre}^a$ and there exists exactly one $c'' \in C''_{post}$ of type $c_{pre}^a \Rightarrow tr''$ creates exactly one c''_p of type c_{pre}^a . Consider c''_p as new c'_p . By Theorem 1, T terminates $\Rightarrow$ number of applied tr'' is finite $\Rightarrow$ eventually either (1) or (2) must apply to such a created c''_p .

Thus, after the application of tr' and before the application of tr , there always exists exactly one $c'_p \in C$ such that either condition (a) or (b) of Definition 5.3.2.5 holds for it.

By Definition 5.3.2.6, hyperedges e_s of t'_{M_s} and correspondence mappings for each of them are considered in the same traversal and mapping ordering O^c , respectively, in each execution of T on an M_s . Furthermore, for each combination of e_s and c there exists maximum one applicable $tr \in T$.

Assume the opposite. There exist distinct $tr_1, tr_2 \in T$ applicable to e_s and $c \Rightarrow$ by Definition 5.3.2.5, c_{pre} of tr_1 and tr_2 is the type of c , and t_{s_1} and t_{s_2} form a prefix of (or equals) $branch'_s(e_s) \Rightarrow t_{s_1}$ is a prefix of (or equals) t_{s_2} or vice versa. This contradicts Definition 5.3.1.9 (1) ($\nexists$) $\Rightarrow$ there exists at most one $tr \in T$ applicable to each e_s and c combination. Hence, the same rules, and in the same order, are applied before tr in each execution $\Rightarrow$ the same tr'' , and in the same order, are applied to $c'_p \in C$ created by $tr' \Rightarrow c'_p$, used in the application of tr , is the same in each execution of T on an M_s . $\square$

The above lemma can now be used in the next proof to show completeness of GBMTs.

Theorem 2 (Completeness). *Let T be a GBMT that, if it is non-local, fulfills ACCC (Definition 5.3.1.12). Then, each input source model $M_s \in L(G_s)$ is transformable (Definition 5.4.2.3) by T .*

Proof. To prove that T is complete we need to show that an execution of T does not fail on any $M_s \in L(G_s)$.

Consider steps of Definition 5.3.2.6. Step 1 constructs an initial TCS $tcs_T(M_s)$ for the execution of T on an M_s and only requires t_{M_s} that is always provided by the parser $\Rightarrow$ step 1 can not fail. Step 2 traverses the extended leftmost source derivation tree t'_{M_s} and can only fail in one of its sub-steps and not in the traversal $\Rightarrow$ consider its sub-steps.

Step 2a searches for a GBMT rule $tr \in T$ applicable (Definition 5.3.2.5) to a combination of e_s and c . It fails if no such tr is found in T . Assume that this is the case.

By Definition 5.3.1.9¹⁰, for all productions $p_s^* = (A_s^*, R_s^*) \in P_s$ and all correspondence mappings $c^* = (e_t^*, (t_{ts}^*, e_s^*)) \in \bigcup_{i=1}^n C_{post}^i \cup \{c^d\}$ such that $lab(e_s^*) = A_s^*$, there exists a *well-formed rule set* $RS_{p_s^*, c_{pre}^*}$ such that c^* is of the type c_{pre}^* .

Production p_s whose tree is mapped to e_s through $branch'_s$ is in P_s by the definition of a derivation tree (Definition 5.3.1.4), and $c \in C$ is either the initial mapping c^d or it was produced as c'_p in step 2(b)ii or step 2(c)iii of a rule application. In the second case, such a c'_p has the same type as some $c_p \in \bigcup_{i=1}^n C_{post}^i$ (see Definition 5.3.2.6) $\Rightarrow$ there is a c^* of the same type as c . Therefore, there exists a well-formed rule set $RS_{p_s, c_{pre}}$ for production p_s and type of c denoted by c_{pre} .

By Definition 5.3.1.8, well-formed rule set $RS_{p_s, c_{pre}}$ either contains a special rule (case 1) or its rules fulfill the coverage criteria (case 2). In case 1, the one-production source derivation tree t_s of the special rule in $RS_{p_s, c_{pre}}$ forms a prefix of (or equals) the sub-tree assigned to e_s via the mapping $branch'_s(e_s)$ in t'_{M_s} . Hence, the special rule is *applicable* to the e_s and c combination ($\not\downarrow$), even if none of the other rules in $RS_{p_s, c_{pre}}$ are and even if the special rule is non-local, by Lemma 2.

In case 2 of Definition 5.3.1.8, the following (*coverage criteria*) holds for $RS_{p_s, c_{pre}}$: for each $tr \in RS_{p_s, c_{pre}}$ and each replacement alternative t_{alt} in G_s for its source derivation sub-tree t_s , there exists a GBMT rule $tr' \in RS_{p_s, c_{pre}}$ such that $t'_s = t_{alt}$; and for all distinct $tr, tr' \in RS_{p_s, c_{pre}}$, t'_s is a replacement alternative of t_s .

As $branch'_s(e_s)$ is a sub-tree of t'_{M_s} with $result(t'_{M_s}) = M_s \in L(G_s)$ and thus, $E_{result(t'_{M_s})}^{N_s} = \emptyset \Rightarrow E_{result(branch'_s(e_s))}^{N_s} = \emptyset$. Therefore, all hyperedges $e \in E_{branch'_s(e_s)}^{N_s}$ have $branch'_s(e) \notin N_s$ making the sub-tree $branch'_s(e_s)$ non-extendable. Thus, additionally to fulfilling the coverage criteria, none of the rules in $RS_{p_s, c_{pre}}$ have t_s extending $branch'_s(e_s)$.

As $RS_{p_s, c_{pre}}$ is not empty by Definition 5.3.1.8, there exists $tr \in RS_{p_s, c_{pre}}$. Then, the sub-tree $branch'_s(e_s)$ either:

- (I) equals (or an extension of) t_s part of tr , and, if tr is non-local, by Lemma 2, there exists a suitable unique correspondence mapping $c^a \in C \Rightarrow tr$ is applicable to the e_s and c combination ($\not\downarrow$); or
- (II) equals (or an extension of) one of the replacement alternatives of t_s .

By Definition 5.3.1.8, for each replacement alternatives of t_s , there must be a GBMT rule $tr' \in RS_{p_s, c_{pre}}$ such that its t'_s is equal to this alternative. As $branch'_s(e_s)$ is a replacement alternative t_{alt} (or its extension) for t_s (II), there is a $tr' \in RS_{p_s, c_{pre}}$ for t_{alt} such that $t'_s = t_{alt}$. Hence, t'_s is equal to (or a prefix of) $branch'_s(e_s)$, and, if tr is non-local, by Lemma 2, there exists a suitable unique correspondence mapping $c^a \in C \Rightarrow tr'$ is applicable to the e_s and c combination ($\not\downarrow$).

¹⁰Star (*) is added to the names in the definition to distinguish its entities from those of the execution.

There is a rule in $RS_{p_s, c_{pre}}$ that is applicable to e_s and c in every case. By Proposition 4, $RS_{p_s, c_{pre}}$ is finite for the G_s of T and, by Definition 5.3.1.5, t_s of each rule in it is finite. By Lemma 1, C is finite, and, if T is non-local, ACCC holds for it. Therefore, the GBMT rule $tr \in RS_{p_s, c_{pre}}$ is *applicable* to the combination e_s and c and the suitable correspondence mapping c^a are found in a finite number of steps ($\downarrow$). Hence, step 2a does not fail to find a tr and, if applicable, the c^a for it.

Consider step 2(b)i. It can not fail since e_t is always available as part of $c \in C$. In step 2(b)ii new correspondence mappings $c'_p = (e_t^p, (t_{ts}^p, e_s'^p))$ are added to C . Both e_t^p and $e_s'^p$ exist – e_t^p was just added in step 2(b)i to t'_{M_t} , and $e_s'^p$ is in $branch'_s(e_s)$ at the same position as e_s^p in t_s since t_s is equal to (or a prefix of) $branch'_s(e_s)$. Hence, step 2(b)ii can not fail.

Consider step 2c for non-local rules. In sub-step 2(c)i c^a fulfills condition (b) of Definition 5.3.2.5 and, by Definition 5.3.1.10, there must exist $e_t^a \in E_{result(t_t^a)}^{N_t}$ identified via hyperedge identity mapping with $root(t_t^a)$ in $tr \Rightarrow$ required $e \in E_{t_t^a}^{N_t}$ is always found $\Rightarrow$ step 2(c)i can not fail. Sub-step 2(c)ii is analogous to step 2(b)i – c^a and t_t^a (both exist) are used instead of c and t_t , respectively. Finally, sub-step 2(c)iii is analogous to step 2(b)ii with t_t^a instead of t_t and an additional use of e_s^a as $e_s'^p$. Since c^a and, hence, e_s^a always exist at this point, if tr is non-local, step 2(c)iii can not fail $\Rightarrow$ step 2c can not fail.

The last execution step that returns the sub-tree $branch'_t(e)$ of t'_{M_t} can not fail, since $e \in E_{S_t}^{N_t}$ always exists in t'_{M_t} by construction (e_t^d added in step 1) $\Rightarrow$ execution of T can not fail on any $M_s \in L(G_s) \Rightarrow T$ is *complete*. $\square$

We have proven that each valid input source model the language $L(G_s)$ is transformable by a GBMT T for this language. Now, we prove that the result of applying T to such a model is also a valid model – a model in the target language $L(G_t)$ of T . We have already referred to this property as *soundness* (see Definition 5.4.2.5).

Theorem 3 (Soundness). *Let T be a GBMT that, if it is non-local, fulfills ACCC (Definition 5.3.1.12), and $M_s \in L(G_s)$ an input source model. Then, an execution of T on M_s returns a target derivation tree $t_{M_t} \in TREE(P_t)$ and an output target model $M_t = result(t_{M_t}) \in L(G_t)$.*

Proof. For a $t_{M_t} = branch'_t(e)$ with $e \in E_{S_t}^{N_t}$ returned after an execution of T on M_s , we need to show that $t_{M_t} \in TREE(P_t)$ and $M_t = result(t_{M_t}) \in L(G_t)$.

By Theorem 2 (Completeness), T does not fail on any $M_s \in L(G_s) \Rightarrow$ it produces an extended target derivation tree $t'_{M_t} = (A'_t, R'_t, branch'_t)$ and returns $branch'_t(e)$ for $e \in E_{S_t}^{N_t}$ in t'_{M_t} for each M_s . Consider the construction of t'_{M_t} in Definition 5.3.2.6. We show, by induction, that $t'_{M_t} \in TREE(P'_t)$ and $t_{M_t} = branch'_t(e) \in TREE(P_t)$ for $e \in E_{S_t}^{N_t}$ in t'_{M_t} .

Start of the induction: In step 1, t'_{M_t} is assigned to $(\epsilon, S_t^\bullet, \text{branch}'_t) \in \text{TREE}(P_t \cup \{p_t^\epsilon = (\epsilon, S_t^\bullet)\})$ and $\text{branch}'_t(e_t^d)$ for $e_t^d \in E_{S_t^\bullet}^{N_t}$ is assigned to $\text{lab}(e_t^d) \in N_t \Rightarrow t_{M_t}^0 \in \text{TREE}(P'_t)$ and $t_{M_t}^0 = \text{branch}'_t(e) \in \text{TREE}(P_t)$ for $e = e_t^d \in E_{S_t^\bullet}^{N_t}$ in $t_{M_t}^0$. Step 2 consists of $|E_{t'_{M_t}}^{N_s}|$ iterations of sub-step 2a that is performed a maximum of $|C|$ times per iteration.

We consider a single execution of step 2a as an induction step. Assume, by induction, that for $t_{M_t}^i$ constructed in the i -th execution of step 2a holds $t_{M_t}^i \in \text{TREE}(P'_t)$ and $t_{M_t}^i = \text{branch}'_t(e) \in \text{TREE}(P_t)$ for $e = e_t^d \in E_{S_t^\bullet}^{N_t}$ in $t_{M_t}^i$. We need to prove that, for $t_{M_t}^{i+1}$ constructed in the $(i+1)$ -th execution of step 2a also holds $t_{M_t}^{i+1} \in \text{TREE}(P'_t)$ and $t_{M_t}^{i+1} = \text{branch}'_t(e) \in \text{TREE}(P_t)$ for $e = e_t^d \in E_{S_t^\bullet}^{N_t}$ in $t_{M_t}^{i+1}$. Then, the same will hold for $t_{M_t} = \text{branch}'_t(e_t^d)$, returned at the end of the execution, which consists of a finite number of iterations of step 2a.

Consider sub-steps in the $(i+1)$ -th iteration of step 2a. Since T is complete by Theorem 2, each iteration makes the following changes to $t_{M_t}^i$ to create $t_{M_t}^{i+1}$:

1. In step 2(b)i, $\text{branch}'_t(e_t)$ of $e_t \in E_{t_{M_t}^i}^{N_t}$ is assigned to $t_t \in \text{TREE}(P_t)$. By induction, $t_{M_t}^i \in \text{TREE}(P'_t) \Rightarrow$ created $t_{M_t}^{i+1} \in \text{TREE}(P'_t)$ and $\text{branch}'_t(e)$ for $e = e_t^d \in E_{S_t^\bullet}^{N_t}$ in $t_{M_t}^{i+1}$ changes to t_t in some sub-tree $\Rightarrow \text{branch}'_t(e) \in \text{TREE}(P_t)$;
2. In step 2(c)i, $\text{branch}'_t(e_t^a) \notin N_t$ and $e \in E_{t_t^a}^{N_t}$, existing in $t_t^a \in \text{TREE}(P_t)$ by Definition 5.3.1.10, is mapped to $\text{branch}'_t(e_t^a)$ from $t_{M_t}^{i+1}$. By induction, $t_{M_t}^i = \text{branch}'_t(e) \in \text{TREE}(P_t)$ for $e = e_t^d \in E_{S_t^\bullet}^{N_t}$ in $t_{M_t}^i \Rightarrow$ by Definition 5.3.1.4, $\text{branch}'_t(e_t^a) \in \text{TREE}(P_t)$ in $t_{M_t}^i$ and step 2(b)i does not change this in $t_{M_t}^{i+1} \Rightarrow$ modified $t_t^a \in \text{TREE}(P_t)$.
3. In step 2(c)ii, branch'_t of $e_t^a \in E_{t_{M_t}^i}^{N_t}$ is assigned to $t_t^a \in \text{TREE}(P_t) \Rightarrow$ created $t_{M_t}^{i+1} \in \text{TREE}(P'_t)$ and $\text{branch}'_t(e) \in \text{TREE}(P_t)$ for $e = e_t^d \in E_{S_t^\bullet}^{N_t}$ in $t_{M_t}^{i+1}$.

Hence, we have $t_{M_t} = \text{branch}'_t(e_t^d) \in \text{TREE}(P_t)$ for the t_{M_t} returned. Next, we need to show that $M_t = \text{result}(\text{branch}'_t(e_t^d)) \in L(G_t)$. By Definition 5.3.0.7, an output model $M_t \in L(G_t) \Leftrightarrow M_t \in \mathcal{H}_{T_t}$ and there exists $S_t^\bullet \Rightarrow_{P_t}^* M_t \Leftrightarrow M_t \in \mathcal{H}_{T_t}$ (I) and, by Proposition 1, there exists $t \in \text{TREE}(P_t)$ with $\text{root}(t) = S_t$ and $\text{result}(t) = M_t$ (II).

Part I: $M_t \in \mathcal{H}_{T_t} \Leftrightarrow E_{M_t}^{N_t} = \emptyset$. Assume the opposite $\Rightarrow \exists e_t \in E_{M_t}^{N_t} \Leftrightarrow \exists e_t \in E_{t_{M_t}}^{N_t}$ with $\text{branch}_t(e_t) \in N_t$ after T has been executed on M_s . By Definition 5.3.2.6, e_t has been added to t_{M_t} in step 1 or during an application of a rule $tr \in T$ as part of its sub-tree t_t or t_t^a (non-local rule). If e_t was added in step 1 as e_t^d , mapping c^d involving it was also added to C . If e_t was added by a rule $tr \Rightarrow e_t \in E_{t_t}^{N_t} \cup E_{t_t^a}^{N_t}$ of tr , and $\text{branch}_t(e_t) \in N_t \Rightarrow e_t \in E_{\text{result}(t_t)}^{N_t} \cup E_{\text{result}(t_t^a)}^{N_t}$. As by Definition 5.3.1.5,

the function C_{post} is total, $\exists c_p \in C_{post}$ in tr such that $c_p = (e_t, (t_{ts}^p, e_s^p)) \Rightarrow$ step 2(b)ii or 2(c)iii of Definition 5.3.2.6 added correspondence mapping c_p' involving e_t and some $e_s'^p \in E_{t_{M_s}}^{N_s}$ to C when tr was applied.

Such a correspondence mapping $c = (e_t, (t_{ts}, e_s)) \in C$ (c^d for $e_t = e_t^d$ or c_p' for $e_t = e_t'^p$) is created either (Ia) before or (Ib) after step 2a was executed for e_s .

In case Ia: by Theorem 2, T is complete and does not fail on any $M_s \in L(G_s) \Rightarrow \exists tr \in T$ applicable to such e_s and c combination. By Definition 5.3.1.5, its $t_t \notin N_t \Rightarrow$ after application of tr , $branch_t'(e_t) \notin N_t$ and, thus, $branch_t(e_t) \notin N_t$ after T has been executed on M_s ($\downarrow$).

In case Ib: c is created by a non-local rule $tr \Rightarrow e_s$ of c is part of the additional correspondence mapping c^a used by tr and re-created as c . As c is created after step 2a is executed for e_s with all its mappings including $c^a \Rightarrow \exists tr \in T$ applicable to the e_s and c^a combination $\Rightarrow$ analogous to case (Ia) $branch_t(e_t^a) \notin N_t \Rightarrow$ in step 2(c)i tr sets $branch_t'(e_t) = branch_t'(e_t^a) \Rightarrow branch_t(e_t) \notin N_t$ after T has been executed on M_s ($\downarrow$). Thus, $M_t \in \mathcal{H}_{T_t}$.

Part II: We have shown that $t_{M_t} \in TREE(P_t)$. By Definition 5.3.2.6, $M_t = result(t_{M_t})$. The rule tr , applied to the initial correspondence mapping c^d involving e_t^d , must have $c_{pre} = (S_s, t_c^{default}, S_t)$. Thus, its t_t assigned to $branch_t'(e_t^d)$ has $root(t_t) = S_t \Rightarrow root(branch_t'(e_t^d)) = S_t \Rightarrow root(t_{M_t}) = S_t \Rightarrow t_{M_t}$ is a derivation tree t for M_t described in Proposition 1. Parts I and II hold $\Rightarrow M_t \in L(G_t)$. $\square$

5.4.3 Determinism

The last property we want to consider for GBMTs is *determinism*. This means that each output target model and its derivation tree are fully determined by the corresponding input source model. In order to show this property for a GBMT T , we require its source HR grammar G_s to be unambiguous (see Definition 5.3.2.3).

When the source grammar G_s is *unambiguous*, then for each input source model $M_s \in L(G_s)$ we obtain exactly one leftmost derivation tree t_{M_s} . Based on the transformation rules of T , the derivation tree t_{M_s} fully determines which rules are applied to construct the target tree t_{M_t} (target sub-derivations) and where (correspondences). This is because each production in the target tree t_{M_t} is uniquely determined by the productions in the source tree t_{M_s} , the correspondence of the source non-terminal hyperedge they rewrite, and the suitable additional correspondence for non-local rules (unique by ACCC). The target side of these correspondences uniquely determines where the related main and, if given, additional target productions of the chosen GBMT rule are applied. We, thus, obtain exactly one target tree t_{M_t} and, consequently, exactly one output target model M_t . Therefore, no two different target models can be the GBMT result for a single M_s .

Next, we formally define the notion of determinism and provide a detailed proof of this property for GBMTs.

Definition 5.4.3.1 (Determinism). *A GBMT T is deterministic if its result $T(M_s)$ on a valid input model $M_s \in L(G_s)$ is the same in each execution.*

To keep the proof of this property concise, we formulate and prove a separate lemma. This lemma shows that the set of correspondence mappings C , considered in each iteration of step 2a of an execution (Definition 5.3.2.6), remains the same throughout multiple executions of a GBMT T on a fixed input source model M_s .

Lemma 3 (Unique Correspondence Mapping Set). *Let T be a GBMT that has an unambiguous source HR grammar G_s and, if it is non-local, fulfills ACCC (Definition 5.3.1.12), and M_s an input source model. Then, the set C of the constructed $tcs_T(M_s)$ yields the same mappings in each iteration of step 2a of Definition 5.3.2.6 between the different executions of T on M_s .*

Proof. Consider execution steps in Definition 5.3.2.6. Step 1 adds the same initial mapping $c^d = (e_t^d, (t_c^{default}, e_s^d))$ to the empty C in each execution. We prove that C always yields the same mappings by induction. Induction begin: set $C^0 = \{c^d\}$ is the same in each execution. Assume, by induction, C^i constructed in the i -th iteration of step 2 is the same in each execution.

Consider construction of C^{i+1} in the $(i+1)$ -th iteration of step 2. This iteration considers combinations of a hyperedge e_s and a mapping $c \in C^i$. By induction, C^i is the same in each execution $\Rightarrow$ the $(i+1)$ -th iteration of step 2 performs sub-step 2a for the same mappings $c \in C^i$ in the same order in each execution.

As grammar G_s is unambiguous, by Definition 5.1.1.3, the leftmost derivation tree t_{M_s} for M_s is the same in each execution, and it is traversed in the same order $\Rightarrow e_s$ in the $(i+1)$ -th iteration of step 2 is the same in each execution. Thus, $(i+1)$ -th iteration is done for the same combinations of e_s and c in the same order in each execution.

As by Theorem 2, T is complete, and an applicable $tr \in T$ is always found for a combination. Assume, there exist distinct $tr_1, tr_2 \in T$ applicable to a e_s and c combination $\Rightarrow$ by Definition 5.3.2.5, c_{pre} of tr_1 and tr_2 is the type of c , and t_{s_1} and t_{s_2} form a prefix of (or equals) $branch'_s(e_s) \Rightarrow t_{s_1}$ is a prefix of (or equals) t_{s_2} or vice versa. This contradicts Definition 5.3.1.9 (1) ($\nexists$) $\Rightarrow \exists!$ $tr \in T$ applicable to each e_s and c combination $\Rightarrow$ the same tr is applied to this combination in each execution.

When tr is applied, new correspondence mappings c'_p described in C_{post} of tr are added to C^{i+1} in steps 2(b)ii and 2(c)iii (if tr is non-local). These mappings are the same in each execution since e_s , c , and tr are the same, and, by a non-local tr , by Lemma 2, each execution of T on M_s creates exactly one and the same c^a used in the application of $tr \Rightarrow$ the mappings using e_s^a of c^a are also the same in each

execution. Therefore, all mappings created by tr applied to e_s and c are the same in each execution.

Therefore, in the $(i + 1)$ -th iteration of step 2, the same c'_p mappings are added by the same rules applied to the same combinations in the same order to construct C^{i+1} in each execution $\Rightarrow C^{i+1}$ is the same in each execution. $\square$

Based on the above and other previously proven lemmas, we can now show the last property of interest for GBMTs – *determinism* (see Definition 5.4.3.1).

Theorem 4 (Determinism). *Let T be a GBMT that has an unambiguous source HR grammar G_s and, if it is non-local, fulfills ACCC (Definition 5.3.1.12). Then, for an input source model $M_s \in L(G_s)$, each execution of T on M_s returns the same result.*

Proof. We need to show that, by unambiguous G_s , for an $M_s \in L(G_s)$ any two executions of T on M_s always return the same t_{M_t} as a result. Consider the following execution steps of T described in Definition 5.3.2.6:

- In step 1, the leftmost derivation tree t_{M_s} of M_s is extended. As G_s is unambiguous, by Definition 5.3.2.3, $\exists! t_{M_s} \Rightarrow \exists! t'_{M_s}$ for M_s by construction $\Rightarrow t'_{M_t}$ is the same in each execution of T on M_s . The initial construction of t'_{M_t} is fully determined by S_t – the unique start symbol of G_t – and, therefore, the same in each execution.

The initial C is fully determined by hyperedges $e_s^d \in E_{t'_{M_s}}^{N_s}$ and $e_t^d \in E_{t'_{M_t}}^{N_t}$ labeled by unique start symbols S_s, S_t of G_s, G_t respectively, and by the unique default correspondence type $t_c^{default} \in CT$. By construction of t'_{M_s} and t'_{M_t} , these e_s^d and e_t^d always exists and are unique. Thus, initial C is the same in each execution $\Rightarrow$ initial $tcs(M_s)$ is the same in each execution of T on M_s .

- In step 2, t'_{M_s} is traversed in the same order in each execution of T .
- In step 2a:
 - Mappings $c \in C$ involving the hyperedge e_s are considered: By Lemma 3, C is the same in each iteration of step 2a in each execution of T on M_s $\Rightarrow$ each iteration of step 2a considers the same correspondence mappings in the same order in each execution of T on M_s .
 - As by Theorem 2, T is complete, for each e_s and c combination always exists an applicable $tr \in T$. As already shown in Lemma 3, $\exists! tr \in T$ applicable to one e_s, c combination.
 - By Lemma 2, there exists exactly one $c^a \in C$ for a non-local tr , and it is the same in each execution of T on M_s .
 - Thus, step 2a for a given e_s , considers the same set of $c \in C$ and finds the same $tr \in T$ and $c^a \in C$ for a given e_s and c combination in each execution of T on M_s .

- Steps 2(b)i and 2(b)ii are the same for the given e_s , c , and tr and c^a (chosen in 2a) in each execution of T on M_s .
- Step 2c:
 - In 2(c)i, t_t^a and, by Definition 5.3.1.10, unique $e \in E_{t_t^a}^{N_t}$ with a hyperedge identity mapping are fully determined by tr . Hyperedge e_t^a is fully determined by $c^a \Rightarrow$ step 2(c)i is the same for the given tr and c^a .
 - Steps 2(c)ii and 2(c)iii are the same for the given c , c^a , and tr (from 2a).
- Thus, t'_{M_t} constructed in step 2, is the same in each execution of T on M_s .
- Initialization of t'_{M_t} (step 1) and property of the start symbol S_t ($\forall p = (A, R) \in P_t : E_R^{\{S_t\}} = \emptyset \Rightarrow \exists! e \in E_{t'_{M_t}}^{\{S_t\}}$ (added in step 1) $\Rightarrow$ the last step is the same for a given t'_{M_t} in each execution of T on M_s).

Therefore, each execution of T returns the same result t_{M_t} on an input source model $M_s \Rightarrow T$ is deterministic. $\square$

We have proven that the GBMTs defined using our grammar based approach terminate, are sound and complete, and deterministic, by unambiguous source HR grammars, by construction. These properties are essential for building quality model transformations, however, unlike in our method, they are not always guaranteed by other model transformation development approaches. They are also hard to guarantee by using additional analysis tools on existing model transformations (see Section 4.2).

Now, in order to facilitate the use of our GBMT development approach and benefit from the guaranteed quality properties, one requires a suitable supporting tool or a chain of tools. We discuss this question next.

5.5 Tool Support

To support the grammar based approach to the development of model transformations, we have designed a tool chain involving the following existing state-of-the-art tools: the graph grammar parser of the AGG [Tae00] framework and the EMorF [KW12] transformation engine. Figure 5.18 shows this tool chain with the used artifacts for the case when the TGGs [Sch94] and the TGG engine EMorF are chosen as the target execution platform for the developed model transformations.

This chain requires the source and target HR grammars of the developed model transformation as its input. To model these HR grammars we have created a *meta-model for HR grammars*. Figure 5.19 shows the main elements of this HR grammar meta-model. It contains the HR grammar concepts, that have been introduced and formally defined in Sections 5.1.2 and 5.3, respectively. Thus, we do not repeat their explanation here. To model specific HR grammars like our ADR and CSP HR grammars, also described in

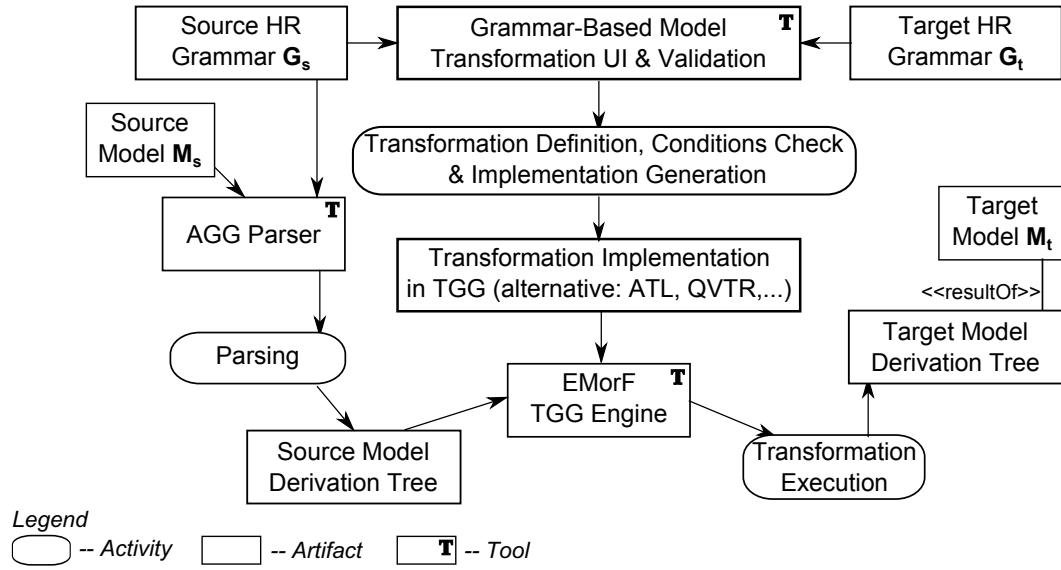


Figure 5.18: Tool support for GBMTs with TGGs/EMorF as an implementation platform

Section 5.1.2, we described them as instances of the HR grammars meta-model. Thereby, we extended the base HR grammars meta-model with the grammar-specific non-terminal and terminal types, like blocks (class *B*), activities (class *Activity*), and so on.

The specified models of the source and target HR grammars are then used as inputs for our *Grammar Based Model Transformation UI*¹¹ and *Validation* tool, that allows the definition of GBMT rules and validates their compliance with the conditions, defined in Section 5.3. This tool also makes use of the existing transformation languages and engines, like TGGs and EMorF engine, by translating the created GBMTs into their declarative, language (and engine) specific implementations. Such implementation then operates on the derivation trees of the source and (partially generated) target models.

Other combinations of transformation languages and engines, like ATL [ATL] or QVT-Relations [QVTb] with their engines, can also be used as alternatives at this implementation generation step. For this purpose, the languages need to at least support the combination of the features required to define our GBMTs based on the above meta-model for HR grammars. This combination of features¹², inspired by the feature model [CHE05] for the classification of model transformation approaches from [CH06], is shown in Figure 5.20. It includes such language features as graph patterns, syntactical typing, and parameter modes. A detailed explanation of these features can be found in [CH06]. Additionally, one can employ further translation steps, like those proposed in

¹¹ UI stands for *user interface* here.

¹² Although the static semantic typing, like well-formedness constraints, in the source domain considered optional here, its presence can be beneficial for our grammar based approach.

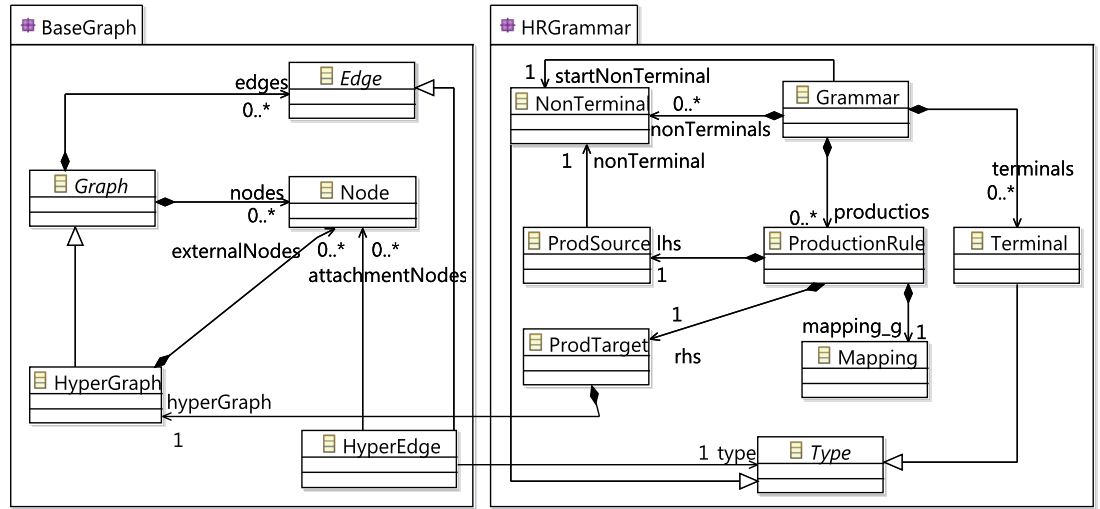


Figure 5.19: Fragment of our meta-model for HR grammars in Ecore

[GK10, ASWK11], to switch between the different model transformation implementation languages and the related platforms.

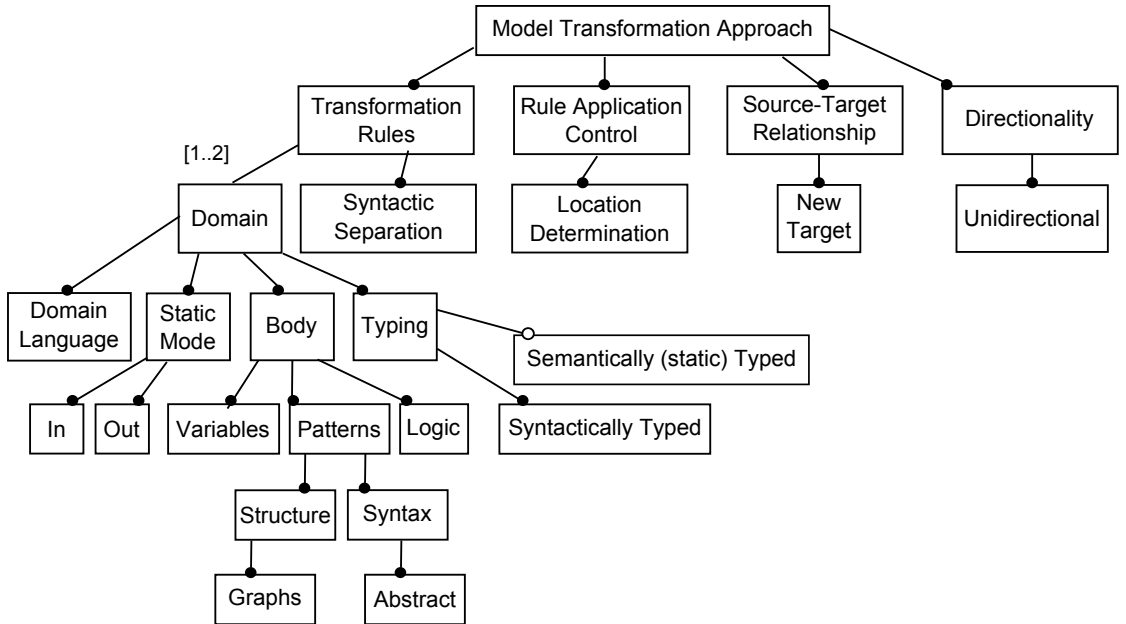


Figure 5.20: Combination of features (from [CH06]) for the meta-model based implementation of GBMTs

Next, each input source model M_s that is to be transformed, has to be represented as a hypergraph. This can be achieved through a simple object-to-hyperedge and link-to-

node conversion [MH01, MM08]. This model hypergraph is then parsed using the AGG parser with the reducing *parse grammar*, that is constructed automatically by inverting the source HR grammar of the developed model transformation. The inversion process constitutes the swap of the left- and the right-hand sides in each production of the grammar, that can be performed automatically for HR grammars.

A slight modification of the AGG parser allows us to obtain the source model derivation tree as its output, whenever the parsing was successful. This tree is compliant with the *derivation tree meta-model*, that we have also defined for our GBMT tool. The derivation tree meta-model describes a tree containing production applications. It references classes from the HR grammar meta-model, shown in Figure 5.19, to re-use its definition of the grammar productions.

Finally, after the derivation tree of the transformed input source model has been obtained, the GBMT implementation generated by the tool, in our case defined using TGGs, is applied to the obtained tree using the engine, here the EMorF engine. From the resulting target derivation tree we can easily extract the output target model.

Overall, all of the described parts of the chain to develop, validate, and execute GBMTs have been designed and validated on the conceptual level. The remaining implementation steps, required for its user-friendly use and a fully automatic execution are a matter of technicality. The supporting parts that need to be implemented include a graphical UI, and several model transformations to integrate the tools for the individual steps into the chain.

5.6 Related Approaches

Now, we compare our grammar based approach to the development of model transformations with other quality-by-construction model transformation development approaches, discussed in Section 4.2.2. We also provide an overview of further related approaches that have similar goals as our approach, but do not guarantee the transformation quality properties at the current stage of their development.

For years, compiler construction has benefited from syntax-directed translation [ASU86]. This technique relates single source and target string grammar productions via 1-to-1 relations and requires both grammars to have the same non-terminals, creating implicit correspondences by name. This way, the syntax-directed translation technique builds a very basic version of our approach. Pratt [Pra71] was the first to propose to apply this technique to graphs and to show, that the resulting transformations are deterministic and reversible (under conditions). Our approach extends [Pra71] to n-to-m relations between the productions (extended GBMT rules), n-to-m correspondences between the non-terminals, and non-local GBMT rules, and shows additional properties for the developed model transformations. Thus, we consider a much larger scope of possible model transformations than the approach in [Pra71].

TGGs proposed in [Sch94], were also inspired by Pratt [Pra71] and were first to contain explicit correspondence nodes and links. Initially, the focus of TGGs was on relating context-sensitive monotone graph grammar productions to support data integration. In [Sch94], the author did not consider quality properties of the resulting transformation, as we do in our approach. Unlike [Sch94], our approach also does not require the individual productions to be monotone.

In MDSD today, TGGs are used to develop model transformations. They are defined on meta-models and relate source and target model patterns (see Section 4.1.3), gradually matched/created during the execution, instead of grammar productions. Thereby, only the language structures described in the meta-models via structuring classes can be used to express structure-based transformation mappings within the TGG rules, and one can no longer benefit from the implicit replacement mechanism of grammars, in order to connect the transformation results. Plus, additional methods, discussed below, are needed to guarantee the quality properties of such meta-model and TGGs based model transformations.

In [EEdL⁺05, EHS09, EEHP09, HEOG10], considered in Section 4.2.2, the authors proposed approaches to prove termination, soundness, completeness, and determinism (confluence) of TGGs based model transformations. As discussed in Section 4.2.2, the authors' definition of the second and third property differs from ours. Their source/target languages consist of all source/target models generated by the respective (source/target) parts of the defined TGG, whereas, we use an additional definition of our source/target languages through the meta-model conform HR grammars (see Section 5.1.2). As for termination and confluence, the conditions the authors of [EEdL⁺05, HEOG10] put on TGGs based model transformations, in order for them to have these properties, are, in general, more complex to show (e.g. critical pair analysis) than the conditions we put on our GBMTs for the same purpose (see Section 5.3).

On the other hand, since TGGs based transformations in MDSD are defined over meta-models, their definition process can not cope very well with inductively-structured languages, like UML Activity diagram or our ADR language (see Section 3.1). For the latter, we have explicitly discussed this issue in Section 4.1.3, where we have considered an implementation of our ADR-to-CSP transformation example from Section 3.3 using TGGs. Our grammar based transformation development approach attempts to solve this issue by using meta-model conform HR graph grammars of the source and target languages as a basis for transformation development. This way, we can also explicitly refer to high-level language structures (e.g. activity blocks) in our transformation rules and benefit from the inductive nature of grammar based definition of our modeling languages, when we prove quality properties of the developed model transformations.

The only approach that we are aware of that, unlike [Sch94], uses TGGs over meta-models and defines model transformations containing structure-base mappings, is proposed in [LGJS07]. The authors consider a transformation of UML Activity diagrams, defined by a meta-model without the structuring classes, into their Business Process Execution Language (BPEL) [OAS07] representation. The proposed approach uses TGG

rules that create the source/target language structures, required for the definition of the example transformation (e.g. work-flow patterns), in a suitably defined correspondence model. They refer to these structure types as "elements representing the main semantic features of the participating languages", and put a special emphasis on the creation of a custom correspondence meta-model, that contains all of the required structuring classes to describe them.

Like our grammar based approach, the approach from [LGJS07] advocates an explicit modeling of structures within the source/target languages, in order to support the implementation of structure-based transformation mappings. The authors also point out that they had to adjust the target meta-model for the purpose of their transformation, which might not always be possible. Such duplication of all transformation-relevant information in the correspondence part of the developed TGG rules and potential meta-model adjustments, make application of this approach and the resulting TGG based transformations quite complex. Furthermore, the authors rely on the well-formedness of source models in their TGG rules, which needs to be ensured separately by checking their conformance with the source meta-model and its OCL constraints (when given). As for the transformation quality properties, that are of interest to us and are guaranteed by our grammar based approach (see Section 4.2.1), no consideration is given to them in [LGJS07].

Halfway between the grammars, as in our approach, and the meta-models in MDSD, is the model transformation development and validation approach, proposed in [Küs06]. We have already discussed its main ideas in Section 4.2.2 and, thus, to avoid the repetition, we only compare it to our approach here. In [Küs06], the author relates source model patterns with target language productions in compound rules, whereas we relate grammar productions on both sides. The author defines source languages using meta-models and target languages using context-free string grammars. It is not discussed, how the approach would deal with graph-based target languages, that can not be defined using string grammars (see Section 5.1.2).

In [Küs06], the transformation execution strategy is defined manually using transformation units, whereas, in our approach, it is automatically obtained during the input model parsing. Although the former approach is more flexible, the latter one is less error prone and sufficient for our translation purpose. Like us, the author of [Küs06], considers transformation quality properties and states several sufficient criteria to guarantee them. The focus is on transformation termination, confluence, and syntactical correctness (weak soundness, see Section 4.2.2). The stated termination criteria require certain structure of transformation units (minimality) and rules (creating at most one non-terminal) to ensure the reduction of the number of non-terminals in the constructed target model after each application of a rule. In our approach, we do not have this significant restriction on individual rules and our inherently finite execution process, led by the derivation tree of the input source model and correspondences, is always finite (see proofs in Section 5.4).

The confluence criteria, stated in [Küs06], aim to eliminate conflicting transformation rules (critical pairs [EPT04]), whose re-ordering can produce different results. They require parallel independence of the compound rules in a unit, which, in turn, requires knowledge of well-formedness of the source language. Such knowledge is not always present in meta-models (see Section 2.2). In our approach, we are able to use this knowledge explicitly from the source and target language grammars. Furthermore, to eliminate the second source of non-determinism during the match selection, the author of [Küs06] restricts the number of matches for each currently applicable rule in the unit (except for the loops) in any source model to at most one. For the loops rules, the author employs elaborate critical pair analysis [EPT04] to check confluence. When the analysis shows that the critical pairs are not confluent, the author suggests resolving the problem by additional constraints on the source/target language or by changing the rules. Our approach does not require such an elaborate analysis technique but also has several non-trivial conditions, including unambiguity of the source language grammars, in order to guarantee determinism (see Section 5.3.1).

As for the syntactical correctness criteria, stated in [Küs06], they are based on rule reachability in a non-terminal dependency graph, constructed for the model transformation. Although, these criteria are simpler than those for termination and confluence, they only guarantee syntactical correctness of the rules and their applicability, and not syntactical correctness of the generated target models, as we do in our approach.

To complete the comparison of our approach with other quality-by-construction techniques for model transformations, discussed in Section 4.2.2, we still need to consider [VVGE⁺06, BLA⁺11, LKR13]. In [BLA⁺11], the method based on a Turing-incomplete language without recursion and loops is used to simplify the model transformation development and the proofs of termination and confluence of the resulting model transformation. Such a design decision significantly restricts the scope of the model transformations, that can be described using this approach. Our approach also has certain restricting conditions on the resulting GBMTs (see Section 5.3), yet we are able to express our example transformations like ADR-to-CSP using it, whereas this is not possible using [BLA⁺11], since we would require recursion/loops (see Section 4.1.3 for discussion).

In [VVGE⁺06], the authors prove termination of (a special class of) graph-based model transformations by translating them into Petri nets and re-using the available termination proofs. We could use this approach to check termination of our TGGs based implementation of the ADR-to-CSP transformation (see Section 4.1.3) with a positive or an inconclusive 'maybe' answer. Our approach, however, provides a definitive guarantee of termination and other quality properties, if the appropriate conditions on GBMTs (see Section 5.3) hold.

Finally, we have mentioned [LKR13], where the authors propose an approach for the automated derivation of correct-by-construction transformation implementations from their high-level specifications by UML Class diagrams with specially-formed OCL constraints. The generated implementations, expressed in a simple procedural language,

then terminate and are confluent and sound. Although the specification languages chosen by the authors provide sufficient expressiveness, their use for more complex transformations like our ADR-to-CSP example seems infeasible due to the complexity of the required specification. Furthermore, such OCL specification has to fulfill several complex side conditions (see [LKR13] for details). Our grammar based transformation definitions, on the other hand, tend to stay relatively simple especially when viewed in concrete syntax (see Section 5.2.1, Figure 5.9).

Several further approaches [Var06, SWG09, LWK10] look at the problem of simplifying model transformations and their development from another perspective. They advocate model transformation development by-example and by-demonstration, and, as the name states, aim to reduce the process complexity by inferring transformation rules from multiple examples. The authors focus on simplifying the transformation development process and making it more demonstrative, and do not consider quality of the resulting transformation as much as we and the approaches mentioned before do. Like us, the by-example/by-demonstration approaches recognize the problem of over abstraction of language definitions through meta-models. They deal with this problem by using examples to describe the corresponding structures of the source and target languages, whereas, we relate graph grammar productions to do this. Although such focus on examples and the development freedom promises better user acceptance for the by-example concept, it also has some drawbacks. Unfortunately, it is not always clear, whether the examples or the model transformation should be adapted when the result is not satisfactory. It is also hard to predict how many examples are needed to learn the required model transformation in full [Var06].

In [FCB⁺13], the authors also attempt to learn from valid and invalid model examples. They, however, address the problem of over-abstraction through meta-models, discussed in Section 2.2, by learning the well-formedness constraints for meta-models. They use genetic programming techniques to synthesize and mutate explicit OCL constraints.

Several other approaches [LD09, IP10] choose the opposite direction and attempt to apply the classic translation techniques for compiler construction [ASU86] to develop model transformations. Unlike in our approach, the authors do not switch from string to graph grammars but keep working with the former. They also do not consider the quality properties of the developed model transformations.

In [LD09], the authors attempt to use TXL [Cor06] – a generic source transformation framework – to develop model transformations. They consider meta-model based definitions of source/target languages and transform these definitions into TXL string grammars. TXL string grammars do not have the expressiveness and visualization advantages of graph grammars, that we use, leading to a very limited applicability of the TXL based approach. Model transformations described in TXL are fine-grained with an explicit execution policy, which makes them flexible, but also complex and difficult to understand and maintain. This method can be placed halfway between the source (syntax-directed) translation technique, used in classic compilers [ASU86], and our grammar based approach to the model transformation development.

In [IP10], the authors attempt to simplify model transformation development by eliminating the need to learn specialized languages. They regard models and meta-models as abstract data types – abstract structures with operations. On top of these types, they define a minimal imperative model transformation language with formal semantics. This approach brings models and model transformations into the world of programming, whereas our approach lifts translation techniques to graphs. Both last approaches use meta-model based definition of modeling languages, and none of them explicitly considers the cases when the developed transformation has to map high-level language structures, like decision/merge blocks in our ADR-to-CSP example (see Section 3.3). In our approach, in such cases the developed makes use of the structures, defined by the source/target language grammars through one or more productions.

In general, the use of alternative definition techniques for modeling language by the above approaches – meta-model vs. graph grammar – raises the issue of integration and interoperability of such approaches and of the related tools. Various approaches attempt to address this issue by defining transformations between these alternatives [WK06, HM11], applying inference to obtain graph grammars from models [SC12], or combining them as different views on the language in the multi-level modeling paradigm [AGT12].

In our approach, we chose the latter option and see graph grammars (HR grammars in particular), defined based on meta-models, as an additional static structure-oriented view on a modeling language. This allows our approach to natively support structure-based transformation mappings, while keeping the resulting rules declarative and concise (see e.g., Figure 5.8). The inductive structure of grammar productions also simplifies the proofs, required to guarantee the quality properties of the resulting grammar based transformations (see Section 5.4).

Finally, we want to point out, that a very simple version of the idea, that our method is based on, has been recently successfully used in [JAB⁺12] for semantics-based machine translation in the field of computational linguistics.

5.7 Evaluation and Discussion

To evaluate our grammar based approach to the development of model transformations, we compare it with the most common transformation development practices in MDSD, that we have discussed in Section 4.1. There, we have also provided two possible implementations of our example ADR-to-CSP transformation, from Section 3.3, using the ADR and CSP meta-models from Sections 3.1 and 3.2, and either declarative TGGs or a hybrid model transformation language ATL. We later use these implementations and a grammar based definition of the same ADR-to-CSP model transformation, provided in Section 5.2.1, to describe a detailed comparison of the approaches. We summarize the results in Table 5.1. Thereby, we compare the TGGs based approach to the development

of model transformation from [Sch94] in the MDSD context with the meta-model based language definitions instead of grammars, unless otherwise stated.

In general, in the context of MDSD model transformations are defined based on meta-models for their source and target languages. Such meta-models may or may not contain additional structuring classes (e.g., *StructuralActivityNode* in UML) to group (other) classes and, thereby, represent high-level structures of the languages. In our example ADR language structures like activity block sequences, decision/merge blocks, and fork/join blocks could be represented through the respective structuring classes, as discussed in Section 4.1.2.

On the other hand, on the side of model transformation development in MDSD, there are multiple implementation options with declarative [QVTc, QVTa], imperative [QVTd, Ker], and hybrid languages [ATL, KPP08, Eps] with further differences with respect to the main concepts, like relations, functions, operations, and so on. These options may be combined with the design steps for the transformation mappings (see Section 4.1.2). Furthermore, several authors [GdLK⁺10, VG08] have proposed to apply MDSD techniques to manage, combine, and extend these development steps.

Nevertheless, there are several issues that are linked to the meta-modeling foundation of these methods, that are unresolved. First, as discussed in Section 2.2, a meta-model often lacks the explicit information about high-level structures of the language it describes. Thus, model transformations, defined based on such meta-models for their source and target languages, can not access this structural information directly, and their developer is forced to (recursively) search the input source models to gain such information and trigger the respective rule(s) to create the corresponding structure(s) in the target model(s). We have seen an example of this issue in Section 4.1.3, in the provided ATL and TGGs based implementations of our example ADR-to-CSP transformation. There, the corresponding decision/merge nodes of a source ADR model needed to be located, in order to transform the resulting decision/merge activity block structure into an if-then-else CSP process expression, according to the requirements from Table 3.1.

In Section 4.1, we have considered the most evident solution to this issue – the introduction of structuring classes into meta-models. This solution allows a transformation developer to obtain the required structural information from the input source models, that now also contain instances of structuring classes, and create the respective structures in the generated target model. In Section 4.1.2, we have shown an example of a structuring class for our ADR meta-model and, in Section 4.1.3, we discuss the use of this class in the transformation implementation process on our ADR-to-CSP example. We show that such classes allow to simplify the development of model transformations requiring structure-based mappings. Yet, often the given meta-models and/or source models can not be changed/re-modeled to allow this.

In our grammar based method for transformation development we tackle this issue of under-specification through meta-models from another direction. We use HR graph grammars, that are conform to the respective meta-models, in order to describe source

and target languages of the developed model transformation. This way, we benefit from the structure-oriented nature of grammar based language definition (see Section 5.1) and can explicitly model language structures, as parts of grammar productions without having to change its given meta-models. Our approach then advocates the use of such grammar based language definitions as a better (than meta-model) basis for the definition of various model transformation mappings and, of structure-based mappings in particular. Through the direct use of productions for the transformation definition, the developer can directly map high-level language structures in the rules and is not forced to use imperative constructs and recursion to find them first. In Section 5.2.1, we provided examples of such concise structure based mappings for our ADR-to-CSP transformation to illustrate the benefits of our grammar based approach.

In Figure 5.21, we sketch another example to illustrate the seamless modeling of structure based mappings with our approach. Here, we provide one of the two¹³ non-trivial mappings, that are needed to flatten hierarchical activity diagrams, with activities inside nodes, into simple activity diagrams. This mapping relates the production of an activity node with two nested activities to the production for a fork/join block. For simplification, both productions are shown in concrete syntax. To complete the mapping, we added a pre-condition correspondence between the non-terminals B and correspondences between the respective non-terminal hyperedges, representing the created blocks.

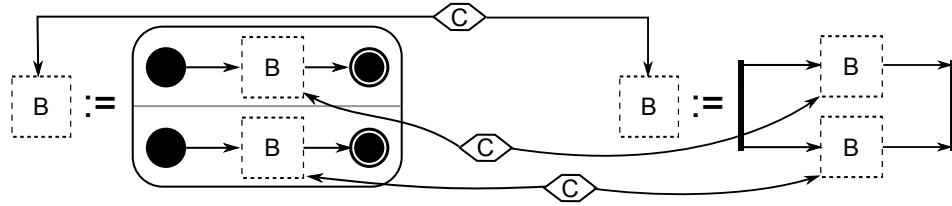


Figure 5.21: Example structure based mapping: flattening hierarchical activities

The HR grammar based language descriptions, used in our approach, in general, are more restrictive and complex than pure meta-model (with structuring classes) based ones. However, when compared to meta-models with OCL constraints of comparable expressiveness, that enforce (when possible) the same structural well-formedness constraints on e.g., activity diagrams (see end of Section 3.1 for examples), HR grammars typed over meta-models present a more concise, intuitive (possibly with concrete syntax), and powerful way to describe such structural constraints. Like meta-models, they only need to be created once per language and can then be re-used by any transformation developer, that employs our approach. The quality and complexity of an HR grammar can affect the characteristics of the grammar based model transformations using it, but this is also the case with meta-models and model transformations defined based on them.

Another issue, that model transformation developers often face, while using meta-model based MDSD techniques, is the connection and composition of the results of the

¹³Only valid activities with non-empty nested activities are taken into consideration.

individual mappings into the end target model. Depending on the way the mappings are implemented, e.g., as imperative rules or declarative relations, and on the chosen particular implementation language, different composition mechanism can be used.

In general, however, explicit implementation of the connection logic is necessary in meta-model based approaches. We have seen examples of such connection logic and discussed their purpose and necessity in Section 4.1.3, for both ATL and TGGs based implementations of our ADR-to-CSP transformation. There, we also pointed out the additional complexity that such a connection logic, like assignments in *do* blocks (in ATL) and additional *bound* elements in the rules (in TGGs), introduces. In some transformation languages, like ATL, this can convert otherwise declarative rules into imperative ones, which, in turn, complicates their quality assurance (see Section 4.2 for details).

In our grammar based approach this connection issue is resolved differently. To connect the results of the individual grammar based rules, we employ the connection/gluing mechanism of HR productions of the target language, which is implicitly present in each of these rules. We provide multiple examples of grammar based rules for our ADR-to-CSP transformation in Section 5.2.1 as a demonstration.

When a grammar based rule is applied, a non-terminal hyperedge (or two in the case of a non-local rule) of the target derivation tree fragment is replaced by the target sub-tree(s) of the rule (see Section 5.2.2 for details). Thereby, the replacing tree is automatically connected to the remaining target tree according to the mappings of its top production. Thus, one does not need to define additional, potentially complex, connection logic in our approach. This fact, together with the previously discussed explicit modeling of language structures by productions, allows us to keep our grammar based transformation rules declarative.

The declarative form of model transformation definitions has several advantages, including the following: better readability/understandability; absence of complex imperative constructs, like recursion; and, thus, easier development and more effective quality assurance (e.g., [CCGdL10, BECG12], see Section 4.2.3) and, hence, higher quality.

With respect to the last point, our grammar based approach goes even further. In Section 5.2.1, we discuss and later, in Section 5.3.1, formalize the conditions on the form of both single grammar based transformation rules and the complete model transformations, that are sufficient to guarantee the following four quality properties, described in Section 4.2.1: termination, soundness, completeness, and determinism. There, we also show that our grammar based definition of the ADR-to-CSP transformation fulfills the required conditions and, thus, has the named quality properties. In Section 5.4, we then provide the formal proofs of these properties for the grammar based transformations, that fulfill the respective conditions.

As we have discussed in Section 4.2.2, none of the existing state-of-the-art quality assurance approaches for model transformation can guarantee all of these four properties by-construction, like our approach. The same is true for the quality assurance approaches for already existing model transformation (by-checking), that have been surveyed in

Section 4.2.3. Plus, if in the first scenario the considered model transformation can still be adjusted to try to fulfill the quality requirements, in the second scenario, only a non-invasive check is possible.

	Language Definition Technique		
Transformation Characteristics	Meta-Model	Meta-Model with Structuring Classes	Graph Grammar
Structure Based Mappings			
Declarative	[Sch94, Küs06, QVTc] (complex)	[Sch94, Küs06, QVTc] (simpler), [LGJS07]	[Pra71, BSW14], [Sch94] (without meta-models)
Imperative	[Küs06] (units), [BLA ⁺ 11] (insufficient), [LKR13] (pre-defined), [IP10],[QVTd], [Ker, ATL, Eps]		-
Mappings Results Composition			
Implicit	[Küs06]	[Küs06]	[Pra71, BSW14]
Explicit	[Sch94, IP10, BLA ⁺ 11, LKR13, QVTc, QVTd, Ker, ATL, Eps]	[Sch94, IP10, LGJS07, BLA ⁺ 11, LKR13, QVTc, QVTd, Ker, ATL, Eps]	-
Quality Properties By-Construction (defined in Section 4.2.1)			
Termination	[Küs06, BLA ⁺ 11, LKR13]		[Pra71, EE ^d L ⁺ 05, BSW14]
Soundness	[LKR13]		[BSW14]
Completeness	[NHBE14] (partially)		[BSW14]
Determinism	[Küs06, BLA ⁺ 11, LKR13]		[Pra71, HEOG10, BSW14]

Table 5.1: Classification of model transformation development approaches, considered in Sections 4.2.2 and 5.6, and of our approach, published in [BSW14]

Such model transformation quality guarantees by-construction, of course, pose some restrictions on our grammar based approach which, otherwise, could be used to simulate Turing complete TGGs. The conditions on the form of the rules and on the complete grammar based model transformations, that we have already mentioned before, restrict the way a given transformation logic can be expressed with them. Some of these restrictions can be overcome through the changes to the transformation rules and, if allowed, to the source/target grammars. For instance, in our grammar based definition of the ADR-to-CSP transformation example, we were able to fulfill the required conditions and guarantee the properties without having to change the grammars. In Section 7.2, we discuss another GBMT definition, where this was also possible when the source HR grammar took a certain form. In general, however, this is not always possible to achieve and, although the resulting GBMTs can still be used, their quality can no longer be

guaranteed in this case. Still, the existing testing and verification techniques, surveyed in Section 4.2.3, can be employed to check them.

Depending on the given transformation requirements, the resulting grammar based rules can have different complexity. Later, in Section 7.2.2, we show an example GBMT definition other than our ADR-to-CSP example, where we can clearly see this complexity. In such cases with complex sub-trees in the GBMT rules, their readability/understandability decreases, which is also the case for complex meta-model based transformation definitions. Unlike in many MDS approaches, in our grammar based approach we can attempt to reduce this complexity by switching to the concrete syntax representation, as advocated in [Küs06, Var06] and shown before in Figures 5.9 and 5.21. Of course, this syntax form has to be available.

Another source of complexity in our grammar based approach is model parsing, required to obtain the structural information and the derivation trees (for context-free grammars). However, as discussed in Section 5.1.2, parsing based on HR grammars is decidable and can also be done efficiently for our case [Lau90, CAB⁺13]. Furthermore, we see benefits in splitting this parsing step from the model transformation execution, since it is then only needed to be performed once for each model and grammar combination. In the approaches [LGJS07, VHV08], where such parsing in the search for structural information is integrated into the matching – first step of the model transformation execution process – it has to be repeated on every transformation. In [VHV08], the authors refer to such implicit forms of parsing during the matching as *recursive pattern matching*.

On the other side of the parsing process are HR grammars (see Section 5.1.2). This type of graph grammars, that we use in our model transformation development approach, explicitly restricts their applicability scope. At this point, we can define model transformations between the modeling languages, that can be described using context-free HR grammars. This limitation, however, can be overcome by considering an extension of our approach to context-sensitive graph grammars, which we outline in Section 8.4. In simpler cases, one can also employ the duplication of model elements or references, as we show on an example and discuss in Sections 7.2.1 and 7.3.

Overall, we consider our grammar based approach to be the next step towards an efficient and quality-aware model transformation development, that can be realized based on the existing state-of-the-art model transformation languages and engines (see Section 5.5), including the commonly used ATL and TGGs.

With respect to the model transformation classification from Section 4.1.1, our grammar based approach can be applied to describe both out-place and in-place transformations, as a special case of the former with equivalent source and target HR grammars. Model-to-text transformations can also be expressed in a natural way using our approach, since textual languages are usually defined using string grammars, that can be expressed as HR graph grammars in a straightforward way.

As for the different model transformation application directions from Section 4.1.1 – abstraction, refinement, re-factoring, migration, and representation change – applicability of our approach with guaranteed quality properties highly depends on the particular transformation requirements and structures of the source/target HR grammars. In the cases when parts of an input source model need to be removed for e.g., abstraction, and nothing should be created in the corresponding target model for them, it might be difficult to express this in terms of GBMT rules. This can require additional target grammar productions with the same left- and right-hand sides, that have no effect on target models. We refer to such productions as *idle*. In Section 8.4, we discuss this and further possible extension and improvement of our grammar based approach.

Chapter 6

Weaving Based Approach to the Re-Use of Model Transformations

In the previous chapter, we have proposed a model transformation development approach based on graph grammars. In the context of our case study from Chapter 3, this method can be applied to transform the individual layers models of the system (into e.g., CSP). In this chapter, we address the following second major question brought up by our case study: How to allow the re-use of the developed grammar based and other model transformations, and of their results for the models of the individual layers? This is particularly interesting when the layered system to be transformed can be re-configured – new layers models added, re-connected, and/or removed.

There are distinct conceptual ways to approach this question, resulting from the following choices to (or not to) compose the layers models (Ms) and the re-used model transformations (MTs):

1. keep the layers models separate (n Ms);
2. keep the model transformations for the layers models separate (m MTs);
3. compose or model the system layer models in one complete model (1 M);
4. compose or define the re-used model transformations in one complete model transformation (1 MT).

This leads us to the following four possible directions and respective approach ideas: combining models and model transformations (1 M + 1 MT), combining models and keeping separate model transformations (1 M + m MTs), keeping separate models and combining model transformations (n Ms + 1 MT), and, finally, keeping both models and model transformations separated (n Ms + m MTs). The first direction suggests quite a compact solution following the all-in-one principle that, however, is not very flexible when the system needs to be re-configured or some of the model transformations have to be changed. The second and third combinations are less compact, but allow more flexibility, either when changing one of the transformations or one of the layers models, respectively. The last direction is the least compact yet the most flexible one. Like the third option, it requires a separate model for the interactions/inter-relations between the

layers and, like the second option, it requires additional steps to allow the transformation re-use by chaining (see Section 4.3.1).

Several proposed approaches [KR10, NRKR13, Gro13, VVBJB07, CM09, ASWK11, SHNG12] explore the directions named above. Next, we provide their brief discussion and classification based on these directions, summarized in Table 6.1.

In [KR10], the authors follow the first direction and use annotations to enrich the given application layer system model with the details from the chosen platform/infrastructure layers, like message-oriented middleware. Such annotations are defined in the form of feature model configurations with the transformation fragments attached to the individual features or feature sets. The refinement transformation for the application layer model is then generated by a higher-order transformation, based on the annotations and the related transformation fragments. These fragments introduce the respective model completions, taken from a library, to obtain the complete system model (1 M). The generated refining transformation then produces a system model where the annotated application layer model elements are expanded to their detailed platform/infrastructure dependent specifications. The authors then transform this model, by a single transformation (1 MT), into a queuing network to perform model-based performance analysis.

The solution proposed in [KR10] provides a quite flexible feature model based way to enrich the application models with the platform/infrastructure information for further analysis. Yet, it hides these details in the feature models and transformation fragments, which does not allow their explicit modeling as further system layers, that can also interact with other layers. Furthermore, the structure of the allowed configurations is restricted by the options of the given feature models, which might not always be sufficient. It is also not clear how to deal with potentially interacting or conflicting transformation fragments within such feature models.

On the other hand, in [NRKR13, Gro13], the authors employ new types of infrastructure and resource interfaces, introduced in [BKR09], as a part of a major extension of the modeling language and the framework used in [KR10]. With the help of these interface sub-types, the authors explicitly model all system layers (n Ms) and their interactions in an assembly model, which provides a more flexible solution than the previously discussed one-model option. The proposed solutions follow the third (n Ms + 1 MT) direction and provide more flexibility, yet require all system layers to be modeled in one language (Palladio component model [BKR09]). On the model transformation side, these approaches still use the same inflexible one-transformation (1 MT) strategy for all models as [KR10].

In [VVBJB07, CM09, ASWK11, SHNG12], the authors attempt to keep the advantages of a compact single system model (1 M), that can be a composition of layers models, while allowing flexible re-use and exchange of multiple model transformations (m MTs) for the individual layers models. With respect to the combinations we have identified before, these approaches follow the second direction (1 M + m MTs).

In [VVBJB07], the authors focus on the composition of model transformations into chains by using the traceability information. They propose a way to extend model transformations in order to generate the required traceability models. If a model transformation depends on the traceability information from the other transformations, the specification of this transformation needs to include the corresponding required and provided trace tags. Overall, this approach requires significant adjustments of the re-used model transformations in order to produce/interpret the required traceability information.

In [CM09], the re-used model transformations are grouped in phases into the composition. During the first phase, the independent model transformations are composed and, in the second phase, the model transformations that depend on the outcomes of previously applied transformations, are composed. The authors use implicitly created traceability information, that can be queried by model transformations through a function they provide. This approach, thus, requires model transformations to implement such traceability functions, in order to allow the composition of inter-depending model transformations based on such additional context information.

In [ASWK11], the authors focus on the composition of model transformations based on algebraic graph transformations [EEPT06] at the rule level. The main idea behind this approach is to combine the individual transformations into one, that produces the same result. The authors achieve the composition of transformation sequences by repetitively computing the transitive transformations of two given model transformations using the result of the previous composition or the first transformation and the next transformation in the chain. Apart from the transformations ordering as a sequence, this method is further restricted by the requirement on each of the participating transformations to be exogenous (see Section 4.1.1) and to fulfill several criteria for its individual rules. To extend the applicability of their method, the authors discuss the translation of the transformations implemented in dedicated model transformation languages, like ATL, to the algebraic graph transformation specifications they use.

Like in [VVBJB07, CM09], in [SHNG12], the authors also focus on the composition of model transformations through their given context (application of other model transformations) and do not glue them into one transformation, like in [ASWK11]. The authors propose a framework for both sequential and contextual types of model transformation composition, where single transformations are considered as black-boxes and the interpretation of the traceability information of such model transformations is shifted to the framework level. In the framework, the individual model transformations are encapsulated into modules with the information about their produced/consumed models and model elements. Such modules are then composed by the transformation designer into a sequential/contextual composition to be executed. To allow the use of heterogeneous transformations in a composition, the authors provide technology adapters for the widely used model transformation implementation languages, including ATL and TGGs.

If we go back to the multi-layer systems, like our example from Section 3.1, in such a setting we want to be able to flexibly re-configure our system by exchanging the layers

Re-Use Option	1 Model Transformation	M Model Transformations
1 Model (all-in-one)	[KR10]	[WH05, VVBJB07, CM09, ASWK11, SHNG12, AEM12]
N Models (with inter-links)	[NRKR13, Gro13]	[SCGM08], [BWWB12] (our)

Table 6.1: Classification of the considered re-use approaches and of our approach

models and allowing them to be modeled in different languages (n Ms). We also want to be able to re-use the model transformation (m MTs), that exist for those languages, without having to adjust them. Thus, we are interested in the last solution combination with separate layers models and separate model transformations (n Ms + m MTs). To the best of our knowledge, none of the existing approaches, including the most closely related ones considered above, allow this.

To address this issue, we propose a weaving based model and model transformation re-use approach, published in [BWWB12]. It allows a non-invasive composition of a set of possibly heterogeneous models and of their transformations (and their results) in the context of more complex systems (like our multi-layer system) and heterogeneous transformations. We also classify our approach in Table 6.1, putting it in the same category as [SCGM08], provided that the technique proposed there is transferred to the inter-model links between the input source models.

Next, we describe our weaving based re-use approach in detail and discuss its application in the context of our example multi-layer system model and its transformation from Chapter 3, in order to allow the re-use of model transformations for the individual layers models. Such model transformations, in turn, can be developed using our grammar based approach, presented in Chapter 5.

6.1 Conceptual Description of the Approach

The transformation re-use approach that we propose works on a set of input models, possibly defined in different languages, and an additional model, that describes inter-relations between the elements of these input models. It then transforms this input set into the desired target language. We use a technique called *model weaving* [DdF07] in order to describe the inter-relations between the input models' elements in a separate *configuration* model. We assume that the model transformations from the languages of the individual input models into the desired target language already exist and can be re-used. Furthermore, we require the target language to provide compositional operator(s), which we discussed in detail later.

Figure 6.1 shows an overview of our weaving based approach to the re-use of model transformations. The input models in the set $LModel\ 0, \dots, LModel\ N$ are transformed

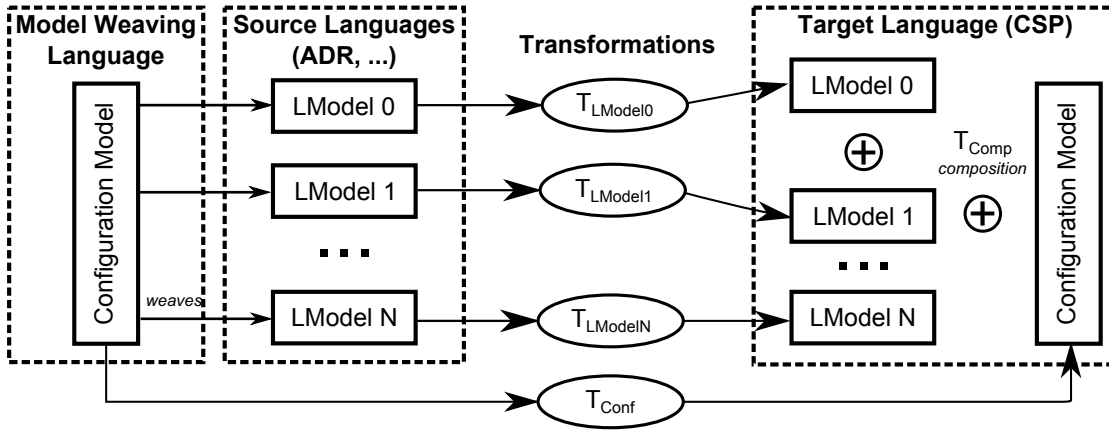


Figure 6.1: Overview of our weaving based transformation re-use approach

by multiple model transformations $T_{LModel0}$, ..., $T_{LModelN}$ (can be the same), which are executed in parallel. In our case study, these models contain activities describing behavior of the individual operations of the components of our multi-layer system (see Chapter 3). The *configuration model* is transformed by a separate transformation T_{Conf} . The results of these $N + 1$ transformations, here in CSP, are composed (T_{Comp}) into the final target model.

To consider our weaving based re-use approach in detail, we first introduce the mechanism we use to describe the configuration models in Section 6.1.1. Then, in Section 6.1.2, we discuss our modular concept for the execution of the re-used model transformations and, finally, the composition of the transformation results. In the description, all of these techniques are applied to our ADR-to-CSP transformation case study in order to allow its effective execution and re-use.

6.1.1 Weaving Based Modeling of Models Inter-Relations

In order to describe the inter-relations between the given models elements in a separate *configuration model*, we use a technique called *model weaving*, proposed in [DFBJ⁺05]. *Model weaving*, previously called model mapping [Ome03, HK03], at its core allows the definition of various types of links between the elements of different (meta-)models. These links are usually defined manually, sometimes with the help of heuristics [FV09]. The models that are linked are called *woven models* and the links between the models elements are called *weavings*. All these terms were introduced in [DFBJ⁺05].

Several approaches were proposed to facilitate model mapping/weaving, including [Ome03, HK03, DdF07]. They have been applied in various contexts and on various model levels in [KKK⁺06, FV09, VCFM10, FBV⁺09, GV09, MMPT10]. In [KKK⁺06, FV09], the authors use model weaving on the meta-model level to define mappings between classes (for all model elements) and later use them as a basis for the generation

of model transformations between the models of the respective languages. Later, in [MMPT10], this idea is extended to a method, that allows the use of multiple architectural languages during system development and provides interoperability between them.

Other authors use model weaving on the model level in such contexts like web engineering [VCFM10], aspect-oriented modeling (AOM) [GV09], and performance engineering [FBV⁺09]. In [VCFM10] and [FBV⁺09], weaving models are used as annotation models (mark models, see Section 2.3) to automate model-driven development of web information systems and performance engineering, respectively. In [GV09], the authors automatically generate weavings between the model elements to support model-driven engineering of software production lines. However, instead of keeping the weavings in a separate model, they use aspect-oriented composition techniques to generate complete product variant models based on them.

In our approach we use the generic model weaving technique, proposed in [DFBJ⁺05], in a different context – for the modeling of the interactions in multi-layer systems on the model level. Instead of using weaving models for the annotation, like in [FBV⁺09, VCFM10], we link models elements (our interacting layers elements) in a separate model, independently of the languages of these models. Next, we give a short introduction to the used technique.

Weaving with Atlas Model Weaver (AMW)

The generic model weaving approach was proposed in [DFBJ⁺05], and described in detail and implemented in the tool called *Atlas Model Weaver (AMW)* in [DdF07]. It uses a separate *weaving model* to store the model element links (weavings). This weaving model conforms to a weaving meta-model that specifies the link types. This approach allows extension of a pre-defined minimal *core weaving meta-model* under the consideration of certain requirements described in [DdF07]. The authors also provide a set of extensions of this abstract core meta-model for different purposes, that can also be extended and used as new weaving meta-models.

Figure 6.2 shows this core weaving meta-model of AMW. It contains the classes with the information about the link types, link end-points, and element identifiers. The element identifiers are used to store the unique identification information for the linked models' elements, to later access them in the woven models using a so-called *dereferencing function* ([DdF07], p. 75). This dereferencing mechanism allows to access the elements of the woven models to retrieve their type, attributes, and other linked elements. The primitives to support this mechanism are provided by the additional *AMW Model Handler* [AMW] library, that collects their implementations for different transformation languages. At present, it contains at least a library for the ATL language.

The different classes of the core weaving meta-model have the following purposes:

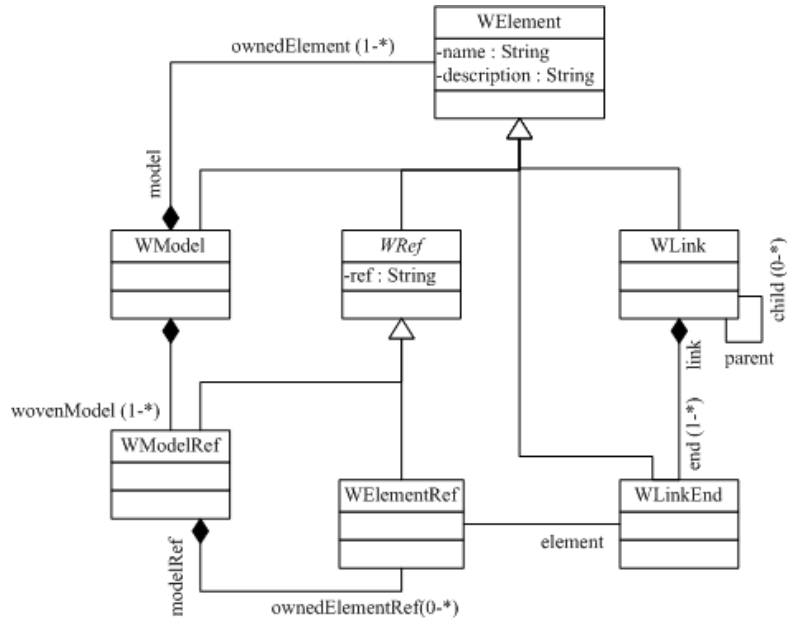


Figure 6.2: Abstract core weaving meta-model of AMW from [DdF07]

- *WElement* is the base class from which all other classes inherit. It has a *name* and a *description* attribute;
- *WModel* represents the root element that contains all model elements. It is composed of the weavings (*ownedElement* association to *WElement*) and the references to the woven models (*wovenModel* association to *WModelRef*);
- *WLink* expresses links between the model elements, i.e., it has a simple linking semantics. It also has an association with a set of link end-points (*WLinkEnd*). To be able to express different link types and semantics, this class should be extended;
- *WLinkEnd* defines the link end-point types. Every link end-point represents a linked model element and references one *WElementRef*. It allows to create N-ary links;
- *WElementRef* elements are used by the dereferencing function to access the linked elements from the woven models.

A weaving model can be used to link models as well as meta-models in different usage scenarios. In some cases basic extensions of the core meta-model are sufficient, but in other cases, like our case study from Chapter 3, we needed to define our own domain-specific weaving meta-model extension. We discuss how such an extension is created for our example scenario in the next section.

Weaving of a Multi-Layer System Configurations

In our case study from Chapter 3, we consider a model of a system that consists of multiple interacting layers. The interactions/inter-relations between the layers have to be modeled in some explicit way in order to be used by the transformations and other operations on the system models.

In our example system, component *TripPlanner* belongs to the application layer model and the *ThreadPool* component to one of the platform layer models. As an example of interactions between these layers, we consider threads acquisition on each invocation of *TripPlanner.getTrip()* and *MapDB.getMap()* operations with their later release on the operations reply.

In our approach, we employ the model weaving technique from [DdF07], that was described above in Section 6.1.1, and the supporting AMW tool. With its help, we model the inter-layer interactions of the system in a separate *configuration model*. A configuration model then contains the information about the models of the individual layers (woven models) as well as their interactions (weavings).

In our example system, we have identified the following two types of interactions between the layers elements:

- Type 1: an action from one layer triggers an operation in another layer as a *resource* once to, e.g., perform some computations or acquire/release a thread. This action can have any type;
- Type 2: an action, that performs a call to an operation or a resource acquisition/release within its layer, uses an operation from another layer to support this *communication* between the caller and the called operation or an action on a resource.

Due to the hierarchy principle in the layered system, the triggering actions are always located in higher (towards the application layer) layers than the triggered/used operations. We assume that each action interacts with at most one operation from the rest of the layers. In the cases, when multiple interactions of an action need to be described, we suggest their nesting inside the used operations.

To model the above interaction types we extend the core weaving meta-model of AMW by the following two types of interaction links: *resource handling* (class *ResourceHandling*) and *communication* (class *Communication*) for the types 1 and 2, respectively. We refer to the action that triggers/uses the other one as a *consumer* and to the used action/operation as a *resource*. For instance, in our layered system from Chapter 3, we use the resource handling type of a weaving link to weave the actions of activities (e.g., *initial/final* nodes of *TripPlanner.getTrip()* and *MapDB.getMap()* activities) as *consumers* to the *resources* within the other layers that they require (e.g., *ThreadPool.acquire()* and *ThreadPool.release()* operations within one platform layer model).

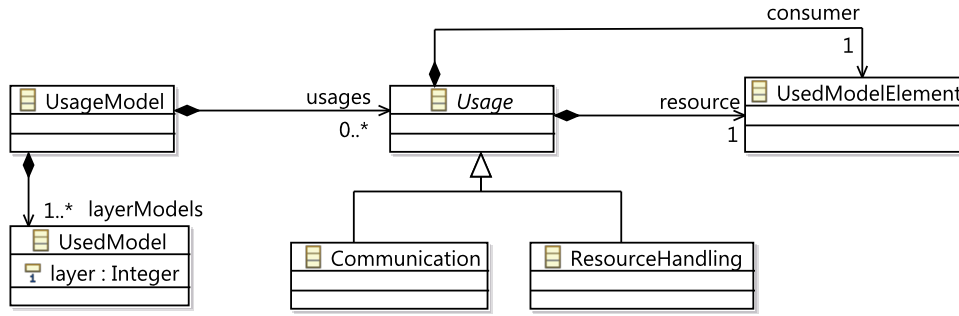


Figure 6.3: Weaving meta-model for multi-layer system configuration modeling in Ecore

Figure 6.3 illustrates the extension of the core AMW weaving meta-model that we have created to be able to define configuration models. It contains the mentioned link classes *ResourceHandling* and *Communication* extending the abstract class *Usage*, that models an abstract usage weaving between a *resource* and a *consumer* represented by the class *UsedModelElement*. The class *UsageModel* represents a configuration model itself with *usages* (weavings) and associations *layerModels* to the *UsedModel* class, that represents the woven models of the individual layers. The *UsedModel* class also has a relative layer number in the system represented by the attribute *layer*. Furthermore, the classes *UsedModel* and *UsedModelElementRef* encapsulate the mechanism (attribute *ref*) for referencing woven models and model elements, respectively. This mechanism is inherited by these extending classes from their parent class (*WRef*) from the core weaving meta-model described before. It is then used by the dereferencing function mentioned before.

Figure 6.4 shows a different tree-based representation of our extension of the core weaving meta-model of AMW. It contains the following three packages: the extended core AMW meta-model *mwcore*, our extension *mmw_usage*, and the meta-model for some primitive types. For each class (marked by a yellow rectangle with three sections before the name) in one of the meta-model packages, this representation shows the classes it inherits from. This inheritance relation is represented by an arrow ($\rightarrow$) from the child class (sub-class) to the parent (super-class).

With the above weaving meta-model extension we can now model configurations of multi-layer systems, like our example from Chapter 3. Figure 6.5 shows a fragment of one possible configuration model for our example, where activity *MapDB.getMap()* acquires a thread upon its invocation/initial node and releases it on the reply/final node. This configuration model weaves the application layer model, that contains activities like *MapDB.getMap()* (Figure 6.5, left), with the platform layer model, containing the activities *ThreadPool.acquire()* and *ThreadPool.release()* (Figure 6.5, right), through two instance of the weaving meta-model class *ResourceHandling* (Figure 6.5, middle). The first *resource handling* weaving ‘On *getMap* invoke *acquire*’ (red rectangle) connects the *initial* node of activity *MapDB.getMap()* as the *consumer* to the *ThreadPool.acquire()* ac-

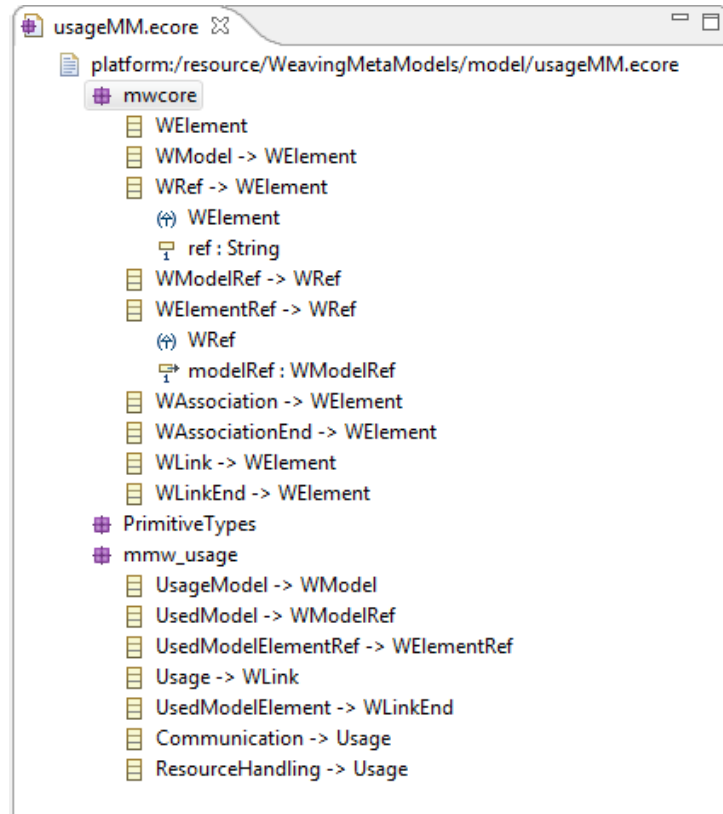


Figure 6.4: Weaving meta-model from Figure 6.3 in a tree-based representation

tivity as the corresponding *resource*. The second *resource handling* weaving ‘On *getMap* reply release’ (blue rectangle) connects the *final* node of activity *MapDB.getMap()* as the *consumer* to the *ThreadPool.release()* activity as the *resource*.

Another possible configuration model for our example system, that consists of the application and the RPC communication platform layer model from Chapter 3, is shown in Figure 6.6. Here, *TripPlanner.getTrip()* activity calls operation *MapDB.getMap()* through call action *callAction1*. This call is performed as an RPC call in both directions – on invocation/initial node of the activity and on its reply/final node. The configuration model in Figure 6.6, weaves the application layer model, that contains activities like *TripPlanner.getTrip()* (Figure 6.6, left), with the platform layer model, containing activities *RpcSender.transmit()* and *RpcReceiver.receive()* (Figure 6.6, right), through an instance of the weaving meta-model class *Communication* (Figure 6.6, middle). This *communication* weaving ‘On *getMap* callAction1 RPC transmit’ (red rectangle) connects action *callAction1*, that calls *MapDB.getMap()* as the *consumer*, to the *RpcSender.transmit()* activity as the corresponding *resource*.

Both example configuration models in Figure 6.5 and 6.6 can also be combined into one model with two layers and three woven models (two platform layer sub-models).

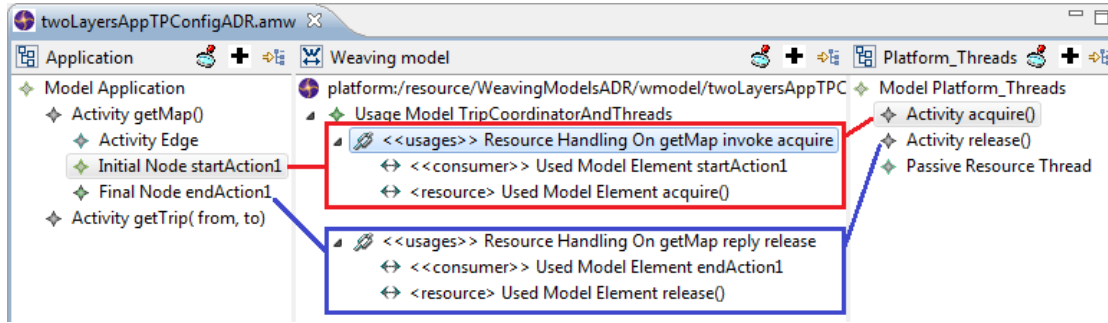


Figure 6.5: Part of a configuration model for a multi-layer system with the application and the threadpool platform layer model from Chapter 3

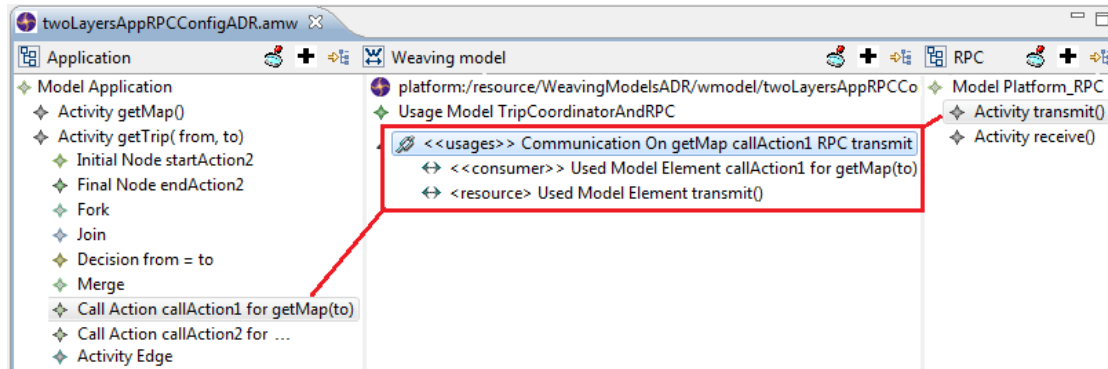


Figure 6.6: Part of a configuration model for a multi-layer system with the application and the RPC platform layer model from Chapter 3

The demonstrated weaving based technique for modeling system configurations, allows us to keep the system layers models free of inter-layer interactions and to make the representation and alteration (re-configuration) of this interaction information more flexible. Next, we consider how a model of a multi-layer system, configured using this weaving based approach, can be transformed into some target language in a way that promotes parallelization and re-use of model transformations (and of their results), that exist for its individual layers models.

6.1.2 Modular Execution of Transformations

Now, we consider a situation when a weaving based configuration model and the layers models it relates need to be transformed. In this setting, the transformation input has a natural fragmentation into a set of models, that we exploit in our transformation re-use approach (see Figure 6.1). We use separate model transformations for the individual models, that are independent from each other, and facilitate their parallel execution

and re-use (also of their results). Once these transformations have been executed, we compose their results in the target language.

Our approach states the following conditions on the source and target languages of the transformation, and on the individual transformations:

1. source languages are not subjected to any conditions apart from the use of a separate configuration model for the models' inter-relations (see Section 6.1.1);
2. target languages must allow composition of model fragments through one or more composition operators;
3. individual transformations between the source and target languages of models must produce target models, that are self-contained and can be used on their own as well as in a composition;
4. transformation of a configuration model must produce a target model fragment, that refers to all self-contained target models whose source models are in its configuration, and is only usable in the composition with these target models.

These generic conditions need to be interpreted for the different combinations of source/-target languages and different transformation goals, in order to ensure them. They also do not include any conditions on the semantic properties of the individual transformations, which are left for the developer to ensure. Next, we demonstrate how we interpret and ensure the conditions from this list for our ADR-to-CSP transformation.

After we have shown some possible configuration models for our example multi-layer system in Section 6.1.1, the entire set of models for our system is given, and we can now transform it into the target language of our choice (i.e., CSP). We have shown in Section 3.2, that CSP language possesses multiple composition operators, that can be used by the composing transformations. Thus, requirements 1 and 2 of the list hold.

Next, we need to consider each transformation into CSP used in our scenario and make sure that the last two conditions 3 and 4 from the list hold for them. First, we discuss the transformations (T_{LModel}) of the layers models, containing activities modeled in ADR. Then, we discuss the transformation (T_{Conf}) of a configuration model that links them, and, finally, the transformation (T_{Comp}) that composes the results of the former transformations. Each of these transformations is described in detail.

T_{LModel} : Parallel Transformation of the Woven Layers Models In our example ADR-to-CSP transformation each model from a specific system layer is transformed into the corresponding self-containing target CSP model based on the requirements described in Table 3.1. We have already discussed how such a transformation can be developed using a state-of-the-art meta-model based transformation development approach (see Chapter 4) and using the GBMT development approach that we have proposed (see Chapter 5). Now, we need to ensure that our ADR-to-CSP transformation fulfills the

respective requirements, stated in the beginning of Section 6.1.2, in order to apply our weaving based transformation re-use approach to it.

Requirement 3 (see Section 6.1.2) states that our ADR-to-CSP transformation must produce target models that are self-contained and can be used on their own as well as in a composition. If we combine the requirements from Table 3.1, we can see that our ADR-to-CSP transformation creates self-contained target models. However, they can not be directly used in a composition, that describes their process interactions, since they do not have any common events¹.

To allow our transformation results to be used in an interacting composition, we provide a mechanism to define which event sequences should take place together with which events. Thereby, all used events are taken from the target models. Later, we use this mechanism to express in CSP the interactions between the *consumer* activity nodes and the *resource* activities from the different layers models, related in a configuration model.

The mechanism that we propose is based on the idea of *splitting* each event ev in a CSP model into two sub-events ev_start and ev_end , representing the beginning and the end of this event, respectively. Such *event splitting transformation* $T_{EventSplit}$ is sequenced after each layers model transformation T_{X2CSP} from some source language X (ADR or other) into the target language (here CSP), as shown in Figure 6.7. This allows us to add a sequence of events between such two sub-events and, this way, ensure that that sequence is executed together with the original event ev from $LModel$. Using the result of this splitting mechanism ($LModel'$), we can express the interactions modeled in our system configuration models in CSP as follows: for each weaving we add the events, representing a call of the resource activity, between the sub-events of the event, that represents the consumer activity node.

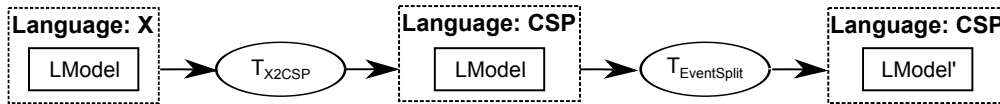


Figure 6.7: T_{LModel} : Layer model transformation into CSP followed by event splitting

The described splitting mechanism allows us to use the modified CSP models $LModel'$ in a composition, that is based on interactions modeled in a system configuration model, thereby fulfilling condition 3 from Section 6.1.2. The details of this composition are discussed later when we address the transformation of configuration models into CSP (T_{Conf}). Before this, we consider some example results of the proposed event splitting transformation and formulate its requirements.

Figure 6.8 shows the result of applying the additional splitting transformation to a fragment of the ADR-to-CSP transformation (see Section 3.3, while Figure 3.11) shows the result for a call of the `MapDB.getMap()` operation. It demonstrates how some of

¹We require unique activity node names throughout the layers models.

```

call_getMap_start.C1ID → call_getMap_end.C1ID →
    awaitreply_getMap_start.C1ID → awaitreply_getMap_end.C1ID →

channel call_getMap_start, call_getMap_end,
    awaitreply_getMap_start, awaitreply_getMap_end : {CallAction1ID}

```

Figure 6.8: Fragment of ADR-to-CSP transformation result for *MapDB.getMap()* call after the event splitting (*C1ID* abbreviates *CallAction1ID*)

the events that this call action is transformed into are split into two sub-events with the same parameter value – *CallAction1ID*. It also shows that the channels for the events are also split.

The splitting transformation converts all events, apart from those with an input parameter – *invoke_calledActivity?x*, *acquireRes?x*, and *releaseRes?x* – into two sub-events with the same (original) parameter. Events with an input parameter, like *invoke_calledActivity?x*, are split into two sub-events – *invoke_calledActivity_start?x* and *invoke_calledActivity_end.x* – with one input and one output parameter *x*, respectively. The same holds for the events of the form *acquireRes?x* and *releaseRes?x*. This is done because we only want to read the parameter *x* once and transfer it further to other sub-events.

```

ACQUIRE = invoke_acquire_start?x → invoke_acquire_end.x →
    call_acquireThread_start.AcquireActionID →
    call_acquireThread_end.AcquireActionID →
    awaitreply_acquireThread_start.AcquireActionID →
    awaitreply_acquireThread_end.AcquireActionID →
    reply_acquire_start.x → reply_acquire_end.x → ACQUIRE

```

Figure 6.9: Fragment of ADR-to-CSP transformation result for *ThreadPool.acquire()* activity after the event splitting

We see an example of such event splitting with an input parameter in Figure 6.9. The first event of the *ACQUIRE* process – *invoke_acquire_start?x* – accepts an input with a caller identification and stores it in the variable *x*, and the second event *invoke_acquire_end.x* simply passes it. This parameter is, as we see later, important for the synchronization of processes and for the composition of the transformation results.

In Figure 6.10 we see another example of the result of splitting *acquireRes* and *releaseRes* events in the CSP specification for the countable resource *Thread* with a ca-

capacity of 3 (Figure 3.5a, Chapter 3). The first sub-events have input parameters x and the second sub-events transfer the value stored in x further. Here, we also see the split channels for these sub-events – `acquireThread_start`, `acquireThread_end`, etc.

```

THREAD(0) = releaseThread_start?x → releaseThread_end.x → THREAD(1)
THREAD(n) = (acquireThread_start?x → acquireThread_end.x → THREAD(n - 1))
           □
           (releaseThread_start?x → releaseThread_end.x → THREAD(n + 1))
THREAD(3) = acquireThread_start?x → acquireThread_end.x → THREAD(2)

channel acquireThread_start, acquireThread_end : ACQUIRE_THREAD_CALLERS
channel releaseThread_start, releaseThread_end : RELEASE_THREAD_CALLERS

```

Figure 6.10: Fragment of the ADR-to-CSP transformation result for resource *Thread* after the event splitting

The described event splitting transformation can be defined using the state-of-the-art meta-model based model transformation development approach (see Section 4.1) or using our grammar based approach (see Section 7.1) proposed in Chapter 5. The resulting transformation is then sequenced after the main ADR-to-CSP transformation. This sequence, in its turn, is applied to each layer model independently, and the results are later (re-)used in the composition of the overall target CSP model (see Figure 6.1, right side).

Now, we put together all the requirements for the event splitting transformation of a layer model, that we have described and illustrated on the examples above. We express these requirements in a concise, example-independent, and complete form in Table 6.2.

Requirements 1 and 2 from Table 6.2 state that an event ev in a process prefix is to be transformed into two subsequent events ev_start and ev_end , extending this prefix. Depending on the direction of the parameter of event ev , the new events either have one input and one output parameter with the same name (requirement 1) or the same output parameter (can be a variable or a number) (requirement 2).

No.	Model Element(s)	CSP Specification Fragment
CSP Prefix Event		

Table 6.2: Overview of requirements on our CSP event splitting transformation

No.	Model Element(s)	CSP Specification Fragment
1	Event with input variable x : $ev?x$	$ev_start?x \rightarrow ev_end.x$
2	Event with output value val : $ev.val$	$ev_start.val \rightarrow ev_end.val$
CSP Event in Event Set X		
3	Event with output value val : $ev.val$	$ev_start.val, ev_end.val$
CSP Channel		
4	Channel $ch : VALUESET$	$ch_start, ch_end : VALUESET$

Table 6.2: Overview of requirements on our CSP event splitting transformation

Requirement 3 states that each event ev in event set X is to be transformed into two events ev_start and ev_end with the same parameter, that are put into the same set. This requirement covers all types of events in an event set, since such events can only have output parameters. Finally, requirement 4 states that channel ch is to be transformed into two channels ch_start and ch_end over the same set of values $VALUESET$.

Later, in Section 7.1, we consider how this event splitting transformation can be defined using our GBMT development approach (see Chapter 5) to guarantee its quality. Next, we discuss how a system configuration model, described in Section 6.1.1, can be transformed into CSP in a way that fulfills our re-use requirement 4, stated in the beginning of Section 6.1.2.

T_{Conf}: Transformation of Configuration Models The weaving models for representing configurations of a multi-layer system, described in Section 6.1.1, contain the interaction information required to compose the transformation results of the individual system layers models into the complete target model. They contain references to all involved layers models and usage weavings, that represent the interactions between their elements. Transformation of such configuration model must produce a model fragment that refers to all self-contained target models, whose source layer models were in the configuration, and that is only usable in the composition with these target models (see condition 4, Section 6.1.2).

We propose the following *CM-to-CSP*² transformation of configuration models to fulfill this condition in the context of our ADR-to-CSP transformation: we transform a configuration model M_C into a CSP specification fragment with one so-called *system* process (*SYSTEM*), that composes multiple processes representing the woven layers models *LModel*, composed into the *layers* process (*LAYERS*), and so-called *adapter* processes or *adapters* (*ADAPTERS*) representing the usage weavings. The processes

²CM stands for Configuration Model.

LAYERS and *ADAPTERS* are then synchronized through a set of common events Y . Figure 6.11 shows the described CSP fragments with some details of their composition and relations to the (elements of) the transformed configuration model M_C .

The *system* process represents the complete multi-layer system with interactions. The resulting CSP fragment contains the complete definitions of the adapter processes, but it only references the processes for the individual layers models (*LModel*). This limits its use to the composition with the CSP transformation results for these layers models.

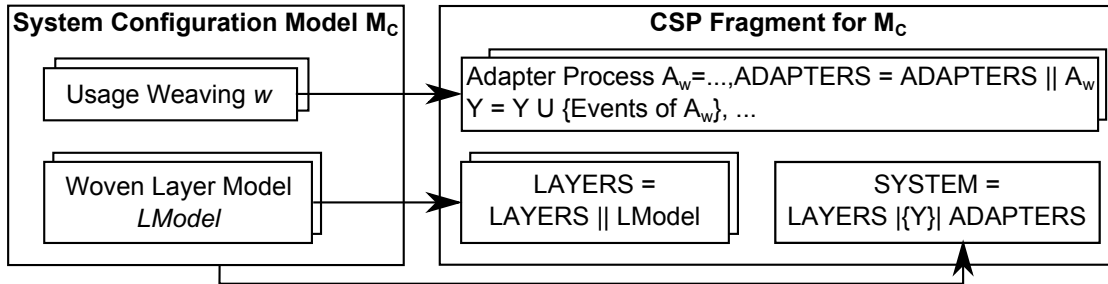
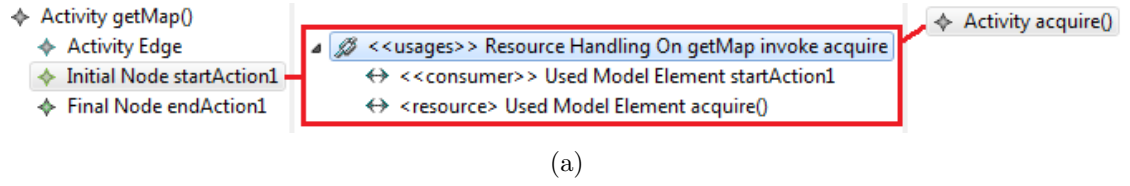


Figure 6.11: T_{Conf} : Configuration model transformation into CSP fragments

The introduced adapter processes provide a way to represent the interactions between the layers models, that were modeled as resource handling and communication weavings, as CSP processes. Based on the *resource* and the *consumer* of the weavings, the corresponding adapter CSP processes then define which events that represent the beginning and the end of the *resource* operation are executed between the beginning and the end events representing the *consumer* activity node. This way, using the sub-events created by $T_{EventSplit}$, we introduce an interaction between the previously independent *consumer* and *resource* element of each *resource handling/communication* weaving w . In order to allow these interactions to take place an arbitrary number of times, determined by the multi-layer system, the adapters A_w are defined as self-repeating processes.

Figure 6.12b shows an adapter process *ADAPTER_STARTACTION1ID_ACQUIRE* generated for the resource handling weaving in Figure 6.12a, with the *ThreadPool.acquire()* activity as the *resource* and *MapDB.getMap()* initial node (ID is *StartAction1ID*) as the *consumer*. The outer events of this adapter allow synchronization with the events representing the *consumer* node and the inner events with the events representing the *resource* activity, respectively. This adapter structure ensures on the CSP specification level, that the consumer action is completed only if the resource activity was successfully completed.

Apart from the adapter process, our CM-to-CSP transformation creates some additional CSP constructs for the weaving considered, shown in Figures 6.11 and 6.12b. It adds, via a parallel composition, the newly created adapter to *ADAPTERS* process containing all adapters of the configuration. It also adds the events of this adapter to the synchronization event set Y between the system process and the adapters, that is



```

ADAPTER_STARTACTION1ID_ACQUIRE =
    invoke_getMap_start?x →
        invoke_acquire_start.StartAction1ID →
            reply_acquire_end.StartAction1ID →
                invoke_getMap_end.x → ADAPTER_STARTACTION1ID_ACQUIRE

```

```

ADAPTERS = ADAPTERS || ADAPTER_STARTACTION1ID_ACQUIRE
Y = Y ∪ {invoke_getMap_start.x, invoke_getMap_end.x,
    invoke_acquire_start.StartAction1ID, reply_acquire_end.StartAction1ID}
ACQUIRE_CALLERS = ACQUIRE_CALLERS ∪ {StartAction1ID}

```

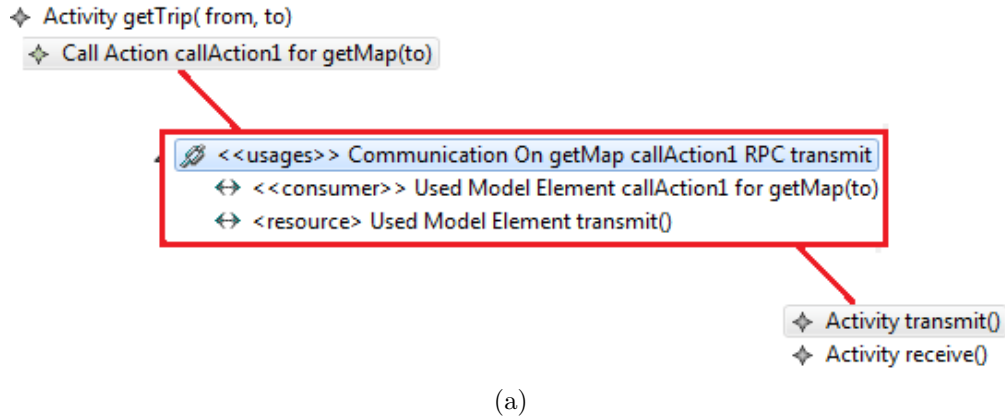
(b)

Figure 6.12: CM-to-CSP transformation result (b) for resource handling weaving ‘*On getMap invoke acquire*’ (a)

described later. Finally, it adds the ID of the consumer of the considered weaving to the set of callers (*ACQUIRE_CALLERS*) of its resource activity.

The adapters generated for the communication weavings have a similar structure to those created for the resource handling, but the former assume resource usage in both communication directions – on the resource/operation invocation and its reply. Therefore, the generated adapter process has a more complex structure.

Figure 6.13b shows such an adapter process generated for the communication weaving ‘*On getMap callAction1 RPC transmit*’ with *RpcSender.transmit()* as the *resource* and the *TripPlanner.getTrip()* call action to *MapDB.getMap()* as the *consumer*. This adapter has two alternative sub-processes combined via an external choice (see Section 3.2 for details). They represent the following two cases in which the resource *RpcSender.transmit()* activity is executed: when operation *MapDB.getMap()* is invoked and when it replies with some result to the caller. The outer events of both sub-processes of this adapter allow synchronization with the events representing the *consumer* and their inner events with the events representing the invocation and the reply of the *resource*. This ensures that the consumer actions, in this case an invocation and an expectation of a reply, are successfully completed only if the resource activity was successful. Since the call action that calls the *MapDB.getMap()* activity is the only allowed user of the



```

ADAPTER_CALLACTION1ID_TRANSMIT =
call_getMap_start.CallAction1ID → invoke_transmit_start.CallAction1ID →
reply_transmit_end.CallAction1ID → call_getMap_end.CallAction1ID →
ADAPTER_CALLACTION1ID_TRANSMIT
□
awaitreply_getMap_start.CallAction1ID → invoke_transmit_start.CallAction1ID
→ reply_transmit_end.CallAction1ID → awaitreply_getMap_end.CallAction1ID
→ ADAPTER_CALLACTION1ID_TRANSMIT

```

(b)

Figure 6.13: Fragment of the CM-to-CSP transformation result (b) for communication weaving ‘*On getMap callAction1 RPC transmit*’ (a)

generated communication adapter for this weaving, all events in the adapter have its ID in the *callerID* parameter.

Our CM-to-CSP transformation creates further constructs for the communication weavings, which are very similar to those created for resource handling weavings (see Figure 6.12). We do not show them here to avoid the repetition.

As we have mentioned before, apart from the adapters, the CM-to-CSP transformation result for a configuration model contains a composition of all processes, that represent the individual layers models (woven models). This is a parallel process composition that we denote by a process called *LAYERS*. This composition does not define any synchronization alphabets between the individual processes, and, hence, is an interleaving. This is done to emphasize the absence of direct calls between the layers, which would need

to be synchronized in the resulting CSP specification. For the same reason, all adapter processes are also composed in an interleaved way in a process called *ADAPTERS*.

To represent the complete multi-layer system the above two processes *LAYERS* and *ADAPTERS* are composed in parallel defining our system process called *SYSTEM*. This parallel composition has a synchronization alphabet *Y* that is, like the adapter processes, also generated from the weavings of the configuration model. For our first example configuration model, shown in Figure 6.5 in Section 6.1.1, the generated *LAYERS* and *ADAPTERS* processes would look as follows:

```

LAYERS = (Mapplication || Mplatform_threads)
ADAPTERS = (ADAPTER_STARTACTION1ID_ACQUIRE || ADAPTER_ENDACTION1ID_RELEASE)

Adapters ADAPTER_STARTACTION1ID_ACQUIRE and ADAPTER_ENDACTION1ID_RELEASE, as described before, ensure that the invocation and response actions of MapDB.getMap() activity are only completed, when the activities of the ThreadPool.acquire() and ThreadPool.release() operations used were successfully completed. The SYSTEM process for the first configuration model example then combines the above processes as follows:

SYSTEM = (LAYERS || ADAPTERS)
           Y
Y = {invoke_getMap_start.x, invoke_getMap_end.x,
     reply_getMap_start.x, reply_getMap_end.x,
     invoke_acquire_start.StartAction1ID, reply_acquire_end.StartAction1ID,
     invoke_release_start.EndAction1ID, reply_release_end.EndAction1ID}

```

Thereby, the synchronization alphabet *Y* contains events to synchronize with the *MapDB.getMap()* activity on its invocation – sub-events *invoke_getMap_start.x* and *invoke_getMap_end.x* – as well as on its response – sub-events *reply_getMap_start.x* and *reply_getMap_end.x*. It also contains events to synchronize with *ThreadPool.acquire()* and *ThreadPool.release()* activities on their invocation and response – sub-events *invoke_acquire_start.StartAction1ID*, *reply_acquire_end.StartAction1ID*, *invoke_release_start.EndAction1ID*, and *reply_release_end.EndAction1ID*.

The first four events of *Y* can have any caller, which is represented by their parameter variable *x*. This parameter *x* allows us to synchronize this adapter with all the callers of *MapDB.getMap()* operation, including those from within its own layer model. The last four events of *Y*, on the other hand, have a fixed caller (*callerID* parameter) – the consumer actions (*StartAction1* and *EndAction1*) of the respective weavings.

For the second example configuration model, shown in Figure 6.6 in Section 6.1.1, the generated *LAYERS* and *ADAPTERS* processes would look as follows:

```

LAYERS = (Mapplication || Mplatform_RPC)
ADAPTERS = (ADAPTER_CALLACTION1ID_TRANSMIT)

```

This example configuration model only has one weaving and, thus, only one adapter process *ADAPTER_CALLACTION1ID_TRANSMIT*, whose structure we have already considered (see Figure 6.13). The *SYSTEM* process for the second configuration model example combines the above *LAYERS* and *ADAPTERS* processes as follows:

$$\text{SYSTEM} = (\text{LAYERS} \parallel \text{ADAPTERS})$$

$$Y = \{\text{call_getMap_start.CallAction1ID}, \text{call_getMap_end.CallAction1ID}, \\ \text{awaitreply_getMap_start.CallAction1ID}, \text{awaitreply_getMap_end.CallAction1ID}, \\ \text{invoke_transmit_start.CallAction1ID}, \text{reply_transmit_end.CallAction1ID}\}$$

Thereby, the synchronization alphabet Y contains events to synchronize with the *MapDB.getMap()* activity on its invocation by the *CallAction1* action (*consumer* in the communication weaving ‘On *getMap* *callAction1* RPC transmit’) as well as on its response to this invocation – sub-events *call_getMap_start.CallAction1ID* and *call_getMap_end.CallAction1ID* for the invocation, and sub-events *awaitreply_getMap_start.CallAction1ID* and *awaitreply_getMap_end.CallAction1ID* for the reply. Synchronization on these events is only performed if the caller of *MapDB.getMap()* operation is *CallAction1*. All other calls to this operation will need to be synchronized through other events with their caller IDs as a parameter.

The event set Y for the second configuration also contains events to synchronize with the resource – *RpcSender.transmit()* operation activity – on its invocation and response. Adapter *ADAPTER_CALLACTION1ID_TRANSMIT* then ensures that the invocation and response of the *MapDB.getMap()* operation by the weaving’s consumer *CallAction1* are only completed, when the *RpcSender.transmit()* activity was successfully completed on this invocation/response, respectively.

If we want to combine the generated CSP fragments for both example configuration models from Figure 6.5 and 6.6 into one, we just need to combine all layers models processes interleaved in the *LAYERS* process, combine all adapter processes also interleaved into the *ADAPTERS* process, and unite synchronization alphabets Y in the *SYSTEM* process.

Now, that we have considered two example results of our CM-to-CSP transformation, we can put together the complete set of requirements on it. We express them in a concise, complete, and example-independent form in Table 6.3.

No.	Model Element(s)	CSP Specification Fragment
1	Configuration model M_c	$\text{SYSTEM} = (\text{LAYERS} \parallel \text{ADAPTERS})$ $Y = \emptyset$ (synchronization events for the processes) $\text{LAYERS} = \text{SKIP}$ (layer processes for the woven models) $\text{ADAPTERS} = \text{SKIP}$ (adapter processes for the weaving)

Table 6.3: Overview of requirements on our CM-to-CSP transformation

No.	Model Element(s)	CSP Specification Fragment
2	Woven model M_{name} in M_c	$LAYERS = (LAYERS \parallel M_{name})$
Any Usage Weaving		
3	With consumer $Cons$ and resource Res .	$ADAPTERS = ADAPTERS \parallel ADAPTER_CONSID_RES$ $RES_CALLERS = RES_CALLERS \cup \{ConsID\}$
Resource Handling Weaving (consumer $Cons$, resource Res)		
4	$Cons$ is an initial action of activity A .	$ADAPTER_CONSID_RES =$ $invoke_A_start?x \rightarrow$ $invoke_Res_start.ConsID \rightarrow$ $reply_Res_end.ConsID \rightarrow$ $invoke_A_end.x \rightarrow ADAPTER_CONSID_RES$ $Y = Y \cup \{invoke_A_start.x, invoke_A_end.x,$ $invoke_Res_start.ConsID, reply_Res_end.ConsID\}$
5	$Cons$ is a final action of activity A .	$ADAPTER_CONSID_RES =$ $reply_A_start?x \rightarrow$ $invoke_Res_start.ConsID \rightarrow$ $reply_Res_end.ConsID \rightarrow$ $reply_A_end.x \rightarrow ADAPTER_CONSID_RES$ $Y = Y \cup \{reply_A_start.x, reply_A_end.x,$ $invoke_Res_start.ConsID, reply_Res_end.ConsID\}$
6	$Cons$ is an internal action in activity A .	$ADAPTER_CONSID_RES =$ $Cons_start?x \rightarrow$ $invoke_Res_start.ConsID \rightarrow$ $reply_Res_end.ConsID \rightarrow$ $Cons_end.x \rightarrow ADAPTER_CONSID_RES$ $Y = Y \cup \{Cons_start.x, Cons_end.x,$ $invoke_Res_start.ConsID, reply_Res_end.ConsID\}$

Table 6.3: Overview of requirements on our CM-to-CSP transformation

No.	Model Element(s)	CSP Specification Fragment
7	<i>Cons</i> is a call action of activity <i>calledAactivity</i> .	<pre>ADAPTER_CONSID_RES = call_calledActivity_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → awaitreply_calledActivity_end.ConsID → ADAPTER_CONSID_RES</pre> $Y = Y \cup \{call_calledActivity_start.ConsID, \\ awaitreply_calledActivity_end.ConsID, \\ invoke_Res_start.ConsID, reply_Res_end.ConsID\}$
8	<i>Cons</i> is an acquire action for resource <i>R</i> in activity <i>A</i> .	<pre>ADAPTER_CONSID_RES = call_acquireR_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → awaitreply_acquireR_end.ConsID → ADAPTER_CONSID_RES</pre> $Y = Y \cup \{call_acquireR_start.ConsID, \\ awaitreply_acquireR_end.ConsID, \\ invoke_Res_start.ConsID, reply_Res_end.ConsID\}$
9	<i>Cons</i> is a release action for resource <i>R</i> in activity <i>A</i> .	<pre>ADAPTER_CONSID_RES = call_releaseR_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → awaitreply_releaseR_end.ConsID → ADAPTER_CONSID_RES</pre> $Y = Y \cup \{call_releaseR_start.ConsID, \\ awaitreply_releaseR_end.ConsID, \\ invoke_Res_start.ConsID, reply_Res_end.ConsID\}$
Communication Weaving (consumer <i>Cons</i>, resource <i>Res</i>)		

Table 6.3: Overview of requirements on our CM-to-CSP transformation

No.	Model Element(s)	CSP Specification Fragment
10	<i>Cons</i> is a call action of activity <i>calledAactivity</i>. (<i>Res</i> used both on invocation and response of <i>calledAactivity</i>)	<pre> ADAPTER_CONSID_RES = (call_calledActivity_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → call_calledActivity_end.ConsID → ADAPTER_CONSID_RES) □ (awaitreply_calledActivity_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → awaitreply_calledActivity_end.ConsID → ADAPTER_CONSID_RES) Y = Y ∪ {call_calledActivity_start.ConsID, call_calledActivity_end.ConsID, awaitreply_calledActivity_start.ConsID, awaitreply_calledActivity_end.ConsID, invoke_Res_start.ConsID, reply_Res_end.ConsID} </pre>
11	<i>Cons</i> is an acquire action for resource <i>R</i> in activity <i>A</i>. (<i>Res</i> used both on invocation and response of acquisition of <i>R</i>)	<pre> ADAPTER_CONSID_RES = (call_acquireR_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → call_acquireR_end.ConsID → ADAPTER_CONSID_RES) □ (awaitreply_acquireR_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → awaitreply_acquireR_end.ConsID → ADAPTER_CONSID_RES) Y = Y ∪ {call_acquireR_start.ConsID, call_acquireR_end.ConsID, awaitreply_acquireR_start.ConsID, awaitreply_acquireR_end.ConsID, invoke_Res_start.ConsID, reply_Res_end.ConsID} </pre>

Table 6.3: Overview of requirements on our CM-to-CSP transformation

No.	Model Element(s)	CSP Specification Fragment
12	<i>Cons</i> is a release action for resource <i>R</i> in activity <i>A</i>. (<i>Res</i> used both on invocation and response of release of <i>R</i>)	<pre> ADAPTER_CONSID_RES = (call_releaseR_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → call_releaseR_end.ConsID → ADAPTER_CONSID_RES) □ (awaitreply_releaseR_start.ConsID → invoke_Res_start.ConsID → reply_Res_end.ConsID → awaitreply_releaseR_end.ConsID → ADAPTER_CONSID_RES) Y = Y ∪ {call_releaseR_start.ConsID, call_releaseR_end.ConsID, awaitreply_releaseR_start.ConsID, awaitreply_releaseR_end.ConsID, invoke_Res_start.ConsID, reply_Res_end.ConsID} </pre>

Table 6.3: Overview of requirements on our CM-to-CSP transformation

Next, we provide a short description and explanation of these requirements. The first requirement in Table 6.3 states that a configuration model is to be transformed into a CSP specification with three main processes *SYSTEM*, *LAYERS*, and *ADAPTERS*, which a structure and purpose as we have discussed before. The last two processes are initialized by an empty process *SKIP*, and the synchronization event set *Y* of the *SYSTEM* process is also initially empty. Later, transformation results for the different configuration model elements add processes and events to them.

Requirement 2 states that for each woven layer model M_{name} in the configuration model, a process (with the same name) is to be added to the *LAYERS* process. Analogously, requirement 3 states that for each weaving, an adapter process (*ADAPTER_CONSID_RES*) created for it is to be added to *ADAPTERS* process. Additionally, requirement 3 states that the ID of the consumer (*ConsID*) of the weaving has to be added to the set of callers of the resource (*RES_CALLERS*) of the weaving.

Further requirements 4 – 12 are concerned with the transformation of different types of usage weavings as follows: resource handling (requirements 4 - 9) and communication (requirements 10 – 12), respectively. Each of them states how the recursive adapter process (*ADAPTER_CONSID_RES*) for one particular combination of a weaving type and types of consumer/resource elements should be generated, and which synchronization events (from that adapter) should be added to the set *Y*. Since the requirements

within one usage type do not differ significantly, we only discuss one of them for each type to avoid the repetition. We have chosen requirements 7 and 11, since they were not considered in the previous examples.

Requirement 7 states that a resource handling weaving with a call action as a consumer is to be transformed into an adapter process, such that, its outer events are the sub-events representing the beginning (*call_calledActivity_start.ConsID*) and the end (*awaitreply_calledActivity_end.ConsID*) of this consumer action, and its inner events are the sub-events representing the beginning of the invocation (*invoke_Res_start.ConsID*) and the end of the response (*reply_Res_end.ConsID*) of the resource operation. For these four events the call action is the caller and, thus, its ID is stored in their parameter. Additionally, requirement 7 states that these events are added to the synchronization set Y without any changes.

Finally, requirement 11 states that a communication weaving with an acquire action as a consumer is to be transformed into an adapter process, such that, it contains two cases for the invocation and the response of the acquire action, that are composed via an external choice. Each of these cases describes an equivalent of a simple adapter process structure used for resource handling weavings, where the outer events are the sub-events representing the beginning (*call_acquireR_start.ConsID* and *awaitreply_acquireR_start.ConsID*) and the end (*call_acquireR_end.ConsID* and *awaitreply_acquireR_end.ConsID*) of the consumer action (here sub-action), and the inner events are the sub-events representing the beginning of the invocation (*invoke_Res_start.ConsID*) and the end of the response (*reply_Res_end.ConsID*) of the resource.

In analogy to requirement 7, for all events in the adapter generated in requirement 11, the acquire action is the caller and, thus, its ID is stored in their parameter. These events are also added to the synchronization set Y .

We have described how the individual layers models and their interactions can be modeled and transformed in a way that enables parallelization and re-use of the existing model transformations for the individual layers, and of their results. Later, in Chapter 7, we develop the described event splitting and CM-to-CSP transformations using our GBMT development approach with guaranteed transformation quality (see Chapter 5). Now, we still need to consider the remaining step of our weaving based approach to the parallelization and re-use of transformations – the composition of the obtained/re-used transformation results into the complete target model.

T_{Comp}: Composition and Re-Use of Transformation Results This last transformation has more of a technical supporting nature. With its help, we combine the transformation results for all layers models and for the configuration model. It allows us to re-use the transformation results for the individual layers models when the configuration changes, as long as the model is still a part of the changed configuration.

This model transformation is a straightforward composition. It simply combines the transformation results. It relies on the transformations of the individual layers models and of the configuration model to implement the compositional logic, based on the operators available in the target language, and does not create any further target elements of its own. When our weaving based transformation re-use approach is applied to the example multi-layer system and its transformation from Chapter 3, this final transformation simply combines all the transformation results in CSP.

6.2 Tool Support

To support the weaving based approach to the re-use of model transformations, we have designed a tool chain involving the existing state-of-the-art model weaver AMW [DFBJ⁺05] and the existing model transformations for the individual layers models of our example system from Chapter 3. Figure 6.14 shows this tool chain with the used artifacts for the case when our multi-layer system is modeled using the ADR language (see Section 3.1) for system layers and the weaving based configuration language for their interactions (see Section 6.1.1), and the CSP language (see Section 3.2) is chosen as the target language of the model transformation to be composed.

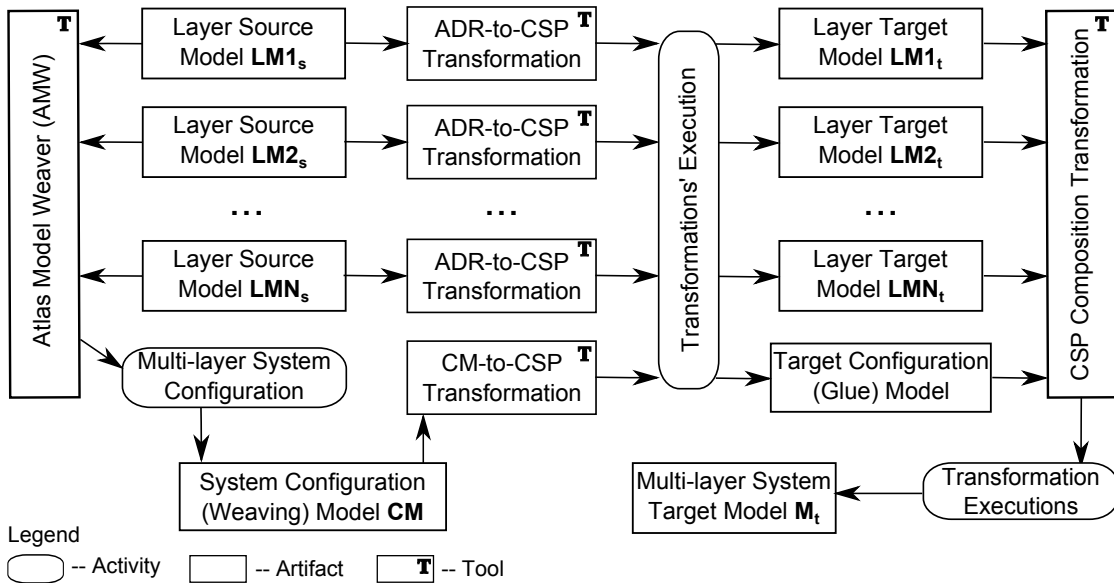


Figure 6.14: Tool support for the weaving based re-use of model transformations

This chain requires models of all system layers ($LM1_s, \dots, LMN_s$) to be modeled in some source transformation languages (here ADR). These models serve as input to the AMW tool, used to manually model interactions between these models in a separate configuration model CM . We have defined a language for such configuration models as

an extension of the core meta-model of AMW and presented it in Section 6.1.1. Thus, we do not repeat its description here and refer to Figure 6.3 for its description.

The layers models are transformed into the target language (here CSP) by our ADR-to-CSP transformation (see Section 3.3) in a modular way. Thereby, each model is transformed independently and arbitrarily interleaved with other model transformations. The results of these model transformations – target models $LM1_t, \dots, LMN_t$ – are then composed together through a separate composition transformation (here for CSP).

The previously created configuration model CM is also transformed into a CSP model by the respective CM-to-CSP transformation, described in Section 6.1.2 as T_{Conf} . Its result represents the glue CSP code, that is used to connect (via adapters) previously independent target models $LM1_t, \dots, LMN_t$ together into the final target model M_t for the complete multi-layer system. The glue code is added to the previously composed target models using the same composition transformation for CSP, mentioned before.

Although the above tool chain in Figure 6.14 is designed for the transformation re-use between the two specific source and target languages ADR and CSP, respectively, the chains for other source/target languages would look very similar. Only the individual transformations for the layers models and the configuration model would need to be exchanged with the ones for the chosen source and target language, and the respective composition transformation for the target language would need to be defined/found.

In our example chain, we have used ATL to implement all of the involved model transformations, including the composition. However, any other language for the development of model transformations with sufficient expressiveness can be used for this purpose. In Chapter 7, we show how our grammar based approach from Chapter 5 can be used for this purpose.

In order to transform the resulting CSP model M_t into its textual representation, to use as an input for CSP analysis tools like FDR, we have employed a dedicated model-to-text transformation language called MOFScript [MOFb] and the respective tool (not shown in Figure 6.14). To complete the picture we show examples of the MOFScript rules used.

```

1 csp.Prefix::printProcExpr() {
2   self.event.printProcExpr() '->' self.pe.printProcExpr()
3 }
4
5 csp.Skip::printProcExpr() {
6   'SKIP'
7 }

```

Listing 6.1: MOFScript model-to-text rules for *Prefix* and *Skip* CSP classes

Listing 6.1 shows the rules for *Prefix* and *Skip* classes of our CSP meta-model (see Section 3.2). These rules define how an instance of the respective class is represented in a textual form, either directly or through calling other MOFScript rules for its attributes

and references. Thereby, the rules are named the same to allow dynamic selection (overloading) of an appropriate rule for each sub-class of a process expression. In our example rules, such a mechanism is required to generate a textual representation of a prefix, since its process expression pe can be of any sub-type.

6.3 Related Approaches

Now, we compare our weaving based model and model transformation re-use approach with other re-use approaches, discussed at the beginning of this chapter and in Section 4.3. We also discuss further related approaches, that cover one or more aspects considered by our approach and/or use similar techniques.

Previously, we have described different combinations for the re-use of multiple models and model transformations in complex scenarios, like the transformation of our multi-layer system example from Chapter 3. There, we have classified our approach as the only one situated in the multiple models and model transformations (n Ms + m MTs) category, whereas, the remaining classified approaches [VVB07, CM09, KR10, ASWK11, SHNG12, NRKR13, Gro13] spread across the remaining three categories (1 M + 1 MT, n Ms + 1 MT, 1 M + m MTs). This classification shows the major difference between our and these approaches. Yet, to provide a more comprehensive differentiation, we compare each of them with our approach in more detail.

In [KR10], the authors follow the first direction (1 M + 1 MT) and use annotations to enrich the given application layer system model with the details from the chosen platform/infrastructure layers. In our approach, this information is modeled in separate exchangeable models. All models can then be related/configured in a flexible and language-independent way, by using model weaving, without the restrictions through the available feature model options. This way, we do not need an additional refinement transformation of the application layer model to obtain the complete system model. Due to the model composition method, selected in [KR10], the authors have to resort to a single model transformation. Such an option is not very flexible and does not allow direct re-use of single transformations for the individual layer models, like in our approach. Instead, the required model transformation has to be computed from the pre-defined fixed transformation fragments every time.

In [NRKR13, Gro13], the authors follow another direction (n Ms + 1 MT) and employ new types of infrastructure and resource interfaces, introduced to the modeling language used in [KR10]. These interfaces allow the authors to explicitly model all system layers and their interactions, in a separate configuration (assembly) model, like in our approach. This provides a more flexible solution than the one-model option. Yet, it requires all system layers to be modeled in one language, whereas our weaving based method allows us to model interactions between the layers modeled in different languages. On the model transformation side, these approaches use the same inflexible all-in-one transformation as in [KR10].

In [VVBJB07, CM09, ASWK11, SHNG12], the authors follow the direction of a single all-in-one model and multiple model transformations for the individual sub-models ($1\ M + m\ MTs$). In [VVBJB07, CM09], the authors focus on the composition of model transformations into chains by using traceability information, that is obtained through extending these transformations. This way, a transformation that depends on the result of another transformations can be re-used. Overall, these approaches can be useful if all transformations work on one model. Yet, they require significant adjustment of the re-used model transformations to produce/interpret the traceability information. Our approach can not be directly compared to [VVBJB07, CM09], since it considers a different scenario with multiple independent model transformations for separate individual models instead of one all-in-one model.

Like in [VVBJB07, CM09], in [SHNG12], the authors focus on the composition of model transformations through information from their context, such as the results of the application of other model transformations. Unlike the previous approaches, the authors do not require a developer to extend the individual model transformations to produce traceability information, in order to compose them. Instead, single model transformations are considered as black-boxes, that are encapsulated into composable modules with input/output parameters, and the management of traceability information is shifted to the framework. This approach allows sequential/contextual composition of the modules, and it can be considered as an improved and consolidated version of [VVBJB07, CM09].

In [ASWK11], the authors focus on the rule level composition of the individual model transformations into a single one, that produces the same result for a given all-in-one model. For this purpose, the authors repeatedly compute transitive model transformations from given ones. They assume a sequential transformation ordering and put further requirements on the individual transformations, like being exogenous (see Section 4.1.1). This approach is comparable with the superimposition approach, proposed in [Wag08] and discussed in Section 4.3.2. Our approach, on the other hand, is based on the assumption of multiple input models and independent individual model transformations for them, that can be executed in parallel. We do not compose either of these artifacts into a single one. Instead, we compose the transformation results. This composition does not require any further restrictions on the participating model transformations, but assumes a compositional nature of the target language.

Apart from the already mentioned [Wag08], in Section 4.3 we have introduced several transformation re-use approaches, that we now relate to our weaving based approach. The approaches [WH05, Old05, Kle06, VAVB⁺07, RRLB09, GdLK⁺10, HKA10, AEM12, EABP12, BPOB13], that propose to use various notations, like UML Activity diagrams, UML Component diagrams, and similar notations to define chains of model transformations and model their parameters, are complementary to our approach and can be combined with it. Their modeling methods can be used with our approach to describe the parallel composition of the model transformations of the individual layers and config-

uration models, and the sequencing of the event splitting and composing transformations (see Section 6.1.2 for details).

In [WH05, EABP12], the authors also propose approaches to address incompatibility of chained model transformations. Those methods can also be used complementary to our approach in the cases when the existing re-used individual model transformations, splitting (if used), and composing model transformations are not fully compatible. The approach in [WH05] is based on a side transformation to and from the transformation-specific meta-models for each model transformation in the chain. Such isolation of the transformation logic from its application context, like the meta-models of the source/-target languages, can extend the applicability of our approach to the previously non-reusable model transformations. The approach in [EABP12] focuses on the challenge of determining if two chained model transformations perform conflicting operations, like deletion of model elements required later. For this purpose, the authors translate model transformations and their meta-models into Alloy [Jac06] and employ its SAT solver.

Another approach, that can be combined with our approach to extend its applicability and to simplify the classification of the individual re-usable model transformations, has been proposed in [AEM12]. Its authors focus on the automation of model transformation chaining and propose to classify available individual model transformations using a feature model, analyze their constraints, dependencies, and compatibility. Based on this information, they allow the developer to select the features and, as a result, chain the model transformations that implement them. This method can be integrated with our approach to automatically obtain the model transformation chains for the individual models, like layers models in our example (see Section 3.1).

The approach proposed in [WKR⁺11] also has its focus on the re-use of model transformations. However, the authors achieve it not through composition, but through generic model transformations, that are defined over so-called generic meta-model concepts. Such generic transformations, combined with the generic-to-specific meta-model element mappings, constitute the sought specific model transformations. This approach is complementary to ours and can extend its applicability scope by simplifying the creation of the re-usable model transformations for the individual models, based on some generic transformation logic.

Finally, [SCGM08] is the last approach that we have discussed in Section 4.3 in detail. It is, amongst others, concerned with the introduction of inter-model element level links between the target models, that are generated by different re-usable model transformations from some input source model(s). The authors propose to create a new meta-model for bridging the existing target meta-models where for each class of elements, that needs to be linked with another class of elements, a new sub-class with the respective link is created. Such a new meta-model then contains all the required link types and is a basis for the model transformation integration, needed to create the links. The integration constitutes a separate transformation, that imports all individual ones and adds refinement rules, that are used to create the links. Our approach also deals with the element level inter-model links between the models. Primarily, we have considered source models

but can also apply it to the target models. Instead of new integration rules, that add the required links to the models, we use a non-invasive separate weaving model. Overall, our approach can be seen as an improvement of [SCGM08], that is enabled by the meta-model independence of weaving models.

Several further approaches were proposed to compose models [BSML07, AEN⁺08, KM10, JA10] outside of the context of model transformations and their re-use. In [BSML07], model compositions are realized via direct composition of their meta-models, which is quite straightforward, but difficult to maintain when models are exchanged. The authors of [AEN⁺08] propose a more generic and flexible model composition approach, that allows the definition of compositions in specific domains by establishing correspondence relations to a pre-defined generic domain-independent part (similar to [WKR⁺11] for model transformations). This way, the composition operators and strategies can be re-used. Yet, finding correspondences to the generic part can be quite difficult.

A modular model composition approach in [KM10] uses fragment-based glue models, similar to the weaving models we use, to connect the individual models. The connected models, however, have to conform to the same meta-model. Their mechanism for defining mappings between the glue model and the other models relies on fixed integration points within the models, that cannot be flexibly changed and have to be described in the meta-model. In [JA10], the authors explore a concern-based composition of heterogeneous models with integration points within these concerns. As for the other details, this approach is similar to [KM10].

Like in [JA10], the approach presented in [BL08], also addresses composition of heterogeneous models. The authors, however, deal with the issue by employing ontologies to integrate different meta-models based on common concepts (analogous to the generic part in [AEN⁺08]). They use the meta-model level instead of the model level integration points. There exist further techniques that propose meta-model level mappings based on other concepts, like dedicated mapping components [KWRS11] and TGGs [HEO⁺11].

Only a few [KM10, JA10] of the considered model composition approaches address the model level connection of conceptually different, but related model elements, like consumer and resource elements in our weaving based configuration models (see Section 6.1.1). All of them require changes to the composed models and/or their meta-models to add the integration points or classes. For this reason, we have chosen the non-invasive model weaving technique with an extendable meta-model for the links (see Section 6.1.1), in order to model interrelations between the elements of the individual models in a separate model instead of their composition into an all-in-one model.

6.4 Evaluation and Discussion

After giving a general description of our weaving based model and model transformation re-use approach and comparing it to the existing state-of-the-art, we evaluate its benefits

and shortcomings based on the concrete examples. The example models and model transformations, that we use for this purpose, are part of our multi-layer system and its transformation example from Chapter 3.

For the purpose of this evaluation we consider the following various configurations of the multi-layer system, partially inspired by [FEK⁺07]: the single application layer; the application layer and the platform layer with the thread pool of various capacity; the application layer and the platform layer with a simple local call or RPC based [BN84] communication; and the application layer and the platform layer with the thread pool of various capacity and the RPC based communication. Table 6.4 shows these configuration combinations in its left column.

We use the trip planning software system model ($M_{application}$) as the application layer model, the thread pool model ($M_{platform_threads}$) as one of the platform layer models, and a local call model (contains only trivial activities with initial/final nodes) or the RPC communication model ($M_{platform_RPC}$) as another platform layer model. Our method does not limit the number of models for any layer. All of these models were discussed in detail in Chapter 3.

Apart from the various thread pool capacities – sufficient and insufficient for our example system – we also consider two versions of RPC communication – correct and erroneous. In the erroneous RPC model we inject an infinite loop, as an alternative to the correct call action to the connected receiving component, into the activity for *RpcSender.transmit()* operation. This way, a potential livelock is introduced. A livelock occurs when the (multi-layer) system does not make any progress, but is not in a deadlock. We model the described options in otherwise identical thread pool and RPC models, respectively.

The above mentioned factors are very likely to affect the quality properties of the applications, like the trip planning software, running on top of such platforms. Here, we consider how the application’s deadlock and livelock freedom is affected in such different configurations. For this purpose, as discussed in Section 3.3, the configurations have to be translated into an appropriate analysis language, like CSP in our example. To analyze the generated CSP models of the configurations, we use CSP refinement checker FDR, that provides, among others, deadlock and livelock checks.

At this point, we focus on the application of our weaving based model and model transformation re-use approach to the analysis scenario described above. First, we model the described configurations using our configuration modeling language (see Section 6.1.1) and, thereby, create six configuration weaving models that reference two or more of the layers models – $M_{application}$, $M_{platform_threads}$ (with respective capacity), and $M_{platform_RPC}$ (correct or erroneous version). In the cases when the thread pool is used, we assume that each operation execution (except for *acquire()* and *release()* operations of *ThreadPool*) requires a thread on invocation and releases it on response. In the cases when RPC communication is used, we assume that each invocation and reply uses this communication method. Thereby, the RPC operations also require threads when used

together in a configuration with a thread pool. Examples of such configuration weaving models with two layers models for our scenario have been provided in Section 6.1.1 (see Figures 6.5 and 6.6). The modeling of configurations with three or more woven layers models is analogous.

System Configurations	deadlock?	livelock?
Application	no	no
Application + thread pool (1 thread)	yes	no
Application + thread pool (2 threads)	no	no
Application + local call	no	no
Application + RPC (erroneous)	no	yes
Application + RPC (correct)	no	no
Application + thread pool (3 threads) + RPC (correct)	yes	no
Application + thread pool (4 threads) + RPC (correct)	no	no

Table 6.4: Considered multi-layer system configurations and values of sample properties

Once the system configurations of interest have been explicitly modeled using weaving models, we can consider the transformation of the resulting multi-layer system models into CSP. In our approach, the complete transformation constitutes several layers models transformations and a configuration weaving model transformation, that are independent of each other and can be executed in parallel (see Figure 6.1). How these individual reusable model transformations are implemented does not play a role, since only their results need to be composable and need to represent the same layers model in CSP.

The first condition is ensured by the composite nature of CSP and the operators it provides (see Section 3.2). The second condition has to be ensured through an appropriate specification of model transformations requirements and their correct realization in the model transformation design and implementation phases. We have provided such specifications of requirements in Tables 3.1, 6.2, and 6.3 for the example ADR-to-CSP layer model transformation (first part of T_{LModel} in Section 6.1.2), the event splitting transformation in CSP (second part of T_{LModel}), and the CM-to-CSP configuration weaving model transformation (T_{Conf}), respectively. These specifications are expressed as informal mappings between the source and target language elements and structures, that have equivalent meaning. The judgment about their equivalence has been made based on the descriptions of the languages.

Although the definition of formal semantics for the involved languages and the specification of the transformation mappings based on them is possible, it is bound to involve additional complex steps, that we do not address in this thesis. Thus, we also do not provide proofs of semantic equivalence of the CSP results of such transformations with the input source models. For an introduction to the topic of semantic equivalence/preservation in model transformations and how it can be proven, we refer the reader to [Sem12].

With respect to the implementation of the mentioned model transformation specifications, we have already considered an example implementation of the ADR-to-CSP transformation using ATL and TGGs in Section 4.1.3, and using our grammar based approach in Section 5.2.1. Later, in Sections 7.1 and 7.2, we provide implementation examples of both the event splitting and the CM-to-CSP configuration model transformations, using our grammar based approach to the model transformation development.

The results of the execution of these transformations is later composed into one final CSP model per configuration, that we use for the analysis of deadlock and livelock freedom. The results of this analysis for our sample configurations are also presented in Table 6.4 in the respective columns. Thereby, for our experiments, we currently consider only a single user invoking the modeled multi-layer trip planning system.

As Table 6.4 shows, the application layer model $M_{application}$ is both deadlock- and livelock-free. Connected to the thread pool model $M_{platform_threads}$, it deadlocks depending on the number of threads available. As operations that require a thread might be executed inside one another ($getTrip()$ with $getMap()$ and $getLocalTransport()$ or $getRoute()$), providing only one thread, directly acquired and held by operation $getTrip()$, leads to a deadlock. In the case, when at least two threads are provided, no deadlock occurs, since for the execution of operations $getMap()$ and $getLocalTransport()$ (or $getRoute()$) one thread combined with blocked waiting is sufficient.

The application model connected to the trivial communication model $M_{platform_localcall}$ is proved correct, while the erroneous RPC model $M_{platform_RPC_err}$, contains the expected livelock. The corrected RPC model $M_{platform_RPC}$ fixes this. The configurations containing the trip planning application, the thread pool, and the RPC communication model, where the communication operations consume threads too, is analyzed with the following expected results: a deadlock is detected if the thread pool model does not provide enough threads (four threads in this case). Although, on the relatively simple example configurations considered the results obtained are as expected, more complex case studies, like [FEK⁺07], have shown that such analysis is essential to obtain correct multi-layer system configurations (e.g., thread pool capacity).

We have demonstrated that the analysis of the application layer alone is not sufficient, as other layers models, like platform layer models with thread pools and RPC communication, may influence the deadlock- and livelock-freedom of the complete system. Our weaving based approach allowed us to make this analysis process more efficient and flexible. It lets us combine different layers models into a configuration in a non-invasive and flexible way, and re-use the existing model transformations for the individual layers models and the configuration model without having to change them. Furthermore, when only some of the layers model needed to be changed in the configuration, our approach allowed us to re-use the transformation results for the models, that were left unchanged. The same is true, if some model transformations involved needed to be exchanged.

On the other hand, the same flexible non-invasive weaving based configuration modeling approach, that we use, can bring certain disadvantages. With the growing complexity

of the involved models and interactions between their elements, manual adjustments of the used configuration meta-model and modeling of the configurations can become impractical. Since one of the basic assumptions of our approach is that the individual layers models contain no information about their interactions, we can not directly automate the generation of the configuration models through some kind of dependency analysis between them. One possible solution to this issue is to provide a set of pre-defined interaction scenarios to lead the generation of configuration models. We consider this and other possible improvements of our weaving based re-use approach in Section 8.4.

Another essential and potentially complex task of our weaving based approach is the definition of model transformations for the configuration models. This task has to be performed once for each combination of a configuration language and a target language. Thereby, expertise in both languages and, especially, in the compositional mechanisms of the target language, is required. In the cases, when one configuration language is an extension of the other one, with an existing transformation into the target language, some degree of re-use of this model transformation can be possible.

In this chapter, we have defined the following languages of the configuration transformation by meta-models: the configuration languages as an extension of the core weaving meta-model (see Section 6.1.1) and the target language, here CSP, by the CSP meta-model from Section 3.2. Later, in Section 7.2, we consider a grammar based definition of such model transformation for our configuration language, using the approach from Chapter 5.

Chapter 7

Weaving Based Re-Use of Grammar Based Model Transformations

Up to now, we have considered how a single model transformation can be developed using our grammar based approach (see Chapter 5) and how a transformation can be re-used in our approach based on weaving models (see Chapter 6). Now, we consider a combination of these approaches.

We apply the previously proposed GBMT approach to develop the model transformations that facilitate the use of our weaving based re-use approach. Since these model transformations are scenario, as well as source and target language specific, we apply the GBMT approach to develop such transformations for our example multi-layer system. The requirements on these transformations for the individual layers models and for the weaving configuration models were described in Section 6.1.2. Now, we develop two GBMTs that fulfill these requirements and also have guaranteed quality properties by construction (see Section 5.3).

First, in Section 7.1, we define the grammar based transformation for the event splitting in CSP ($T_{EventSplit}$), that allows parallel transformation of the system layers models with later composition of the results. Then, in Section 7.2, we define a grammar based CM-to-CSP transformation (T_{Conf}) for configuration models.

7.1 Grammar Based Transformation of Layers Models

The requirements on this transformation, referred to as the *event splitting* transformation, were described in Table 6.2 in Section 6.1.2. This transformation operates between CSP models and, thus, we require a CSP HR grammar to define the respective GBMT. For this purpose we use the CSP HR grammar introduced at the end of Section 5.1.2.

The complete GBMT definition of this event splitting transformation consists of the following two parts: the GBMT rules that describe the changes produced by the transformation as required in Table 6.2; and the GBMT rules that keep the rest of the model.

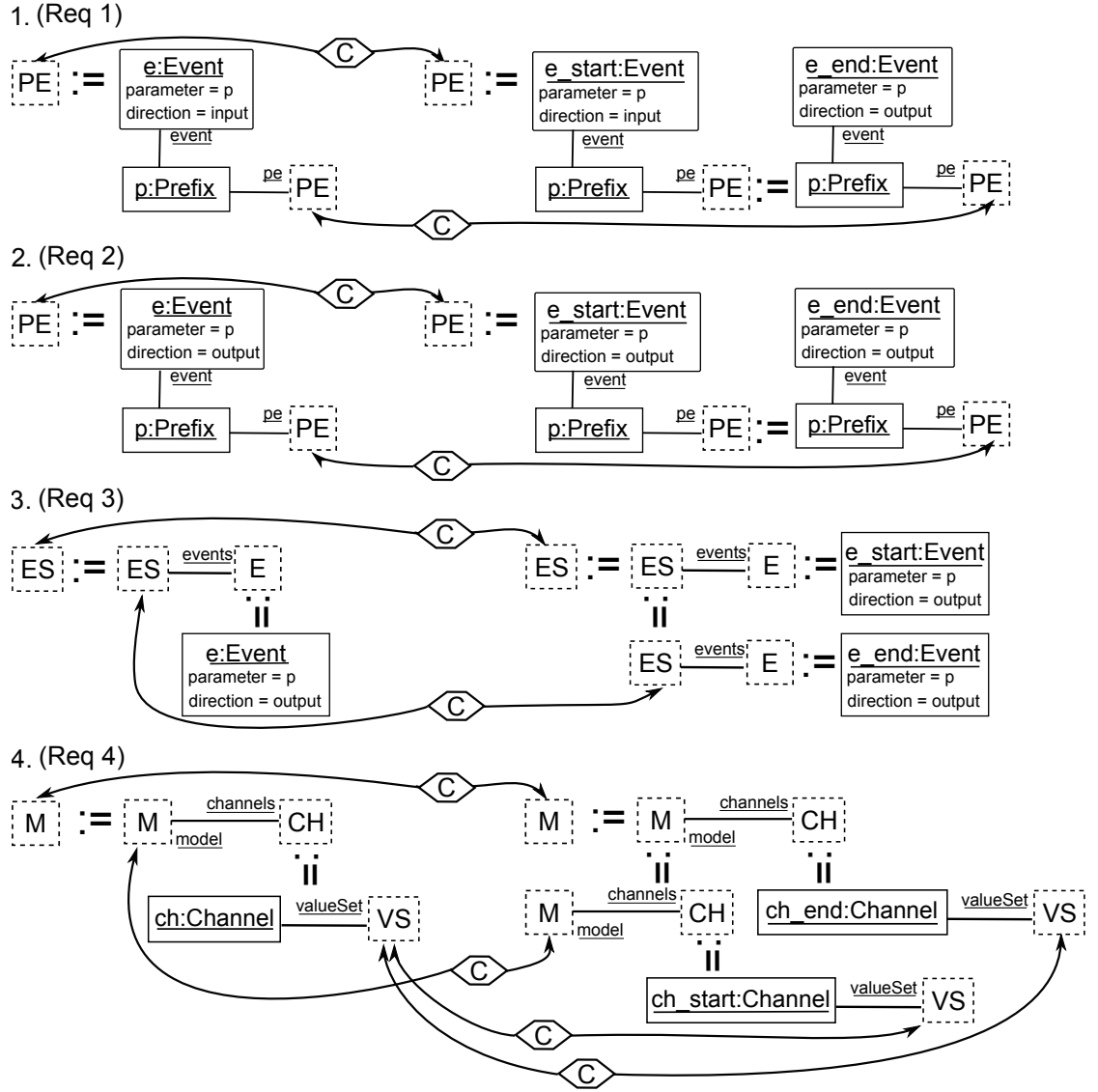


Figure 7.1: Grammar based definition of the event splitting transformation in CSP

All the rules of the first *changing type* are shown in Figure 7.1. Next to their numbers, we indicate which requirements they implement. All of these GBMT rules relate multiple source and/or target CSP productions (extended GBMT rules), but none of them are non-local.

The first two GBMT rules (1 and 2) simply relate the source prefix production with an event e to the target derivation sub-tree that produces a sequence of two prefixes with the corresponding e_start and e_end sub-events and with their parameters set according to the requirement (see Table 6.2). The created correspondence mapping of type C between

the leaf hyperedges of type PE , that represent the next process expressions, ensures that the created sub-events are at the same position in a process definition as the event e .

Rule 3 relates the source derivation sub-tree for an event e in an event set, represented by a non-terminal hyperedge of type ES , to the target derivation sub-tree, that produces two sub-events e_start and e_end with the same parameters and in the same set as e . The created correspondence mapping of type C between the leaf hyperedges of type ES ensures that the created sub-events are in the same set as the original event e .

Finally, the last GBMT rule, number 4, relates the source derivation sub-tree for a CSP channel ch to the target derivation sub-tree for two channels ch_start and ch_end . Analogously to rule 3, the correspondence mappings of type C between the leaf hyperedges of type VS , representing a value set of a channel, ensures that the generated channels have the same value set as the original channel ch . The M -to- M correspondence mapping in this rule ensures that the re-created leaf hyperedges representing models (of type M) stay related for the later application of the rules of this GBMT.

As for the second type of GBMT rules – the *non-changing rules* that keep the rest of the input model the same – they are defined by simply relating each source production, that has not been used as the first production in the source sub-trees of the rules of the first type, to itself. This would include all productions of our CSP HR grammar (see Section 5.1.2, Figure 5.5), apart from the productions for a prefix (productions 11, 12), the productions for an event in an event set (production 20), and the production for a channel in a model (production 13).

Figure 7.2 shows an example of such self-related production in the non-changing GBMT rule for the production for a process (production 4). Note that, the leaf hyperedges of the production are also connected to themselves via a correspondence. These non-changing GBMT rules do not cause any additional development complexity, since they can be generated automatically for the selected productions.

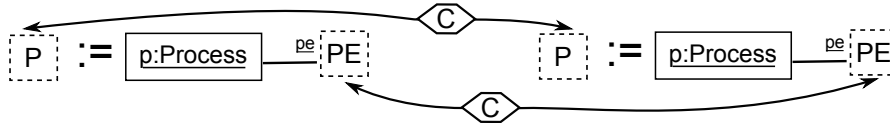


Figure 7.2: Non-changing GBMT rule for the process production from the CSP HR grammar

The last two GBMT rules of the first type, number 3 and 4, in Figure 7.1 have source derivation sub-trees with more than one production. This fact requires the defined GBMT to fulfill the additional conditions, stated in Section 5.3.1, in order to have the desired quality properties – terminate, be sound and complete, and deterministic. Next, we consider these and further conditions, and show that the provided grammar based definition of the event splitting transformation fulfills them.

Since the defined GBMT does not have any non-local rules, we only need to consider the conditions stated in Definition 5.3.1.9 in Section 5.3.1. It requires each rule of our event splitting GBMT to either have the basic or the extended form, defined in Definition 5.3.1.1 and 5.3.1.5, respectively. Since Definition 5.3.1.1 is a special case of Definition 5.3.1.5, we only need to show that the latter is fulfilled. Also, Definition 5.3.1.9 puts two additional conditions on the GBMT, which we consider later.

First, we look at each of the changing rules from Figure 7.1 and at one non-changing rule from Figure 7.2, and show that they fulfill the conditions put on individual GBMT rules. Each of these rules relates finite source and target derivation sub-trees (in some cases only with one production), and has a correspondence between their roots. The correspondence functions defined in these rules map each leaf non-terminal hyperedge of the respective target derivation sub-tree to a correspondence type and a leaf non-terminal hyperedge of the related source derivation sub-tree. Plus, in a rule, none of these target leaf hyperedges are mapped onto more than one correspondence type and source leaf hyperedge pair. Thus, these correspondence mappings are functions and are total.

As for condition 1 inherited by Definition 5.3.1.5 through the extension of Definition 5.3.1.1, it holds since the production with the start non-terminal at the left-hand side ($S := M$) is only related to itself in one non-changing GBMT rule, and we only have one correspondence type C (which is also $t_c^{default}$). Thus, all changing rules and one non-changing rule of our event splitting GBMT fulfill Definition 5.3.1.5. For the rest of the non-changing rules we can show this analogously and, thus, all rules of the event splitting GBMT fulfill this.

Now, we need to show that the event splitting GBMT fulfills both conditions stated in Definition 5.3.1.9. Thereby, we consider all of its rules to have the same priority. All of the changing rules of our event splitting GBMT (Figure 7.1) have distinct source derivation sub-trees and none of them is a prefix of the other one. The non-changing rules of our event splitting GBMT are defined for all productions in the source HR grammar (CSP HR grammar), which are not the first productions in the source derivation sub-trees of the changing rules. Their source derivation sub-trees consist of distinct single productions and, thus, none of them is a prefix of the other, and, due to the selection of these productions, none of them is a prefix nor an extension of any source derivation sub-tree of any changing rule. Thus, condition 1 of Definition 5.3.1.9 holds.

The second condition (2) of Definition 5.3.1.9 requires the event splitting GBMT to have exactly one well-formed rule set ($RS_{p_s, c_{pre}}$) for each combination of a source production (p_s) from the source HR grammar and a type of correspondence (c_{pre}) that a hyperedge of the type of its left-hand side (A_s) might have. By construction, our GBMT has exactly one type of correspondence (C) for each source non-terminal and exactly one rule for each combination of a source production p_s (first production in the source sub-tree of that rule) and such correspondence type for its left-hand side. Thus, there

is exactly one rule set with one rule for each combination of p_s and c_{pre} . Now, we need to show that these rule sets are well-formed (see Definition 5.3.1.8).

This holds for all rule sets formed by respective non-changing rules, since they contain special rules with one-production source derivation sub-trees (Definition 5.3.1.8, 1). For the rule sets formed by the first two changing rules (rules 1 and 2) the same is true. The remaining rule sets formed by extended GBMT rules 3 and 4, however, still need to be considered, to show that they fulfill the coverage criteria (Definition 5.3.1.8, 2).

The rule set formed by rule 3 contains only this rule. By Definition 5.3.1.8, it also has to contain all replacement alternatives (see Definition 5.3.1.6) for its source derivation sub-tree. In our CSP HR grammar, the source sub-tree of rule 3 has no replacement alternatives. Thus, this rule set fulfills the coverage criteria and is well-formed. The well-formedness of the rule set formed by rule 4 is shown analogously.

We have shown that the event splitting transformation $T_{EventSplit}$ for CSP models, described in Section 6.1.2, can be defined using our GBMT development approach, and that this definition is conform with the formal requirements on a GBMT (see Section 5.3.1). This allows us to guarantee the desired quality properties, from Section 4.2, for it.

Now, we can (re-)use our event splitting GBMT in a sequence after the ADR-to-CSP GBMT, defined in Section 5.2.1, to transform the individual layers models of our example system (see Section 3.1). Since both of these GBMTs have been shown to fulfill the conditions required to guarantee their quality properties, their sequence also has these quality properties. Of course, this only holds provided that the target language of the first GBMT is the source language of the second one. This is true in our example.

7.2 Grammar Based Transformation of Configuration Models

To complete the picture, we still need to develop a GBMT to transform configuration models into CSP. The requirements on our transformation of configuration models into CSP, referred to as *CM-to-CSP* transformation, were described in Table 6.3 in Section 6.1.2. This transformation operates between our system configuration language, that was described by an extension of the core weaving meta-model of AMW (see Section 6.1.1, Figure 6.3), and CSP. Thus, to define this transformation using our GBMT development approach, we need both an HR grammar for the configuration modeling language and a CSP HR grammar. As before, for the CSP language we use the HR grammar introduced in Section 5.1.2. As for the configuration modeling language, in Section 7.2.1 we introduce an HR grammar for it. Then, in Section 7.2.2, we use these grammars to define our CM-to-CSP transformation with the grammar based approach.

7.2.1 Grammar of Configuration Modeling Language

Our system configuration language was described as an extension of the core weaving meta-model of AMW in Section 6.1.1. This meta-model represents all used elements of the woven layers models, that are involved in the usage weavings, by one class called *UsedModelElement* (see Figure 6.3). However, our transformation requirements in Table 6.3 demand different rules for the different types of model elements used.

These requirements specify all acceptable types of *consumer* and *resource* elements that can be used in the different types of *usages* (*resource handling* and *communication*). They also require some additional information from the inside of the woven models, e.g., the name of the containing activity for an initial or a final node, in order to produce the desired transformation result. To access the information about the used elements and identify their type, one can employ the dereferencing mechanism of AMW (see Section 6.1.1) and provide its support in our GBMT development approach. Alternatively, one can enrich the configuration meta-model with this transformation relevant information.

The first alternative allows us to keep the configuration meta-model intact and avoid the information duplication, but requires significant changes to the GBMT development approach to allow the use of the dereferencing mechanism in the rules. This would include both changes to the approach formalization as well as its tool support, including the parsing mechanism. The second alternative allows us to keep the GBMT development approach unchanged, but requires changes to the configuration meta-model.

Although, both options have their benefits, and the first option might provide a better long-term solution, for our transformation example, we pursue the second alternative. We duplicate transformation relevant information from the woven models in an *extended system configuration meta-model*, shown in Figure 7.3. In this extended configuration meta-model we distinguish between different types of used model elements, modeled as sub-classes of the *UsedModelElement* class. These types include an activity, a passive resource, different types of activity nodes (with the containing activity) – an initial node, a final node, an internal action, a call action (with the called activity), as well as acquire and release actions (with the related resource). In the extended meta-model these types are represented by the classes with the same names.

Such distinction between the different types of woven model elements allows us to restrict the types of elements that can have the *consumer* or the *resource* role in a usage weaving¹. Now, we can explicitly specify that a resource is always an activity (class *Activity*) and a consumer is, in the case of a resource handling usage (class *ResourceHandling*), any activity node (class *ActivityNode*) and, in the case of a usage for communication (class *Communication*), only a so-called communication activity node (*CommunicationNode* class). Call actions, and acquire and release actions belong to such communication nodes, since they interact with other elements of their layer's model. We

¹Provided that the elements of correct types are assigned to the (used) elements of a usage weaving.

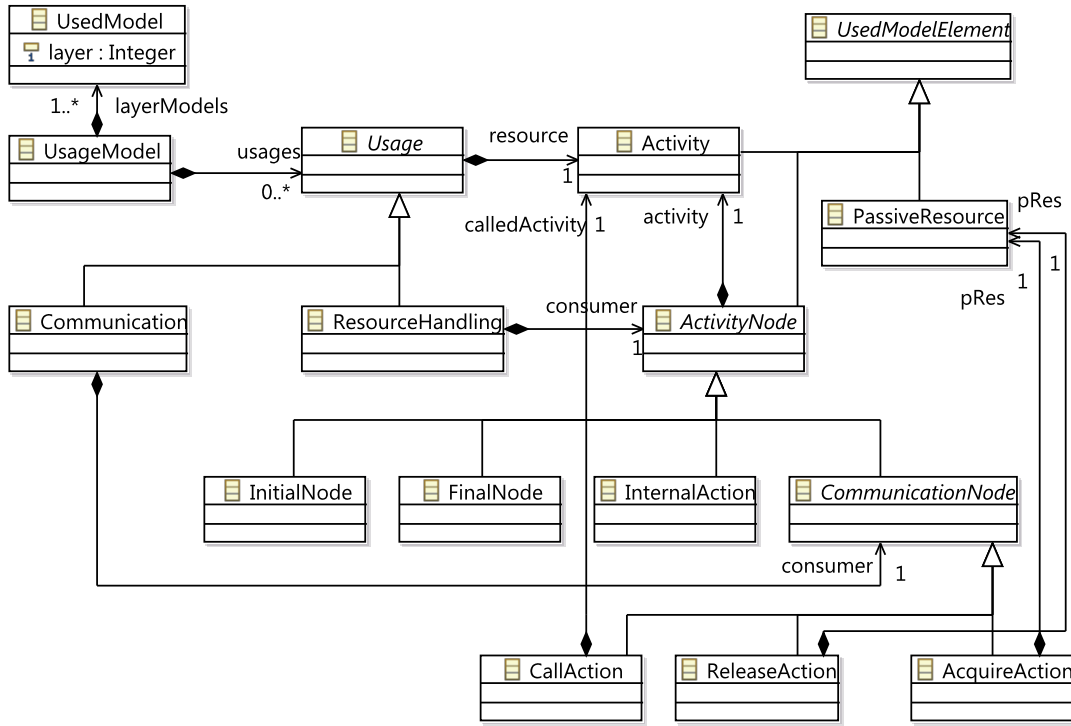


Figure 7.3: Extended system configuration meta-model in Ecore

would not be able to specify these types of constraints for the consumer and resource roles in the original configuration meta-model from Section 6.1.1 (see Figure 6.3).

Based on the extended configuration meta-model in Figure 7.3, we have defined a *configuration* HR grammar to employ in our GBMT definition process. Figure 7.4 shows productions of this HR grammar. It contains the initial production for a usage configuration model non-terminal M (1), the productions to create a usage configuration model instance (2) from a hyperedge of type M , and to add references to the used woven models (3) and to the non-terminal hyperedges of type U representing usages (4) to it.

Furthermore, this configuration HR grammar contains the productions that create instances of different types of usages (5 and 6) with the related resource *res*, *consumer* activity node non-terminal hyperedges (of type AN or CN), and activities a containing these consumer nodes. Non-terminals AN and CN are then used by productions 7 – 15 to create different sub-types of consumer activity nodes.

Note that with the productions, defined in such a way, our configuration HR grammar generates models with multiple elements representing the same (by name) activity/passive resource. This duplication is employed in order to represent references to the activities/resources and to avoid the use of context-sensitive HR productions [DHM12], that are not directly supported by our approach at this point, for this purpose. The use of

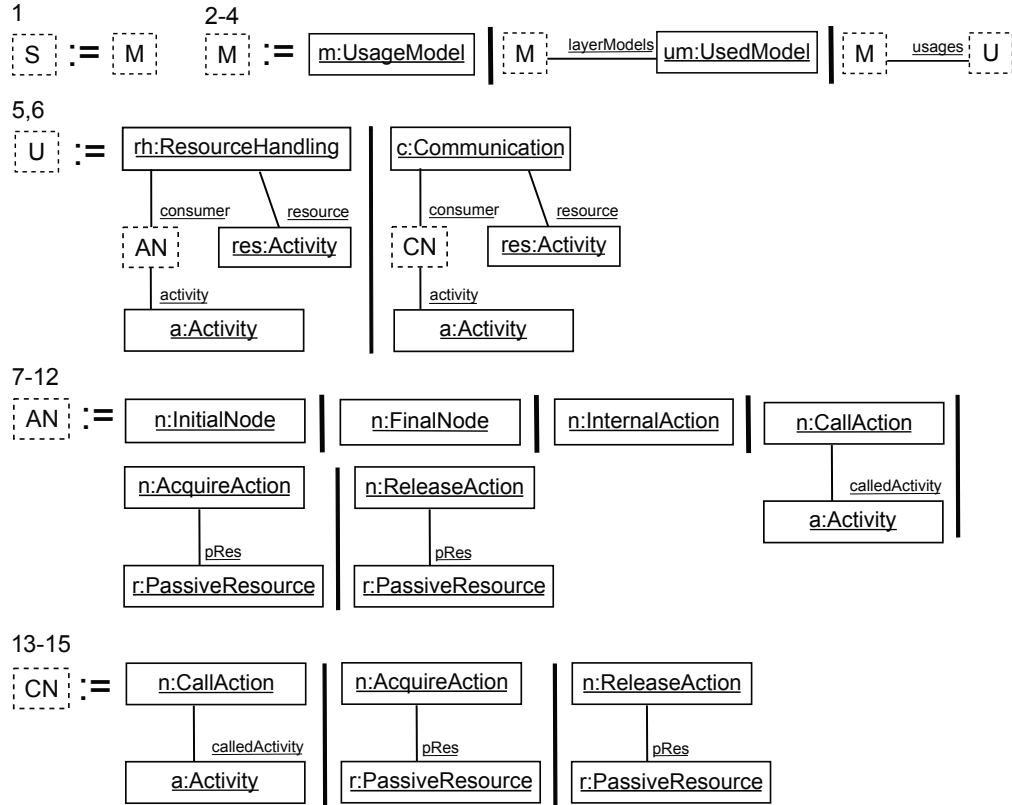


Figure 7.4: Extended system configuration HR grammar

such mechanism requires pre-processing of source configuration models before the parsing, in order to untangle the modeled references to the activities/resources and convert the given graph-based model representations into the tree-based ones with duplicates.

Now, that we have introduced an HR grammar for configuration models, we can proceed with the definition of the CM-to-CSP transformation using the GBMT approach.

7.2.2 Definition of the Configuration Model Transformation

Definition of a GBMT of configuration models into CSP according to the requirements stated in Table 6.3 is more complex than the one defined in Section 7.1 for the event splitting transformation. Its complexity is comparable to the one of the ADR-to-CSP GBMT for the individual system layers, that we have considered in Section 5.2.1. Yet, the requirements on our CM-to-CSP transformation are more repetitive in their structure. Each usage weaving should be transformed into an adapter process and an extension of the event set Y , whose events only slightly differ in their names and consumers. Thus, to keep the description of the CM-to-CSP GBMT concise, we only demonstrate the definition of one rule per usage type. The rest of the rules are described in full.

First, we consider the rules of the CM-to-CSP GBMT that do not transform usage weavings. These rules address requirements 1 and 2 from Table 6.3. Figure 7.5 shows them annotated by the requirements that they cover in round brackets (Req.).

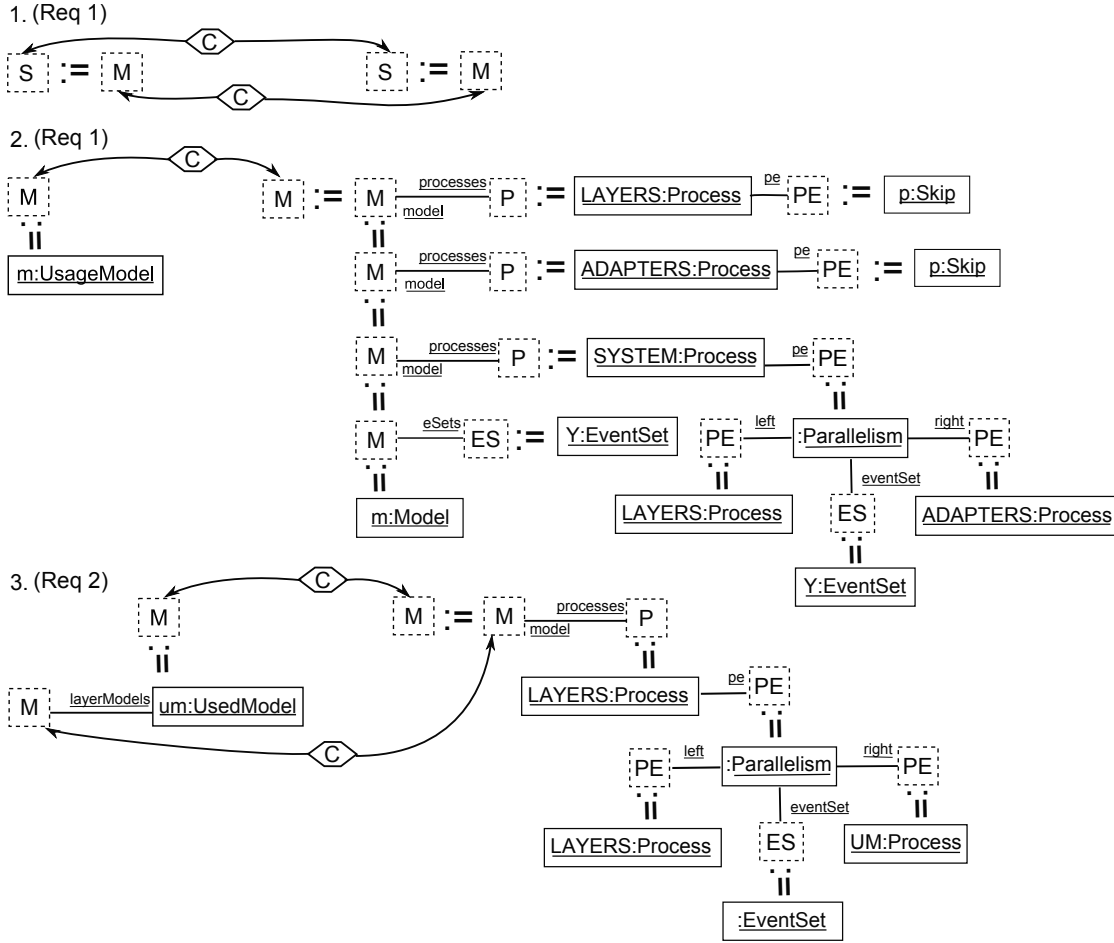


Figure 7.5: GBMT rules for requirements 1 and 2 on the CM-to-CSP transformation

The first rule in Figure 7.5 simply creates a target model non-terminal hyperedge of type M when the source model non-terminal hyperedge of type M is produced. These non-terminals are linked via a correspondence of type C which is used as a basis for the application of other GBMT rules including rules 2 and 3.

The second rule in Figure 7.5 states that, when an instance of a usage (configuration) model has been produced by rewriting a source non-terminal hyperedge of type M , a number of CSP structures should be created in the target model using the corresponding target non-terminal hyperedge of type M (M -to- M correspondence of type C). This rule produces processes $LAYERS$, $ADAPTERS$, $SYSTEM$, and an event set Y according to requirement 2. It also produces an instance of the target model by rewriting the remaining target non-terminal hyperedge of type M . If this last production would not

be added, the rule would violate the conditions that we impose on the form of GBMT rules (see Section 5.3.1). We discuss this topic with regard to our CM-to-CSP GBMT in more detail later.

The last rule in Figure 7.5 states that, when an instance of a used (woven) model has been produced by rewriting a source non-terminal hyperedge of type M , the CSP process $LAYERS$ should be re-defined in the target model using the corresponding target non-terminal hyperedge of type M . Process $LAYERS$ is re-defined by composing it in parallel with a new process UM , that represents the used model. This parallel composition has an empty synchronization alphabet (event set). The definition of the UM process can be obtained through the transformation of the used layer model by the ADR-to-CSP GBMT defined in Section 5.2.1, followed by the event splitting GBMT defined in Section 7.1. This rule also adds a correspondence of type C between the newly produced source and target non-terminal hyperedges of type M for GBMT rules.

Next, we discuss the rules that cover the rest of the requirements starting with requirement 3. These requirements are covered by multiple GBMT rules. There is one *common rule* that is usage weaving, resource, and consumer type independent and that does not require to know the values of their attributes. Plus, there are *type specific rules* for each combination of a usage weaving, resource, and consumer type allowed by our configuration language (see Figure 7.3).

4. (Part of Req. 3-12)

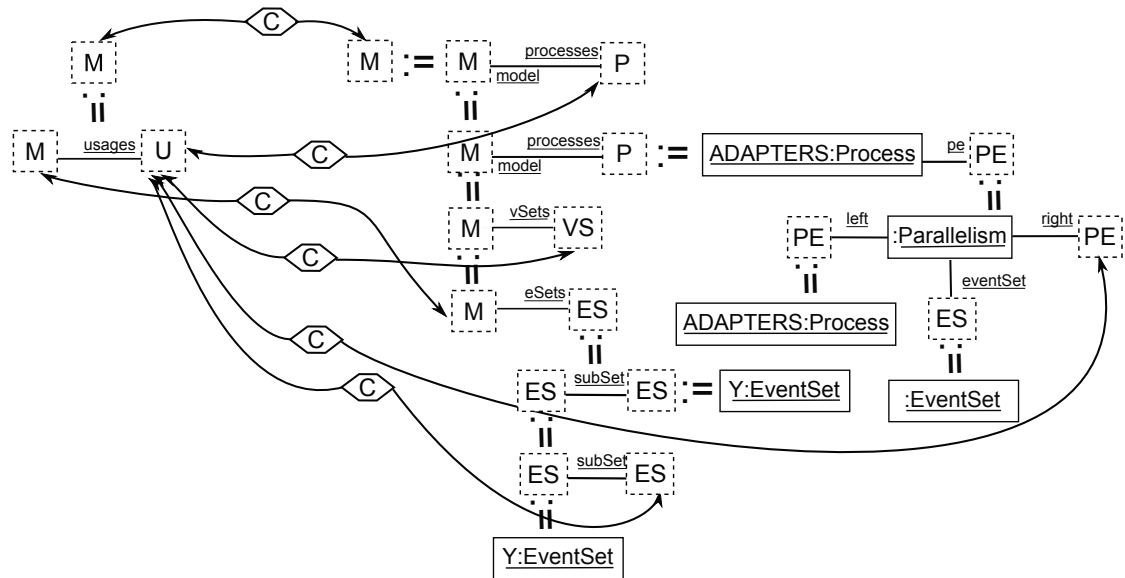


Figure 7.6: Common GBMT rule for requirements 3 – 12 on the CM-to-CSP transformation

First, we consider the common rule, shown in Figure 7.6, that produces the (common) CSP structures created in each of the type specific requirements (4 – 12) and in require-

ment 3. This rule states that, when a new usage represented by a non-terminal hyperedge of type U has been produced by rewriting a source non-terminal hyperedge of type M , a number of CSP structures should be created in the target model by rewriting the corresponding (M -to- M correspondence of type C) target non-terminal hyperedge of type M . It produces the following two processes: one is represented by a leaf non-terminal hyperedge of type P , and another one is a re-definition of the process *ADAPTERS* through its parallel composition with some process expression, represented by a leaf non-terminal hyperedge of type PE . It also produces a values set, represented by a leaf non-terminal hyperedge of type VS , and an event set that re-defines the event set Y by adding a new sub-set to it. This sub-set is represented by a leaf non-terminal hyperedge of type ES .

The produced re-definition of the *ADAPTERS* process with a sub-process, represented by a leaf non-terminal hyperedge of type PE , and a value set non-terminal hyperedge of type VS cover the type and attribute independent part of requirement 3. The rest of the produced constructs cover an adapter process and an event set produced in each of the type specific requirements 4 – 12. The correspondences that are created in this rule, apart from the one that re-creates the correspondence between the model hyperedges of type M , all link the leaf usage weaving non-terminal hyperedge of type U to the previously described non-terminal leaf hyperedges of types P , PE , VS , and ES . They represent the connections of a usage weaving hyperedge of type U to all the created CSP structures and allow us to define new GBMT rule sets with these correspondences as pre-conditions to create the concrete instances of them in the type specific rules. Next, we consider two of these type specific rule sets – for a resource handling and a communication usage weaving.

The first type specific rule set covers requirements 3 and 4 for a resource handling weaving with an initial node as a consumer. It consists of four rules whose source derivation sub-trees are identical. They produce a resource handling usage instance by rewriting a hyperedge of type U and replace the produced hyperedge of type AN by an instance of an initial node (*InitialNode* class). The pre-condition correspondences of these four rules in the set have the types of the correspondences created by rule 4 in Figure 7.6.

The first rule set is shown in Figure 7.7 using a compact notation. We only show the source derivation sub-tree of the rules once, since it is the same in all of them, and link different pre-condition correspondences annotated by rule numbers to its root. The other ends of these correspondences are linked to the different target derivation sub-trees describing diverse effects of the rules. We describe each of these effects next.

Rule 5 in Figure 7.7 covers a part of requirement 4 and produces a set of four event instances with the required names and parameters by rewriting the corresponding (U -to- ES correspondence of type C) non-terminal hyperedge of type ES . Rule 6 covers a part of requirement 3 and produces a process instance with the required name from the corresponding (U -to- PE correspondence of type C) non-terminal hyperedge of type PE . Rule 7 covers the remaining part of requirement 4 and produces an adapter process with the

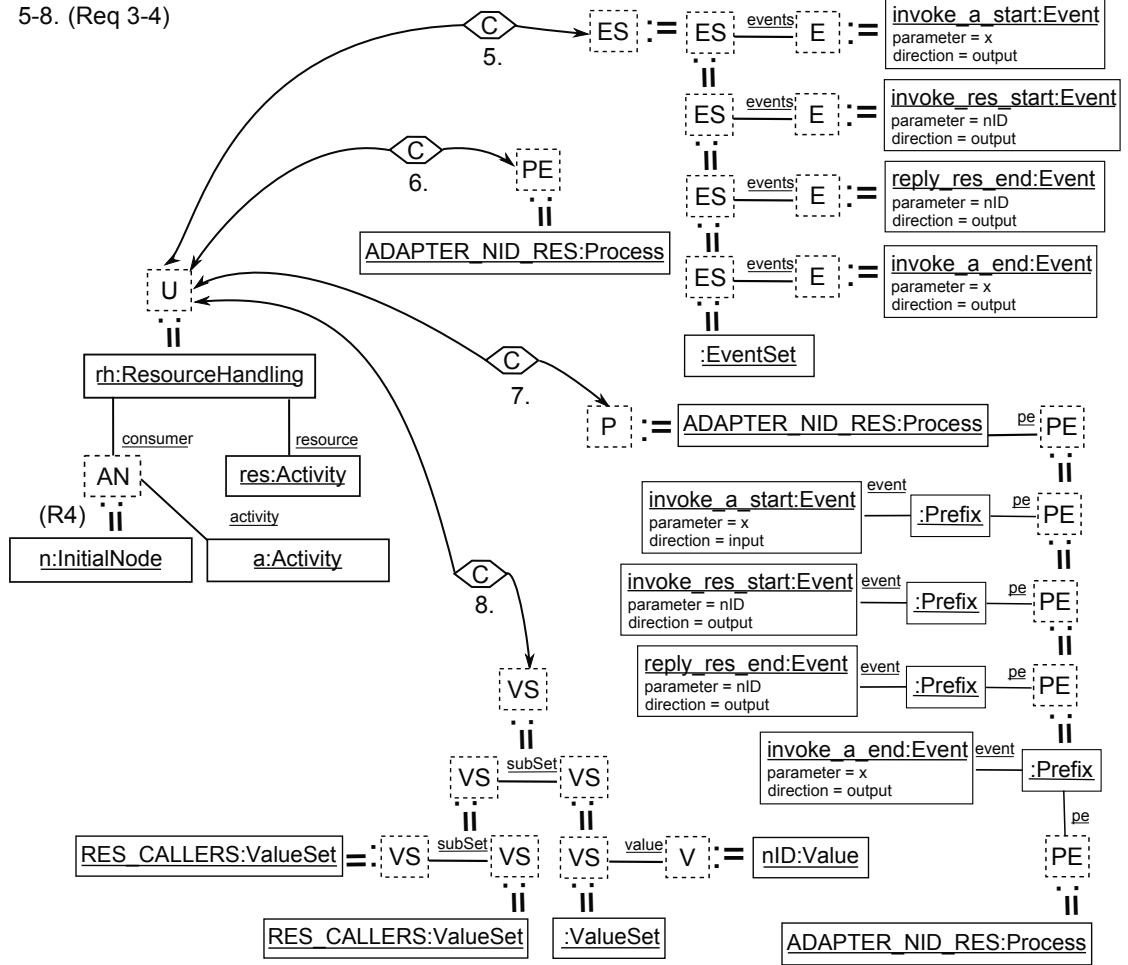
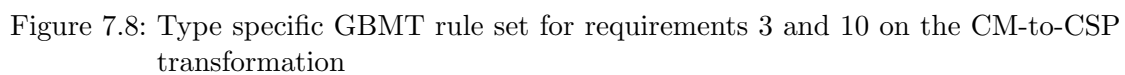


Figure 7.7: Type specific GBMT rule set for requirements 3 and 4 on the CM-to-CSP transformation

required name, structure, and event prefixes from the corresponding (*U-to-P* correspondence of type *C*) non-terminal hyperedge of type *P*. Finally, rule 8 covers the remaining part of requirement 3 and produces a value set that re-defines the *RES_CALLERS* value set by adding the *nID* value to it by rewriting the corresponding (*U-to-VS* correspondence of type *C*) non-terminal hyperedge of type *VS*.

Combined together, rule 4 (Figure 7.6) and rules 5 – 8 (Figure 7.7) cover requirements 3 and 4 for a resource handling usage weaving with an initial node (*InitialNode* class instance) as a consumer. To cover the requirements for all other types of consumers for this usage type (requirements 5 – 9) we also define analogous rule sets. Due to their similarity to the rule set shown in Figure 7.7, we do not give their description here.



Next, we consider the transformation of usage weavings of the communication type. We consider a rule set that covers requirement 10 from Table 6.3 for communication usage

weavings with a call action as a consumer. Like in the case with the resource handling usages, these rules are combined with the common rule 4 from Figure 7.6 to fully cover requirements 3 and 10 together. Figure 7.8 shows this rule set in the same compact notation we used in Figure 7.7. We describe each of these rules separately.

Rule 9 in Figure 7.8 covers a part of requirement 10 and produces a set of six event instances with the required names and parameters by rewriting the corresponding (*U-to-ES* correspondence of type *C*) non-terminal hyperedge of type *ES*. Rule 10 covers a part of requirement 3 and produces a value set that re-defines the *RES_CALLERS* value set by adding the *nID* value to it by rewriting the corresponding (*U-to-VS* correspondence of type *C*) non-terminal hyperedge of type *VS*. Rule 11 covers the remaining part of requirement 10 and produces an adapter process with the required name, structure, and event prefixes from the corresponding (*U-to-P* correspondence of type *C*) non-terminal hyperedge of type *P*. Finally, rule 12 covers the remaining part of requirement 3 and produces a process instance with the required name from the corresponding (*U-to-PE* correspondence of type *C*) non-terminal hyperedge of type *PE*.

Now that we have discussed the rules of our CM-to-CSP GBMT definition, we show that it fulfills the conditions stated in Section 5.3.1, in order to have the desired quality properties – terminate, be sound and complete, and deterministic. Since the defined GBMT does not have any non-local rules, we only need to consider the conditions stated in Definition 5.3.1.9. It requires each rule of our CM-to-CSP GBMT to either have the basic or the extended form, defined in Definition 5.3.1.1 and 5.3.1.5, respectively. Since Definition 5.3.1.1 is a special case of Definition 5.3.1.5, we only need to show that the latter is fulfilled by each rule. Also, Definition 5.3.1.9 puts two additional conditions on the GBMT.

First, we look at the individual rules of our CM-to-CSP GBMT to show that they fulfill the conditions. Each of these rules relates finite source and target derivation sub-trees (in some cases with only one production, rules 1 – 4) and has a correspondence between their roots. The correspondence functions defined in these rules map each leaf non-terminal hyperedge of the target derivation sub-trees to a correspondence type and a leaf non-terminal hyperedge of the related source derivation sub-trees. Furthermore, in a rule, none of these target leaf hyperedges are mapped onto more than one correspondence type and source leaf hyperedge pair. Thus, these correspondence mappings are functions and are total.

As for condition 1 inherited by Definition 5.3.1.5 through the extension of Definition 5.3.1.1, it holds since the productions with the start non-terminals at the left-hand side ($S := M$) are related only to each other in rule 1, and we only have one correspondence type *C* (is also $t_c^{default}$). Thus, all rules of our CM-to-CSP GBMT fulfill Definition 5.3.1.5.

Now, we need to show that the CM-to-CSP GBMT fulfills both conditions stated in Definition 5.3.1.9. Thereby, we consider all of its rules to have the same priority. All of the rules of the GBMT have distinct source derivation sub-trees and there is no

rule with a one-production source sub-tree with a production for a resource handling or a communication usage (productions 5, 6 in the configuration HR grammar, see Figure 7.4). Thus, none of the rules of our CM-to-CSP GBMT have a source sub-tree that is a prefix of the source sub-tree of the other one. Thus, condition 1 of Definition 5.3.1.9 holds for it.

The second condition (2) of Definition 5.3.1.9 requires the CM-to-CSP GBMT to have exactly one well-formed rule set $(RS_{p_s, c_{pre}})$ for each combination of a source production (p_s) from the source HR grammar and a type of correspondence (c_{pre}) a hyperedge of the type of its left-hand side (A_s) might have. By construction, our CM-to-CSP GBMT defines exactly one type of correspondence $(c_{pre}: M\text{-to-}M \text{ with correspondence type } C)$ for the source non-terminal M and four types of correspondences $(c_{pre}: U\text{-to-}ES, U\text{-to-}VS, U\text{-to-}P, \text{ and } U\text{-to-}PE \text{ with type } C)$ for the source non-terminal U .

For each combination of a source HR grammar production with the non-terminal M as its left-hand side (productions 2 – 4) and its only correspondence type, the CM-to-CSP GBMT contains exactly one rule set with one rule – a set with rule 2, rule 3, or rule 4. Each such rule set is well-formed, since it contains a special rule with one-production source derivation sub-trees (Definition 5.3.1.8, 1). The same can be shown analogously for the rule set for production 1 and the only correspondence type for its left-hand side $(c_{pre}: S\text{-to-}S \text{ of type } C)$, that contains exactly one rule – rule 1.

For each combination of a source HR grammar production with non-terminal U as its left-hand side (productions 5, 6) and its four correspondence types, our CM-to-CSP GBMT contains exactly one rule set (8 in total). These rule sets can be divided into the following two groups: those whose rules' source sub-trees start with production 5 (group 1) and those whose source derivation sub-trees start with production 6 (group 2).

First, we consider one rule set from group 1 with the correspondence pre-condition type $U\text{-to-}ES$ with type C . It contains rules for different types of consumer actions in a resource handling weaving. By construction of our CM-to-CSP GBMT, each of the consumer types is produced in the source derivation sub-tree of exactly one rule from this set. Rule 5 from Figure 7.7 is in this set. Its source derivation sub-tree describes the production of an initial node as the consumer (production 7). In order for the considered rule set to be well-formed, it has to contain all replacement alternatives of the source sub-tree of rule 5 and only them (see Definition 5.3.1.8, 2).

The source derivation sub-tree of rule 5 has five replacement alternatives in our configuration HR grammar. In each of them, the hyperedge of type AN , created by production 5, is rewritten by one of the productions 8 – 12. By construction of our CM-to-CSP GBMT, there is exactly one rule for each of these replacement alternatives, as its source sub-tree, in the considered rule set for production 5 and the correspondence pre-condition type $U\text{-to-}ES$ (with type C). Furthermore, there are no other rules in this set and, thus, it is well-formed. This can be shown analogously for the other rule sets in group 1.

Next, we consider one rule set from group 2 with the same correspondence pre-condition type $U\text{-to-}ES$ with type C . It contains rules for different types of consumer

actions in a communication weaving. By construction of our CM-to-CSP GBMT, each of the consumer types is produced in the source derivation sub-tree of exactly one rule from this set. Rule 9 from Figure 7.8 is in this set. Its source derivation sub-tree describes the production of a call as the consumer (production 13). In order for the considered rule set to be well-formed, it has to contain all replacement alternatives of the source sub-tree of rule 9 and only them (see Definition 5.3.1.8, 2).

The source derivation sub-tree of rule 9 has two replacement alternatives in our configuration HR grammar. In each of them, the hyperedge of type CN created by production 6, is rewritten by production 14 or 15. By construction of our CM-to-CSP GBMT, there is exactly one rule for each of these replacement alternatives, as its source sub-tree, in the considered rule set for production 6 and the correspondence pre-condition type $U\text{-to-}ES$ (with type C). Furthermore, there are no other rules in this set and, thus, this rule set is well-formed. This can be shown analogously for the other three rule sets in group 2.

By combining the above conclusions, we have shown that our CM-to-CSP GBMT defines exactly one well-formed rule set for each source production p_s and correspondence type its left-hand side can have in this GBMT. Thus, it is conform with Definition 5.3.1.9. Note that, the productions with the left-hand side non-terminals AN and CN do not have any related rules, since our CM-to-CSP GBMT does not create any correspondences between the hyperedges of these types.

We have shown that the CM-to-CSP transformation T_{Conf} for configuration models, described in Section 6.1.2, can be defined using our GBMT development approach, and that this definition is conform with the formal requirements on a GBMT (see Section 5.3.1). This allows us to guarantee the desired quality properties, discussed in Section 4.2, for our CM-to-CSP GBMT. Now, we can (re-)use this GBMT together with the ADR-to-CSP and the event splitting GBMTs (see Sections 5.2.1 and 7.1) to facilitate our weaving based approach to transformation re-use (see Chapter 6) in our case study (see Chapter 3).

To complete the picture, we mention the last transformation T_{Comp} – the composition of the CSP results produced by the above model transformations. This operation on models has two or more input models, which is currently not supported by our GBMT approach, that is focused on translation of single input models. Thus, at present, we define this composition directly by using ATL, which supports multiple input models. In Section 8.4, we discuss how our GBMT development approach can be extended to support multiple input/output models.

7.3 Discussion

There are several points, that we came across, while defining the example event splitting in CSP and CM-to-CSP transformations, from Section 6.1.2, using our grammar based approach to the development of model transformations from Chapter 5.

The definition of the event splitting transformation, in Section 7.1, required just four relatively simple changing GBMT rules and some trivial non-changing ones (not discussed here). Only two of the changing GBMT rules have source sub-trees with more than one production. Due to the structure of the CSP HR grammar that we use (see Section 3.2), these sub-trees do not have replacement alternatives (see Definition 5.3.1.6). This fact allowed us to keep the number of required GBMT rules low, yet, it indicated the general non-linear influence of the structure of the used source HR grammar on the number of required GBMT rules, when one or more of them are of the extended type (see Definition 5.3.1.5).

On the target side of the GBMT rules of the event splitting transformation, we did not require any non-local effect and, thus, we only used main sub-trees. Each of these target sub-trees has at most four productions. As with respect to the correspondences, the definition of correspondence type pre-conditions (see Definition 5.3.1.1) for the selected rules' sub-trees was evident and, thus, straightforward. The definition of the correspondences between the leaf non-terminal hyperedges of the related sub-trees was also quite straightforward, due to the nature of these sub-trees. Here, we only needed to link the hyperedges of the same type by one-to-one or one-to-two (rule 4, Figure 7.1) correspondences. This, however, might not always be the case for other model transformations. The same is true for the number of used correspondence types, like the one (C) used in this GBMT.

For the provided GBMT for our event splitting transformation, we also showed the satisfaction of the conditions, required for the quality properties (see Section 4.2.1). Although, it was possible to show this manually by going through the defined GBMT rules and taking into account the source grammar structure, as done here for demonstration purposes, in general this process needs to be automated. Such automation is essential in order to allow the transformation developer to define quality model transformations without having to perform such complex manual checks.

The definition of CM-to-CSP, in Section 7.2, required more complex steps. First, we decided to model the knowledge about the different types of woven model elements and the related constraints in an explicit form in the extended meta-model for the system configuration language, shown in Figure 7.3. Based on this extended meta-model, we defined a system configuration HR grammar (see Figure 7.4) with respective productions. Note that, the given HR grammar represents one of the multiple possible versions. Another prominent version of such a system configuration HR grammar could produce a summary communication node (hyperedge of type CN) from AN non-terminal, instead of the direct production of the concrete communication nodes from it (productions 10 – 12, Figure 7.4). We discuss why we chose the former version of this grammar later.

Compared to the event splitting, for the CM-to-CSP transformation we defined a larger number of (often) more complex GBMT rules. This is directly related with the complexity of this example compared to the previous one. On the target side of the rules, we defined more complex derivation sub-trees with more leaf non-terminal hyperedges and, hence, also more correspondences to the respective leafs of the source sub-trees. On

the source side of the rules, we defined both simple (one-production) and more complex sub-trees, where the former type did not provide the sufficient information. The latter build sets of type specific GBMT rules with the same source sub-tree and different correspondence type pre-conditions and target sub-trees.

We have provided examples of such GBMT rule sets for a resource handling rh with an initial node n as the consumer, and a communication usage c with a call action n as the consumer. These example sets are shown in Figures 7.7 and 7.8, respectively. Altogether, there are such sets for all replacement alternatives of the source sub-trees of these example rules. They cover all the possibilities to replace an activity node non-terminal hyperedge (of type AN) as the consumer of a resource handling usage, or a communication node non-terminal hyperedge (of type CN) as the consumer of a communication usage.

For the CM-to-CSP transformation example we also did not require any non-local effects and, thus, the resulting GBMT does not contain any non-local rules. This fact simplifies the analysis of the defined rules in order to conform that they satisfy the sufficient conditions, from Section 5.3, to guarantee the transformation quality properties. With this, the most complex conditions that we need to check are those of extended GBMTs (see Definition 5.3.1.9). One of these conditions, more precisely, the requirement of well-formedness of rule sets for each source production and correspondence type pre-condition combination, influences our choice of the form of the (source) configuration HR grammar. The limitations given by the definition of replacement alternatives (see Definition 5.3.1.6) allows us to define a quality GBMT, that satisfies all conditions, based on the configuration HR grammar from Figure 7.4, but not based on an alternative version that we have discussed above.

This fact indicates the dependency of our approach on the form of the used source HR grammar, in order to guarantee the transformation quality properties from Section 4.2.1. Similar dependency relations, with respect to complexity and maintainability, can be found between the meta-models and the model transformation development approaches based on them. In particular, some conditions on GBMTs imply a certain distribution of generated model elements between the source HR grammar productions (certain depth). It is, however, not always possible to change the given source grammar accordingly. In general, it is possible to weaken this and other sufficient conditions on GBMTs to allow the use of further forms of source HR grammars. We discuss some ideas in this direction in Section 8.4.

Overall, the application of our grammar based approach to the development of model transformations to define the considered event splitting and CM-to-CSP transformations was successful and provided us with useful insights about the scope and limitations of our method. We were able to define both model transformations with it, with a moderate effort and time investment for the provided quality guarantees, that it provides for the correctly-defined resulting GBMTs. Furthermore, the structure-oriented nature of the GBMT rules relating productions, and their declarative nature with an implicit results composition and a possibility to use concrete syntax (if given), encouraged us to think

in terms of the required element and structure based transformation mappings. This fact allowed a more straightforward conversion of the transformation requirements from Tables 6.2 and 6.3 into the GBMT rules.

Finally, since we have compared our grammar based approach with other model transformation development approaches on a general level in Section 5.7, we refrain from discussing this for the considered specific examples here.

Chapter 8

Conclusion

In the beginning of this thesis, in Chapter 2, we discussed MDSD and its main concepts, like models, meta-models, and model transformations. We emphasized the key role of model transformations in MDSD and the importance of their systematic development, quality assurance, and re-use. To this point, in Chapter 4, we discussed the state-of-the-art approaches for model transformations with respect to all of these aspects. To make this discussion more demonstrative, we used the model transformation case study for multi-layer systems introduced in Chapter 3.

During the discussion of this active research field in Chapter 4, we identified several gaps in both the model transformation development (Section 4.1), the quality assurance (Section 4.2), and the re-use (Section 4.3) areas. We pointed them out in the context of our transformation case study for multi-layer systems.

In an attempt to close or, at least, reduce these gaps, we proposed the following two approaches: the grammar based approach to the development of model transformations from Chapter 5 and the weaving based approach to the re-use of model transformations from Chapter 6. We also considered whether and how the above two approaches can be combined in Chapter 7. Next, we summarize these contributions in more detail.

8.1 Thesis Contribution

The key contributions of this thesis constitute the following previously named approaches: the grammar based approach to the development (Chapter 5) and the weaving based approach to the re-use (Chapter 6) of model transformations. They address various issues that have been identified in Chapter 4.

The grammar based approach to model transformation development addresses the issues of both development and quality assurance, with the focus on the transformations between modeling languages with high-level structures, like decision/merge blocks in activities (see Section 3.1). The goals of this approach are the following: to simplify the development/implementation of model transformations and, especially of those requiring structure-based mappings, and, if possible, keep them declarative; to allow a seamless connection of the results of the mappings to the rest of the generated target model;

and to guarantee the quality of the resulting model transformations by construction (see Section 4.2).

The weaving based approach to the re-use of model transformations addresses the issue of the re-use of the already existing grammar based and other model transformations and of their results, when they are used to transform the parts of some system modeled by several models. This is particularly interesting when the system to be transformed can be re-configured – new sub-models added, re-connected, or removed. In our example from Section 3.1, such a system is the trip planning multi-layer system in which the platform/infrastructure layers models can be re-configured.

Overall, the goals of the weaving based approach are the following: to allow a non-invasive modeling of the interactions of the individual system models without having to change them; to provide a concept for the modular transformation execution and for the composition mechanism for their results in the target language; and to define general requirements that need to be fulfilled in order to use the method. Since the composition mechanism, that needs to be proposed, is target language specific, we can not provide a universal solution here. Instead, we demonstrate the CSP composition mechanism, that we propose in Section 6.1.2, as an example.

Next, we analyze the proposed approaches in detail with respect to the design decisions made, the resulting application scope and restrictions, and their strengths and weaknesses. For clarity we separate this analysis into Sections 8.2 and 8.3, respectively.

8.2 Analysis of Grammar Based Development Approach

As any other method, our grammar based approach is based on some key design decisions, that influence and restrict its application scope and form its strengths and weaknesses. Thus, we consider them first.

Major Design Decisions One of the main design decisions when designing an approach to the development of model transformations, is the choice of the method to define the source and target modeling languages. The other main design decision here is the choice of the method to define the model transformations themselves.

For the former decision, we have chosen HR graph grammars (see Section 5.1) conform with the respective source/target meta-models, instead of simply using meta-models (potentially with structuring classes) (see Section 2.2). This way, we were able to benefit from the structure-oriented nature of the grammar based language definition and could explicitly model the high-level language structures, as parts of the productions (e.g., Figure 5.3 in Section 5.1), without having to change the given meta-models.

For the latter decision, we have chosen to model transformations by relating the productions from the source and the target HR grammars, in a way inspired by pair

grammars [Pra71] and TGGs [Sch94] with their correspondences. We extended the rule form used in TGG rules, that in their idea also relate single productions, to the case of n-to-m productions relations. We also did not use any explicit correspondence sub-productions in our GBMT rules. This way, we were able to express structure-based transformation mappings as declarative TGG-like rules relating the productions for the respective language structures.

The above decisions subsequently led to the use of the derivation trees of the transformed input source models in order to execute the grammar based model transformations defined. Additionally, they allowed us to define the conditions on the developed model transformation that guarantee their termination, soundness and completeness, and determinism (see Section 5.3). The form of conditions has been chosen here in order to separate the basic idea of the structure of the grammar based model transformations from the additional conditions, required in order to guarantee their quality. This way, one can also define and use grammar based model transformations, even if they do not meet all of the quality conditions.

Application Scope and Restrictions The type of graph grammars (HR grammars) that we use to express the involved modeling languages, explicitly restricts the applicability scope of our approach to the languages, that can be described using such context-free HR grammars. To overcome this restriction by context-sensitive modeling languages, in simpler cases, one can employ the duplication of model elements or references to them, as done in Sections 7.2.1. In more complex cases, when the first solution is not sufficient, an extension of our approach to context-sensitive graph grammars is possible, as outlined in Section 8.4.

As for the chosen TGG-like form for the grammar based transformation rules, no direct restrictions are added here, since TGGs are Turing complete [BDK⁺12]. However, the additional sufficient conditions in order to guarantee the quality of the developed model transformations put restrictions on the form of the resulting grammar based rules and the complete transformations. In turn, this restricts the way a given transformation logic can be expressed using them. Some of these restrictions can be overcome through changes to the transformation rules and, if allowed, to the source/target HR grammars. In other cases, the resulting grammar based model transformations can still be used, but their quality is no longer guaranteed.

Strengths and Weaknesses Due to the design decisions made, our approach benefits from the structure-oriented nature of the grammar based language definition and explicit modeling of high-level language structures, as parts of HR grammar productions. Through the direct use of such productions for the transformation definition, the developer can directly express the mappings of both the simple language constructs as well as of the high-level language structures in declarative rules, and is not forced

to use imperative constructs and recursion to find them, as in the implementations in Section 4.1.3.

The HR grammar based language descriptions, used in our approach, in general, are more restrictive and complex than pure meta-models (with structuring classes). Yet, compared to meta-models with the structural OCL constraints of comparable expressiveness (when possible), HR grammars typed over meta-models present a more concise and intuitive (possibly with concrete syntax) way to describe such constraints. Like meta-models, they also only need to be created once per language and can be re-used. The quality and complexity of an HR grammar can affect the characteristics of the grammar based transformations that use it. However, the same is true for meta-models and model transformations based on them.

Another benefit, that comes with the use of grammar productions in order to define model transformations, is a straightforward solution of the issue of connecting the transformation results. We can simply use the connection/gluing mechanism of the HR productions of the target language, which is implicitly present in each grammar based transformation rule. Thus, there is no need for an extra, potentially complex, connection logic with our approach.

The declarative TGGs-like form of grammar based transformation rules also has several advantages, like better readability, absence of complex imperative constructs, and, thus, more effective quality assurance techniques (see Section 4.2). Furthermore, the conditions we put on our grammar based model transformations allow us to guarantee their termination, soundness and completeness, and determinism, by construction. None of the existing state-of-the-art approaches for the quality assurance of model transformations (see Section 4.2) can guarantee all of these four properties in the same settings.

On the other hand, the same quality conditions restrict the scope of the model transformations that we can express using the grammar based approach, while guaranteeing the quality properties. The efforts to fulfill them can lead to more complex grammar based rules and complete transformations, and might not always be successful.

Another source of complexity in our grammar based approach is model parsing, required to obtain the source derivation trees. However, as discussed in Section 5.1.2, the parsing based on HR grammars is decidable and can be done efficiently in many cases [Lau90, CAB⁺13]. Furthermore, our approach benefits from splitting this parsing step from the model transformation execution, since it then only needs to be performed once for each source model and grammar combination.

One further question, that we did not consider explicitly in our grammar based approach, is the computation of attribute values. At this point, we only use simple assignments with direct or reference-based access to the values and discuss further possibilities in Section 8.4. Overall, however, we consider our grammar based approach to make a further step towards an efficient and quality-aware development of model transformations, that can be realized using the existing state-of-the-art model transformation implementation languages and engines.

8.3 Analysis of Weaving Based Re-Use Approach

Here, we consider the key design decisions made during the development of our weaving based approach to the re-use of model transformations. These decisions influenced and restricted its application scope as well as its strengths and weaknesses, also considered here.

Major Design Decisions We have already thoroughly discussed our choice to keep the system models and model transformations separated in the considered scenario with multiple models and re-used model transformations in Section 6. Thus, we just point out this design decision here without any additional motivation. For further design decisions, like the choice of a method for the modeling of models inter-relations and for the composition mechanism, we provide such motivation.

The first design choice made for this approach was to use the model weaving technique (see Section 6.1.1) as a basis to model the inter-relations between the system models. For this purpose, we extended the core weaving meta-model to create our system configuration meta-model, to accommodate the relevant information in the resulting configuration models. This decision has been motivated, in the first place, by the non-invasive and model language-independent nature of the model weaving technique, that allows us to put the interaction information into a separate configuration model. Furthermore, the extensibility of this technique allowed its convenient customization for our purpose.

The second design decision is concerned with the choice of the composition mechanism. Since such a mechanism for the target languages is language-specific, the decision which composition concepts/operators to use in the particular target language needs to be made individually. This decision influences the requirements on the transformation of the configuration models into the considered target language, and, possibly, triggers additional post-processing of the transformation results for the individual models.

In Section 6.1.2, we have shown an example of such a composition mechanism in CSP, based on the process synchronization via events. This mechanism puts certain requirements on the model transformation for configuration models (T_{Conf}), with respect to its result, and requires some post-processing of the transformation results for the individual layers models (T_{X2CSP} part of T_{LModel}) via the event splitting transformation ($T_{EventSplit}$ part of T_{LModel}). We have also shown the following concrete examples of transformation mappings, that fulfill the requirements of the chosen composition mechanism: ADR-to-CSP transformation in Table 3.1, CM-to-CSP transformation in Table 6.3, and the event splitting in Table 6.2. In other target modeling languages, such a composition mechanism can look both similar (e.g., in other process algebras, like FSP [FEK⁺07]), as well as completely different (e.g., in Petri nets).

Application Scope and Restrictions Due to the decision to use model weaving to model the inter-relations between the individual system models, our configuration models can now be defined between any number of such models in arbitrary languages. For the same reason, it is also relatively easy to extend the configuration language that we have defined in Section 6.1.1, to accommodate the new or change the current concepts of this language. Thus, there are no direct approach-induced restrictions on the types (languages) and number of system models that can be used as long as they contain the interacting elements, that can be linked via a configuration model.

A similar statement is true for the existing re-usable model transformations for the individual system models. Their concrete implementation does not play any role, since they only need to provide the expected results (depends on a concrete example), that are then, possibly, post-processed (e.g., event splitting in CSP) and composed. This leads us to the main restriction on the target language used with our approach – the existence of suitable composition operators (like parallel composition with synchronization in CSP, Section 3.2) in it. Plus, of course it has to be able to represent the concepts of the source and configuration models in an appropriate way (depends on a concrete example).

Strengths and Weaknesses As we have already mentioned, the model weaving technique allows us to combine different system models into a configuration in a non-invasive and flexible way. This flexible composition then lets us re-use the existing model transformations for the individual system models and for the configuration model, without having to change them or enforce any particular requirements on them. Furthermore, when only some of the models need to be changed in a re-configuration, our approach allows to re-use the transformation results for the unchanged models. The same is true for the involved model transformations that do not need to be exchanged.

The same flexible, non-invasive, weaving based configuration modeling approach can, however, bring up certain issues. With the growing complexity of the involved models and of the interactions between their elements, the manual change of the configuration meta-model and models used can become impractical. However, since the individual system models do not contain the information about their outside interactions, we can not directly generate the configuration models through some dependency analysis. One possible solution here would be to define a set of pre-defined interaction scenarios or rules to lead the generation of configuration models. We consider possibilities to address this issue in Section 8.4.

Another essential and potentially complex task of our weaving based approach is the definition of model transformations for the configuration models. This task is performed once for each configuration language, target language, and a chosen composition mechanism for the latter. Thereby, significant expertise in both of the languages and, especially, in the selection of the composition mechanisms, is required. We consider possibilities to simplify this task in Section 8.4.

Overall, we consider our weaving based approach to make a further step towards an efficient and flexible re-use of model transformations and of their results. It can also be realized with the existing state-of-the-art model weaving (see Section 6.1.1) and model transformation chaining techniques (see Section 4.3.1).

8.4 Future Research Directions

Now, we summarize and elaborate on the possible future extensions of both approaches that we have proposed to improve the systematic development of model transformations. We also discuss further possible applications of these approaches.

The Grammar Based Approach to the Development of Transformations The first extension of this approach, that we think should be researched next, is the integration of attribute computation. It has to allow the computation of more complex attributes, while still keeping the desired model transformation properties. This would require finding both an appropriate notation to describe the attribute computations, as well as finding the conditions on such computations, that will allow to keep the quality guarantees. One possible direction that we see here, is the expression of attribute values as hyperedges connected to the hyperedges representing the objects, similar to [EPT04]. This would allow the use of grammar based rules, in their present or (possibly) extended form, to describe the computation of such attribute hyperedges.

Another related and more complex extension of our grammar based approach is to allow the support for context-sensitive modeling languages, that can not be described using context-free HR grammars. The first step in this direction, that we suggest, is to explore the decidable *contextual HR grammars* proposed by Drewes in [DHM12]. As with the previous extension, potential changes to the form of the transformation rules as well as to the sufficient quality conditions for the model transformation need to be considered in detail.

A further possible extension, that we came across while describing the flattening transformation example for the hierarchical activity diagrams in Section 5.7, is concerned with the development of abstracting/deleting model transformations. There, we provided one of the non-trivial grammar based rules for the flattening example. The second non-trivial rule for the source production of a single nested activity, however, can not be defined in such a concise way at present. This is due to the fact that there are no target production(s) that we can relate to it. The effect that we want to achieve in the target models – the deletion of nesting nodes – can not be expressed using the creating target productions.

To solve this issue we propose to allow *idle target productions*, that only re-create the hyperedge that they replace. With such productions we can then describe the deletion of nesting nodes, as shown in Figure 8.1. After the transformation execution such idle

productions can be replaced by a single hyperedge in the target derivation tree. As with the other extensions, the use of such productions requires the analysis of possible changes to the form of grammar based transformation rules as well as to the form of the sufficient quality conditions.

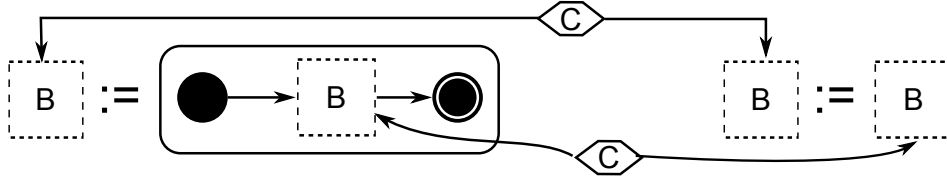


Figure 8.1: Flattening hierarchical activities using an idle target production

In general, further research directions can include attempts to weaken the sufficient quality conditions on grammar based model transformations. Here, we see the conditions on the replacement alternatives as a promising first candidate. Another interesting research direction is concerned with the extension of the proposed approach to multiple potentially heterogeneous input/output models. Here, we expect lower complexity when considering multiple target models than in the extension to multiple source models. In the former case it might be possible to consider such a 1-to-n transformation as a set of 1-to-1 transformations with appropriate conditions on the rules. In the latter case one would need to consider complex inter-dependencies between the productions in the source grammars and in their transformation-specific combination.

As for the further application areas for our grammar based approach, we can imagine its successful use to resolve structural heterogeneity between the languages, when semantically similar modeling concepts are defined by different meta-modeling concepts, as described in [KKR⁺08a]. Our approach can simplify the specification of the model transformations needed to resolve this type of heterogeneity.

Finally, we want to point out, that a simpler version of the idea, that our method is based on, has been successfully used in the field of computational linguistics. In [JAB⁺12], the authors use HR grammars for the semantics-based statistical machine translation between the natural languages. It would be interesting to see if the more complex version of our grammar based approach has a potential to improve their method, and if the graph-shaped intermediate semantics representations, used in [JAB⁺12], can improve our method.

The Weaving Based Approach to the Re-Use of Transformations As we have pointed out in Section 8.3, there are various possible extensions to our weaving based approach that can improve it and simplify its application. The first improvement is useful in order to address the complexity of the configuration models, that comes with the growing number of interactions between the woven models. One possible solution that we see here is to provide a set of pre-defined usage scenarios, similar to the technique used in [KB10], to allow a (semi-)automatic generation of configuration models and,

possibly, also of configuration meta-models based on them. Another possible solution would be to define a form of interaction rules, showcased in [FEK⁺07] on a Web Server example, that can be expressed in languages like the business rules of [SBV].

Another complex task in our weaving based approach, that can be improved, is the development of a model transformation for the given configuration language, target language, and the chosen composition mechanism in the latter. This task requires significant expertise. Thus, an ability to re-use the already existing transformations of such a kind can simplify it. In the cases, when one configuration language is an extension of the other, with an existing model transformation into the same target language and with the same composition mechanism, some degree of re-use can be achieved by using various methods, like [GDA13], for the meta-model and model transformation co-evolution, or methods for the chaining of model transformations (see Section 4.3.1).

As for further application areas for our weaving based approach, we see its use for further target languages for, e.g., performance and reliability analysis. In general, it can be applied in various scenarios where a (changing/re-configurable) set of inter-related models has to be transformed in parallel, if possible, and these results then need to be combined. We see examples of such scenarios in areas like software-product lines [WL99], self-adapting systems [CDLG⁺09], and aspect-oriented modeling [GV09].

Bibliography

- [AEM12] Vincent Aranega, Anne Etien, and Sebastien Mosser. Using Feature Model to Build Model Transformation Chains. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 562–578. Springer Berlin Heidelberg, 2012.
- [AEN⁺08] Adil Anwar, Sophie Ebersold, Mahmoud Nassar, Bernard Coulette, and Abdelaziz Kriouile. Towards a Generic Approach for Model Composition. In *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA)*, pages 83–90. IEEE Computer Society Washington, DC, USA, 2008.
- [AGT12] Colin Atkinson, Ralph Gerbig, and Christian Tunjic. Towards Multi-level Aware Model Transformations. In Zhenjiang Hu and Juan de Lara, editors, *International Conference on Model Transformation (ICMT)*, volume 7307 of *Lecture Notes in Computer Science*, pages 208–223. Springer Berlin Heidelberg, 2012.
- [AL13] Anthony Anjorin and Marius Lauder. A Solution to the Flowgraphs Case Study using Triple Graph Grammars and eMoflon. In Pieter Van Gorp, Louis M. Rose, and Christian Krause, editors, *Proceedings of the Sixth Transformation Tool Contest (TTC)*, volume 135 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 69–74. Open Publishing Association, 2013.
- [ALS⁺12] Moussa Amrani, Levi Lucio, Gehan Selim, Benoit Combemale, Jurgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, (ICST), pages 921–928. IEEE Computer Society Washington, DC, USA, 2012.
- [AMW] AMW Model Handler. http://wiki.eclipse.org/AMW_Model_Handler, accessed: 15.02.15.
- [AN99] Peter R. J. Asveld and Anton Nijholt. The Inclusion Problem for Some Subclasses of Context-free Languages. *Theoretical Computer Science*, 230(1-2):247–256, 1999. Elsevier Science Publishers.

- [ARW13] Lukman Ab. Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software and Systems Modeling*, pages 1–26, 2013. Springer Berlin Heidelberg.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Boston, 1986.
- [ASWK11] Mark Asztalos, Eugene Syriani, Manuel Wimmer, and Marouane Kessentini. Simplifying Model Transformation Chains by Rule Composition. In Juergen Dingel and Arnor Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 293–307. Springer Berlin Heidelberg, 2011.
- [ATL] ATL Transformation Language Project. <http://www.eclipse.org/atl>, accessed: 13.10.14.
- [BBG05] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer Berlin Heidelberg, 2005.
- [BCC⁺10] Hugo Brunelière, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin. Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and Francois Terrier, editors, *Modelling Foundations and Applications*, volume 6138 of *Lecture Notes in Computer Science*, pages 32–47. Springer Berlin Heidelberg, 2010.
- [BCDM14] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032, 2014. Elsevier Science.
- [BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012.
- [BDK⁺12] Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier, and Jan Stückrath. On the Decidability Status of Reachability and Coverability in Graph Transformation Systems. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA)*, volume 15 of *LIPICs*, pages 101–116. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
- [BECG12] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL Transformations Using Transformation Models and Model Finders. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods (ICFEM)*, volume 7635 of *Lecture Notes in Computer Science*, pages 198–213. Springer Berlin Heidelberg, 2012.

- [Ben87] Herbert D. Benington. Production of Large Computer Programs. In *Proceedings of the 9th International Conference on Software Engineering (ICSE)*, pages 299–310. IEEE Computer Society Press Los Alamitos, CA, USA, 1987.
- [BH07] Dénes Bisztray and Reiko Heckel. Rule-Level Verification of Business Process Transformations using CSP. *Electronic Communication of the European Association of Software Science and Technology ECEASST*, 6, 2007.
- [BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer Berlin Heidelberg, 2005.
- [BKE11] Eduard Bauer, Jochen M. Küster, and Gregor Engels. Test Suite Quality for Model Transformation Chains. In Judith Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 3–19. Springer Berlin Heidelberg, 2011.
- [BKR09] Steffen Becker, Heiko Koziolk, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. Special Issue: Software Performance - Modeling and Analysis, Elsevier Science.
- [BL08] Matthias Bräuer and Henrik Lochmann. An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications, 5th European Semantic Web Conference (ESWC)*, volume 5021 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin Heidelberg, 2008.
- [BL09] Matthias Biehl and Welf Löwe. Automated Architecture Consistency Checking for Model Driven Software Development. In Raffaella Mirandola, Ian Gorton, and Christine Hofmeister, editors, *Architectures for Adaptive Software Systems*, volume 5581 of *Lecture Notes in Computer Science*, pages 36–51. Springer Berlin Heidelberg, 2009.
- [BLA⁺11] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. DSLTrans: A Turing Incomplete Transformation Language. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 296–305. Springer Berlin Heidelberg, 2011.
- [BN84] Andrew Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984. ACM New York, NY, USA.

- [BPOB13] Fábio Paulo Basso, Raquel Mainardi Pillat, Toacy Cavalcante Oliveira, and Leandro Buss Becker. Supporting Large Scale Model Transformation Reuse. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, (GPCE), pages 169–178. ACM New York, NY, USA, 2013.
- [BRS⁺00] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal System Development with KIV. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 363–366. Springer Berlin Heidelberg, 2000.
- [BSML07] Krishnakumar Balasubramanian, Douglas C. Schmidt, Zoltan Molnar, and Ákos Lédeczi. Component-Based System Integration via (Meta)Model Composition. In *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 93–102. IEEE Computer Society, 2007.
- [BSS12] Cristiano Bertolini, Martin Schäfer, and Volker Stolz. Towards a Formal Integrated Model of Collaborative Healthcare Workflows. In Zhiming Liu and Alan Wassyng, editors, *Foundations of Health Informatics Engineering and Systems*, volume 7151 of *Lecture Notes in Computer Science*, pages 57–74. Springer Berlin Heidelberg, 2012.
- [BSW14] Galina Besova, Dominik Steenken, and Heike Wehrheim. Grammar-Based Model Transformations. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 1601–1610. IEEE Computer Society, 2014.
- [BWB12] Maarten M. Bezemer, Robert J.W. Wilterdink, and Jan F. Broenink. Design and Use of CSP Meta-Model for Embedded Control Software Development. In Peter H. Welch, Frederick R.M. Barnes, Kevin Chalmers, Jan Baekgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures*, volume 69 of *Concurrent System Engineering Series*, pages 185–199. Open Channel Publishing Ltd. England, 2012.
- [BWW11] Galina Besova, Heike Wehrheim, and Annika Wagner. Reputation-based Reliability Prediction of Service Compositions. *Proceedings of the 8th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA)*, *Electronic Notes in Theoretical Computer Science*, 279(2):3–16, 2011. Elsevier Science.
- [BWWB12] Galina Besova, Sven Walther, Heike Wehrheim, and Steffen Becker. Weaving-Based Configuration and Modular Transformation of Multi-layer Systems. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin

- Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 776–792. Springer Berlin Heidelberg, 2012.
- [CAB⁺13] David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. Parsing Graphs with Hyperedge Replacement Grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1)*, pages 924–932. Association for Computational Linguistics, 2013.
- [CCGdL10] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and Validation of Declarative Model-to-model Transformations Through Invariants. *Journal of Systems and Software*, 83(2):283–302, 2010. Elsevier Science Inc. New York, NY, USA.
- [CDLG⁺09] Betty HC Cheng, Rogerio De Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. In *Software engineering for self-adaptive systems*, pages 1–26. Springer Berlin Heidelberg, 2009.
- [CGdL14] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Reverse Engineering of Model Transformations for Reusability. In Davide Di Ruscio and Dániel Varró, editors, *Proceedings of the 7th International Conference on Model Transformation (ICMT)*, volume 8568 of *Lecture Notes in Computer Science*, pages 186–201. Springer Berlin Heidelberg, 2014.
- [CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006. IBM Corp. Riverton, NJ, USA.
- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005. John Wiley & Sons, Ltd.
- [CM09] Jesús Sánchez Cuadrado and Jesús García Molina. Modularization of model transformations through a phasing mechanism. *Software and Systems Modeling*, 8(3):325–345, 2009. Springer Berlin Heidelberg.
- [CMR⁺97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

- [Cor06] James R. Cordy. The TXL source transformation language. *Science of Computer Programming, Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA)*, 61(3):190–210, 2006. Elsevier North-Holland, Inc. Amsterdam, The Netherlands.
- [CS13] Daniel Calegari and Nora Szasz. Verification of Model Transformations: A Survey of the State-of-the-Art. *Proceedings of the 38th Latin American Conference in Informatics (CLEI), Electronic Notes in Theoretical Computer Science*, 292(0):5–25, 2013. Elsevier Science.
- [DdF07] Marcos Didonet del Fabro. *Metadata management using model weaving and model transformation*. PhD thesis, Universite de Nantes, 2007.
- [DFBJ⁺05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwann Breton, and Guillaume Gueltas. AMW: A Generic Model Weaver. In *1 ere Journées sur l’Ingénierie Dirigée par les Modèles (IDM)*, pages 105–114, 2005.
- [DHM12] Frank Drewes, Berthold Hoffmann, and Mark Minas. Contextual Hyperedge Replacement. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 7233 of *Lecture Notes in Computer Science*, pages 182–197. Springer Berlin Heidelberg, 2012.
- [dLT04] Juan de Lara and Gabriele Taentzer. Automated Model Transformation and Its Validation Using ATOM3 and AGG. In Alan F. Blackwell, Kim Marriott, and Atsushi Shimojima, editors, *Proceedings of the Third International Conference on Diagrammatic Representation and Inference*, volume 2980 of *Lecture Notes in Computer Science*, pages 182–198. Springer Berlin Heidelberg, 2004.
- [DREP12] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Model Transformations. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 91–136. Springer Berlin Heidelberg, 2012.
- [EABP12] Anne Etien, Vincent Aranega, Xavier Blanc, and Richard F. Paige. Chaining Model Transformations. In *Proceedings of the First Workshop on the Analysis of Model Transformations (AMT)*, pages 9–14. ACM New York, NY, USA, 2012.
- [EE06] Hartmut Ehrig and Karsten Ehrig. Overview of Formal Concepts for Model Transformations Based on Typed Attributed Graph Transformation. *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT), Electronic Notes in Theoretical Computer Science*, 152(0):3–22, 2006. Elsevier Science.

- [EE08] Hartmut Ehrig and Claudia Ermel. Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 194–210. Springer Berlin Heidelberg, 2008.
- [EE^{dL}+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination Criteria for Model Transformation. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer Berlin Heidelberg, 2005.
- [EEH08] Hartmut Ehrig, Claudia Ermel, and Frank Hermann. On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars. In *Proceedings of the Third International Workshop on Graph and Model Transformations (GRaMoT)*, pages 9–16. ACM New York, NY, USA, 2008.
- [EEHP09] Hartmut Ehrig, Claudia Ermel, Frank Hermann, and Ulrike Prange. On-the-Fly Construction, Correctness and Completeness of Model Transformations Based on Triple Graph Grammars. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 241–255. Springer Berlin Heidelberg, 2009.
- [EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, EATCS Series. Springer Berlin Heidelberg, 2006.
- [EHS09] Hartmut Ehrig, Frank Hermann, and Christoph Sartorius. Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. *Electronic Communications of the ECEASST*, 18, 2009. European Association of Software Science and Technology (ECEASST).
- [EKHL03] Gregor Engels, Jochen M. Küster, Reiko Heckel, and Marc Lohmann. Model-Based Verification and Validation of Properties. *Uniform Approaches to Graphical Process Specification Techniques (Satellite Event for {ETAPS})*, *Electronic Notes in Theoretical Computer Science*, 82(7):133–150, 2003. Elsevier Science.
- [EKR⁺08] Gregor Engels, Anneke Kleppe, Arend Rensink, Maria Semenyak, Christian Soltenborn, and Heike Wehrheim. From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 94–109. Springer Berlin Heidelberg, 2008.

- [EMF] Eclipse Modeling Framework (EMF). <http://www.eclipse.org/modeling/emf>, accessed: 08.10.14.
- [EMM⁺12] Romina Eramo, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. A model-driven approach to automate the propagation of changes among Architecture Description Languages. *Software and Systems Modeling*, 11(1):29–53, 2012. Springer Berlin Heidelberg.
- [Eps] Epsilon Transformation Language Project. <http://eclipse.org/epsilon>, accessed: 19.01.15.
- [EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Proceedings of the Second International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer Berlin Heidelberg, 2004.
- [FBB⁺07] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 482–497. Springer Berlin Heidelberg, 2007.
- [FBV⁺09] Mathias Fritzsche, Hugo Bruneliere, Bert Vanhooft, Yolande Berbers, Frédéric Jouault, and Wasif Gilani. Applying Megamodelling to Model Driven Performance Engineering. In *Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 244–253. IEEE Computer Society, 2009.
- [FCB⁺13] Martin Faunes, Juan Cadavid, Benoit Baudry, Houari Sahraoui, and Benoit Combemale. Automatically Searching for Metamodel Well-Formedness Rules in Examples and Counter-Examples. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 187–202. Springer Berlin Heidelberg, 2013.
- [FDR] Failures-Divergence Refinement (FDR3) model checker. <http://www.cs.ox.ac.uk/projects/fdr>, accessed: 10.01.15.
- [FEK⁺07] Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David S. Rosenblum, and Sebastián Uchitel. Model checking service compositions under resource constraints. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 225–234. ACM New York, NY, USA, 2007.

- [FHRW13] Andreas Fuhr, Tassilo Horn, Volker Riediger, and Andreas Winter. Model-driven software migration into service-oriented architectures. *Computer Science - Research and Development*, 28(1):65–84, 2013. Springer Berlin Heidelberg.
- [FNTZ98] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformations (TAGT), Selected Papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer Berlin Heidelberg, 1998.
- [FV09] Marcos Didonet Del Fabro and Patrick Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and Systems Modeling*, 8(3):305–324, 2009. Springer Berlin Heidelberg.
- [GC12] Carlos A. González and Jordi Cabot. ATLTest: A White-Box Test Generation Approach for ATL Transformations. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 449–464. Springer Berlin Heidelberg, 2012.
- [GDA13] Jokin Garcia, Oscar Diaz, and Maider Azanza. Model Transformation Co-evolution: A Semi-automatic Approach. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 144–163. Springer Berlin Heidelberg, 2013.
- [GdLK⁺10] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. *transML: A Family of Languages to Model Model Transformations*. In Dorina C. Petriu, Nicolas Rouquette, and Oystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 106–120. Springer Berlin Heidelberg, 2010.
- [GGMT08] Stefano Gallotti, Carlo Ghezzi, Raffaella Mirandola, and Giordano Tamburrelli. Quality Prediction of Service Compositions through Probabilistic Model Checking. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *Proceedings of the 4th International Conference on the Quality of Software-Architectures (QoSA)*, volume 5281 of *Lecture Notes in Computer Science*, pages 119–134. Springer Berlin Heidelberg, 2008.
- [GK10] Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with

- Triple Graph Grammars. *Software and Systems Modeling*, 9(1):21–46, 2010. Springer Berlin Heidelberg.
- [GL12] Holger Giese and Leen Lambers. Towards Automatic Verification of Behavior Preservation for Model Transformation via Invariant Checking. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 7562 of *Lecture Notes in Computer Science*, pages 249–263. Springer Berlin Heidelberg, 2012.
- [GLKE10] Christian Gerth, Markus Luckey, Jochen Malte Küster, and Gregor Engels. Detection of Semantically Equivalent Fragments for Business Process Model Change Management. In *Proceedings of the 2010 IEEE International Conference on Services Computing (SCC)*, pages 57–64. IEEE Computer Society, 2010.
- [GPR06] Volker Gruhn, Daniel Pieper, and Carsten Röttgers. *MDA: Effektives Software-Engineering mit UML2 und Eclipse*. Springer Berlin Heidelberg, 2006.
- [GPR11] Joel Greenyer, Sebastian Pook, and Jan Rieke. Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In Robert B. France, Jochen Malte Küster, Behzad Bordbar, and Richard F. Paige, editors, *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA)*, volume 6698 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg, 2011.
- [Gro13] Henning Groenda. *Certifying Software Component Performance Specifications*. Number 11 in The Karlsruhe Series on Software Design and Quality. KIT Scientific Publishing Karlsruhe, Germany, 2013.
- [GV09] Iris Groher and Markus Völter. Aspect-Oriented Model-Driven Software Product Line Engineering. *Transactions on Aspect-Oriented Software Development VI*, 6:111–152, 2009. Springer Berlin Heidelberg.
- [HEO⁺11] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 668–682. Springer Berlin Heidelberg, 2011.
- [HEOG10] Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Graph Transformations*, volume 6372 of *Lecture Notes in Computer Science*, pages 155–170. Springer Berlin Heidelberg, 2010.

- [HK03] Jan Hendrik Hausmann and Stuart Kent. Visualizing Model Mappings in UML. In *Proceedings of the 2003 ACM Symposium on Software Visualization (SoftVis)*, pages 169–178. ACM New York, NY, USA, 2003.
- [HKA10] Florian Heidenreich, Jan Kopcsek, and Uwe Aßmann. Safe Composition of Transformations. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2010.
- [HKR⁺10] Mathias Hülsbusch, Barbara König, Arend Rensink, Maria Semenyak, Christian Soltenborn, and Heike Wehrheim. Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In Dominique Méry and Stephan Merz, editors, *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM)*, volume 6396 of *Lecture Notes in Computer Science*, pages 183–198. Springer Berlin Heidelberg, 2010.
- [HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer Berlin Heidelberg, 2002.
- [HM11] Berthold Hoffmann and Mark Minas. Generating Instance Graphs from Class Diagrams with Adaptive Star Grammars. *Electronic Communications of the ECEASST*, 39, 2011. European Association of Software Science and Technology (ECEASST).
- [Hoa85] Charles A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- [HP01] Annegret Habel and Detlef Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin Heidelberg, 2001.
- [IP10] Jerónimo Irazábal and Claudia Pons. Model Transformation Languages Relying on Models as ADTs. In Mark Brand, Dragan Gašević, and Jeff Gray, editors, *Software Language Engineering (SLE)*, volume 5969 of *Lecture Notes in Computer Science*, pages 133–143. Springer Berlin Heidelberg, 2010.
- [ISO01] Extended Backus-Naur Form (EBNF). ISO/IEC 14977, International Organization for Standardization, 2001.

- [JA10] Jendrik Johannes and Uwe Aßmann. Concern-Based (de)composition of Model-Driven Software Development Processes. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, pages 47–62. Springer Berlin Heidelberg, 2010.
- [JAB⁺12] Bevan Jones, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. Semantics-Based Machine Translation with Hyperedge Replacement Grammars. In Martin Kay and Christian Boitet, editors, *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 1359–1376. Indian Institute of Technology Bombay, 2012.
- [JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. Elsevier Science.
- [Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
- [Jen87] Kurt Jensen. Coloured Petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin Heidelberg, 1987.
- [KAER07] Jochen M. Küster and Mohamed Abd-El-Razik. Validation of Model Transformations - First Experiences Using a White Box Approach. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204. Springer Berlin Heidelberg, 2007.
- [Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2nd edition, 2002.
- [KB10] Lucia Kapová and Steffen Becker. Systematic Refinement of Performance Models for Concurrent Component-based Systems. *Proceedings of the 7th International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA), Electronic Notes in Theoretical Computer Science*, 264(1):73–90, 2010. Elsevier Science.
- [Ker] KerMeta Project. <http://www.kermeta.org>, accessed: 13.10.14.
- [KKK⁺06] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 528–542. Springer Berlin Heidelberg, 2006.

- [KKR⁺08a] Gerti Kappel, Horst Kargl, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, Michael Strommer, and Manuel Wimmer. A Framework for Building Mapping Operators Resolving Structural Heterogeneities. In Roland Kaschek, Christian Kop, Claudia Steinberger, and Günther Fliedl, editors, *Information Systems and e-Business Technologies*, volume 5 of *Lecture Notes in Business Information Processing*, pages 158–174. Springer Berlin Heidelberg, 2008.
- [KKR08b] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph Transformation Units - An Overview. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 57–75. Springer Berlin Heidelberg, 2008.
- [Kle06] Anneke Kleppe. MCC: A Model Transformation Environment. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin Heidelberg, 2006.
- [KM10] Pierre Kelsen and Qin Ma. A Modular Model Composition Technique. In David Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin Heidelberg, 2010.
- [KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008.
- [KR10] Lucia Kapova and Ralf Reussner. Application of Advanced Model-Driven Techniques in Performance Engineering. In Alessandro Aldini, Marco Bernardo, Luciano Bononi, and Vittorio Cortellessa, editors, *Computer Performance Engineering*, volume 6342 of *Lecture Notes in Computer Science*, pages 17–36. Springer Berlin Heidelberg, 2010.
- [KSW04] Jochen M. Küster, Shane Sendall, and Michael Wahler. Comparing Two Model Transformation Approaches. In *Proceedings of the Workshop on OCL and Model Driven Engineering*, pages 114–127. Springer Berlin Heidelberg, 2004.
- [KSW⁺13] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *Software and Systems Modeling*, pages 1–36, 2013. Springer Berlin Heidelberg.

- [Küs06] Jochen M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006. Springer Berlin Heidelberg.
- [KW07] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.
- [KW12] Liliya Klassen and Robert Wagner. EMorF - A tool for model transformations. *Electronic Communications of the ECEASST*, 54, 2012. European Association of Software Science and Technology (ECEASST).
- [KWRS11] Gerti Kappel, Manuel Wimmer, Werner Retschitzegger, and Wieland Schwinger. Leveraging Model-Based Tool Integration by Conceptual Modeling Techniques. In Roland Kaschek and Lois Delcambre, editors, *The Evolution of Conceptual Modeling*, volume 6520 of *Lecture Notes in Computer Science*, pages 254–284. Springer Berlin Heidelberg, 2011.
- [LAD⁺14] Levi Lucio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Software and Systems Modeling*, Online First(TBD):1–38, 2014.
- [Lau90] Clemens Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27(5):399–421, 1990. Springer Berlin Heidelberg.
- [LD09] Hongzhi Liang and Juergen Dingel. A Practical Evaluation of Using TXL for Model Transformation. In Dragan Gašević, Ralf Lämmel, and Eric Wyk, editors, *Software Language Engineering (SLE)*, volume 5452 of *Lecture Notes in Computer Science*, pages 245–264. Springer Berlin Heidelberg, 2009.
- [LGJS07] Carsten Lohmann, Joel Greenyer, Juanjuan Jiang, and Tarja Systä. Applying Triple Graph Grammars For Pattern-Based Workflow Model Transformations. *Journal of Object Technology*, 6(9):253–273, 2007.
- [LKR11] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Model-Driven Development of Model Transformations. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 47–61. Springer Berlin Heidelberg, 2011.
- [LKR13] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Constraint-based specification of model transformations. *Journal of Systems and Software*, 86(2):412–436, 2013. Elsevier Science.

- [LKR14] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Model-Transformation Design Patterns. *IEEE Transactions on Software Engineering*, 40(12):1224–1259, 2014. IEEE Computer Society.
- [LPH11] Gilles Lasnier, Laurent Pautet, and Jérôme Hugues. A Model-Based Transformation Process to Validate and Implement High-Integrity Systems. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 67–74. IEEE Computer Society, 2011.
- [LWK10] Philip Langer, Manuel Wimmer, and Gerti Kappel. Model-to-Model Transformations By Demonstration. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations (ICMT)*, volume 6142 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin Heidelberg, 2010.
- [MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. Elsevier Science.
- [MH01] Mark Minas and Berthold Hoffmann. Specifying and Implementing Visual Process Modeling Languages with DiaGen. *Uniform Approaches to Graphical Process Specification Techniques (Satellite Event of ETAPS)*, *Electronic Notes in Theoretical Computer Science*, 44(4):67–79, 2001. Elsevier Science.
- [MM08] Steffen Mazanek and Mark Minas. Parsing of Hyperedge Replacement Grammars with Graph Parser Combinators. *Electronic Communications of the ECEASST*, 10, 2008. European Association of Software Science and Technology (ECEASST).
- [MMea03] Joaquin Miller, Jishnu Mukerji, and et al. *MDA Guide Version 1.0*, 2003. <http://www.omg.org/cgi-bin/doc?omg/2003-05-01>, accessed: 06.10.14.
- [MMPT10] Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien A. Tamburri. Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. *IEEE Transactions on Software Engineering*, 36(1):119–140, 2010. IEEE Computer Society.
- [MOFa] Meta Object Facility (MOF) Core. <http://www.omg.org/spec/MOF>, accessed: 08.10.14.
- [MOFb] MOFScript tool for model to text transformation. <http://eclipse.org/gmt/mofscript>, accessed: 10.01.15.
- [MTM08] Tom Mens, Gabriele Taentzer, and Dirk Müller. *Model-Driven Software Refactoring*, In *Model-Driven Software Development: Integrating Quality Assurance*, pages 170–203. IGI Global Hershey, 2008.

- [NHBE14] Nico Nachtigall, Frank Hermann, Benjamin Braatz, and Thomas Engel. Towards Domain Completeness for Model Transformations Based on Triple Graph Grammars. In Moussa Amrani, Eugene Syriani, and Manuel Wimmer, editors, *Proceedings of the Third International Workshop on Verification of Model Transformations (VOLT)*, volume 1325, pages 46–55. CEUR Workshop Proceedings, 2014.
- [NRKR13] Qais Noorshams, Andreas Rentschler, Samuel Kounev, and Ralf Reussner. A Generic Approach for Architecture-level Performance Modeling and Prediction of Virtualized Storage Systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 339–342. ACM New York, NY, USA, 2013.
- [NSC13] Soroosh Nalchigar, Rick Salay, and Marsha Chechik. Towards a Catalog of Non-Functional Requirements in Model Transformation Languages. In Benoit Baudry, Jürgen Dingel, Levi Lucio, and Hans Vangheluwe, editors, *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT)*, volume 1077. CEUR Workshop Proceedings, 2013.
- [OAS07] OASIS. Web Services Business Process Execution Language Standard, version 2.0, 2007. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, accessed: 10.01.15.
- [OCLa] Eclipse OCL Project. <http://projects.eclipse.org/projects/modeling.mdt.ocl>, accessed: 08.10.14.
- [OCLb] Object Constraint Language (OCL). <http://www.omg.org/spec/OCL>, accessed: 08.10.14.
- [Old05] Jon Oldevik. Transformation Composition Modelling Framework. In Lea Kutvonen and Nancy Alonistioti, editors, *Distributed Applications and Interoperable Systems*, volume 3543 of *Lecture Notes in Computer Science*, pages 108–114. Springer Berlin Heidelberg, 2005.
- [Ome03] Borys Omelayenko. RDFT: A Mapping Meta-Ontology for Web Service Integration. In Borys Omelayenko and Michel C. A. Klein, editors, *Knowledge Transformation for the Semantic Web*, pages 137–153. IOS Press, 2003.
- [OMG] Object Management Group. <http://www.omg.org>, accessed: 10.06.14.
- [PR10] Klaus Pohl and Chris Rupp. *Basiswissen Requirements Engineering - Aus- und Weiterbildung zum "Certified Professional for Requirements Engineering": Foundation Level nach IREB-Standard (2. Aufl.)*. dpunkt.verlag, 2010.
- [Pra71] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971. Elsevier Science.

- [QVTa] mediniQVT Project. <http://projects.ikv.de/qvt>, accessed: 13.10.14.
- [QVTb] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (QVT). <http://www.omg.org/spec/QVT>, accessed: 12.10.14.
- [QVTc] QVT Declarative Project. <http://www.eclipse.org/mmt/?project=qvtd>, accessed: 13.10.14.
- [QVTd] QVT Operational Project. <http://www.eclipse.org/mmt/?project=qvto>, accessed: 13.10.14.
- [QVT02] OMG: MOF 2.0 Query/Views/Transformations RFP, April 2002. <http://www.omg.org/cgi-bin/doc?ad/2002-4-10>, accessed: 15.10.14.
- [RB08] Jorg Rech and Christian Bunse. *Model-Driven Software Development: Integrating Quality Assurance*. Information Science Reference - Imprint of: IGI Publishing, Hershey, 2008.
- [RBB⁺11] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Kozirolek, Heiko Kozirolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical Report 2190-4782, KIT, Fakultät für Informatik, Karlsruhe, 2011.
- [Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific Publishing Co., Inc., 1997.
- [RRLB09] José Eduardo Rivera, Daniel Ruiz-González, Fernando López-Romero, and José María Bautista. Wires*: A Tool for Orchestrating Model Transformations. In Antonio Vallecillo and Goiuria Sagardui, editors, *XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 158–161, 2009.
- [Rum02] Bernhard Rumpe. Agile Modeling with the UML. In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future, 9th International Workshop (RISSEF)*, volume 2941 of *Lecture Notes in Computer Science*, pages 297–309. Springer Berlin Heidelberg, 2002.
- [SBV] Semantics of Business Vocabulary and Rules (SBVR). <http://www.omg.org/spec/SBVR>, accessed: 18.02.15.
- [SC12] Andrew Stevenson and James R. Cordy. Grammatical Inference in Software Engineering: An Overview of the State of the Art. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 204–223. Springer Berlin Heidelberg, 2012.

- [SCGM08] Jesús Sánchez Cuadrado and Jesús García Molina. Approaches for Model Transformation Reuse: Factorization and Composition. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin Heidelberg, 2008.
- [Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin Heidelberg, 1994.
- [Scu05] Frantisek Scuglik. Relation Between UML2 Activity Diagrams and CSP Algebra. In *Proceedings of the 5th WSEAS International Conference on Simulation, Modelling and Optimization*, pages 561–566. World Scientific and Engineering Academy and Society (WSEAS) Stevens Point, Wisconsin, USA, 2005.
- [SE12] Eugene Syriani and Hüseyin Ergin. Operational semantics of UML activity diagram: An application in project management. In *Second IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE)*, pages 1–8. IEEE Computer Society, 2012.
- [Sem12] Maria Semenyak. *Full semantics preservation in model transformation*. PhD thesis, University of Paderborn, 2012.
- [SHNG12] Andreas Seibel, Regina Hebig, Stefan Neumann, and Holger Giese. A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations. In Anthony Sloane and Uwe Aßmann, editors, *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 19–39. Springer Berlin Heidelberg, 2012.
- [SLD08] Jun Sun, Yang Liu, and Jin Song Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer Berlin Heidelberg, 2008.
- [SMR11] Kurt Stenzel, Nina Moebius, and Wolfgang Reif. Formal Verification of QVT Transformations for Code Generation. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 533–547. Springer Berlin Heidelberg, 2011.
- [Sof98] Rational Software. Rational Unified Process: Best Practices for Software Development Teams. *White Paper, IEEE Annals of the History of Computing*, TP026B, Rev 11/01, 1998. IEEE Computer Society.

- [SWG09] Yu Sun, Jules White, and Jeff Gray. Model Transformation by Demonstration. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 712–726. Springer Berlin Heidelberg, 2009.
- [Tae00] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In Manfred Nagl, Andreas Schürr, and Manfred Münch, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE) Workshop*, volume 1779 of *Lecture Notes in Computer Science*, pages 481–488. Springer Berlin Heidelberg, 2000.
- [Tis14] Massimo Tisi. Turing Completeness of ATL. private communication, 2014.
- [UML] Unified Modeling Language (UML). <http://www.omg.org/spec/UML>, accessed: 08.10.14.
- [VAB⁺08] Dániel Varró, Mark Asztalos, Dénes Bisztray, Artur Boronat, Duc-Hanh Dang, Rubino Geiß, Joel Greenyer, Pieter Van Gorp, Ole Kniemeyer, Anantha Narayanan, Edgars Rencis, and Erhard Weinell. Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 5088 of *Lecture Notes in Computer Science*, pages 540–565. Springer Berlin Heidelberg, 2008.
- [Var06] Dániel Varró. Model Transformation by Example. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 410–424. Springer Berlin Heidelberg, 2006.
- [VAVB⁺07] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. UniTI: A Unified Transformation Infrastructure. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2007.
- [VCFM10] Juan M. Vara, Maria Valeria De Castro, Marcos Didonet Del Fabro, and Esperanza Marcos. Using Weaving Models to Automate Model-Driven Web Engineering Proposals. *International Journal of Computer Applications in Technology*, 39(4):245–252, 2010. Inderscience Publishers.
- [VG08] Pieter Van Gorp. *Model-driven Development of Model Transformations*. PhD thesis, 2008. Universiteit Antwerpen (Belgium).
- [VHV08] Gergely Varró, Ákos Horváth, and Dániel Varró. Recursive Graph Pattern Matching. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*,

- volume 5088 of *Lecture Notes in Computer Science*, pages 456–470. Springer Berlin Heidelberg, 2008.
- [VVBJB07] Bert Vanhooft, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Traceability as input for model transformations. *Third ECMDA traceability workshop*, 2007:37–46, 2007.
- [VVG⁺06] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination Analysis of Model Transformations by Petri Nets. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 260–274. Springer Berlin Heidelberg, 2006.
- [VVL07] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing (ICSOC)*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin Heidelberg, 2007.
- [Wag08] Dennis Wagelaar. Composition Techniques for Rule-Based Model Transformation Languages. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2008.
- [WH05] Edward D. Willink and Philip J. Harris. The Side Transformation Pattern: Making Transforms Modular and Re-usable. *Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra)*, *Electronic Notes in Theoretical Computer Science*, 127(3):17–29, 2005. Elsevier Science.
- [WK06] Manuel Wimmer and Gerhard Kramler. Bridging Grammarware and Modelware. In Jean-Michel Bruel, editor, *Satellite Events at the Model Driven Engineering Languages and Systems*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer Berlin Heidelberg, 2006.
- [WKC06] Junhua Wang, Soon-Kyeong Kim, and David Carrington. Verifying Meta-model Coverage of Model Transformations. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 270–282. IEEE Computer Society Washington, DC, USA, 2006.
- [WKK⁺12] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Fact or Fiction - Reuse in Rule-Based Model-to-Model Transformation Languages. In Zhenjiang Hu and Juan de Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 280–295. Springer Berlin Heidelberg, 2012.

- [WKR⁺11] Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Jesús Sánchez Cuadrado, Esther Guerra, and Juan De Lara. Reusing Model Transformations Across Heterogeneous Metamodels. In *Proceedings of the 5th International Workshop on Multi-Paradigm Modeling*. European Association of Software Science and Technology (ECEASST), 2011.
- [WKS⁺09] Manuel Wimmer, Gerti Kappel, Johannes Schönböck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. TROPIC: a framework for model transformations on petri nets in color. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 783–784. ACM New York, NY, USA, 2009.
- [WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.
- [WTEK08] Jessica Winkelmann, Gabriele Taentzer, Karsten Ehrig, and Jochen M Küster. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. *Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Electronic Notes in Theoretical Computer Science*, 211(0):159–170, 2008. Elsevier Science.
- [WW13] Sven Walther and Heike Wehrheim. Knowledge-Based Verification of Service Compositions - An SMT Approach. In *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems*, pages 24–32. IEEE Computer Society, 2013.
- [XMP09] Dong Xu, Huaikou Miao, and N. Philbert. Model Checking UML Activity Diagrams in FDR. In *Proceedings of the Eighth IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, pages 1035–1040. IEEE Computer Society, 2009.
- [ZSZ11] Iyad Zikra, Janis Stirna, and Jelena Zdravkovic. Analyzing the Integration between Requirements and Models in Model Driven Development. In Terry Halpin, Selmin Nurcan, John Krogstie, Pnina Soffer, Erik Proper, Rainer Schmidt, and Ilia Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing*, pages 342–356. Springer Berlin Heidelberg, 2011.

List of Figures

2.1	Tasks of the MDSD process arranged in a cycle (from [GPR06])	12
2.2	Models in the MDA process (based on [GPR06])	14
2.3	Key concepts of MDA (adapted from [GPR06]).	15
2.4	Meta-model based language definition stack in general (left) and for UML in MDA (right) (from [GPR06]).	16
2.5	Main elements of MOF (from [MOFa]) in Ecore.	17
2.6	Fragment of the UML Superstructure for activity diagrams (from [UML]) in Ecore.	18
2.7	Simple activity model in concrete (left) and abstract (right) syntax	19
2.8	Fragment of the UML Superstructure with structuring classes in Ecore . .	20
2.9	Well-formedness constraint 4 for a join node in OCL (from [UML], p. 394)	20
2.10	Meta-model based definition of transformation mappings (adapted from [MMea03])	22
3.1	PIM models of the example multi-layer software system	25
3.2	Component diagram of the $M_{application}$ model of the application layer . . .	26
3.3	Structure of the composite component <i>TripCoordinator</i>	27
3.4	Activities modeling the behavior of the operations of the application layer	28
3.5	Component diagrams of the PIM models of the platform layer	28
3.6	Activities for the operations provided by the platform layer	29
3.7	Meta-model for the Activity Diagrams with Resources (ADR) language .	30
3.8	Well-formedness constraints on an initial activity node in OCL	32
3.9	ADR model fragment with the activity <i>RpcSender.transmit()</i> as an in- stance of the meta-model from Figure 3.7 (abstract syntax)	32
3.10	Meta-model for the CSP modeling language	35
3.11	Call of the operation <i>MapDB.getMap()</i> (a) and a part of the result of its transformation into CSP (b)	36
3.12	Activity <i>ThreadPool.acquire()</i> (a) and a part of its transformation into CSP (b)	37
3.14	Resource <i>Thread</i> of the component <i>ThreadPool</i> (a) and a part of its trans- formation into CSP (b)	38
3.15	Fragment of the transformation of the <i>TripPlanner.getTrip()</i> activity into CSP	46
4.1	Classification of model transformations based on [GPR06]	49

4.2	Selected high-level mappings for the ADR-to-CSP transformation (notation from [GPR06])	51
4.3	Sketch of the meta-model based transformation definition approach	52
4.4	Selected detailed meta-model based transformation mappings for the ADR-to-CSP transformation	52
4.5	Structuring classes for the ADR meta-model from Figure 3.7 in Ecore . .	53
4.6	Detailed meta-model based mapping for the structuring class <i>SequenceNode</i>	54
4.7	TGG production sketch	64
4.8	Forward interpretation of an example TGG production from Figure 4.7 .	65
4.9	Axiom of our TGGs based ADR-to-CSP transformation implementation .	66
4.10	TGG rule for the high-level mapping <i>Activity2Process</i> of ADR-to-CSP . .	67
4.11	TGG rule for the high-level mapping <i>InitialNode2Prefix</i> of ADR-to-CSP .	68
4.12	TGG rule for the high-level mapping <i>FinalNode2Prefix</i> of ADR-to-CSP .	68
4.13	TGG rule for the high-level mapping <i>DecisionMergeBlock2Condition</i> of ADR-to-CSP	69
4.14	Transformation chaining sketchl	84
4.15	Transformation sequencing: UML Activity diagram to CSP through ADR	86
4.16	Transformation parallelization: ADR models of system layers and their interactions/inter-relations into CSP	87
5.1	A derivation tree deriving $\sigma_2\sigma_1\sigma_2\sigma_3$ from S in G_σ	91
5.2	Sketch of production application in HR grammars	93
5.3	HR grammar productions for the ADR modeling language (abstract syntax)	95
5.4	Some productions of the HR grammar from Figure 5.3 in concrete syntax	96
5.5	HR grammar productions for the CSP language (abstract syntax)	97
5.6	Overview of our grammar based approach to transformation development	98
5.7	GBMT rule: Activity block sequence to process expression sequence . . .	100
5.8	Further GBMT rules for the ADR-to-CSP transformation	101
5.9	GBMT rule: ADR model to a set of processes and an event set	103
5.10	GBMT rule: Non-empty activity to a processes sub-expression	104
5.11	GBMT rule: Fork/join structure to a parallel process composition	105
5.12	GBMT rule: Passive resource to a set of processes, channels, and value sets	106
5.13	GBMT rule: Loop structure to an equivalent recursive process and a referencing process expression	108
5.14	GBMT rule: Internal action to an event prefix and a channel	108
5.15	GBMT rules: Activity (entry) to a channel, process, and a value set (with additional value)	109
5.16	GBMT execution sketch	111
5.17	Replacement alternatives for sub-tree t in the ADR HR grammar	119
5.18	Tool support for GBMTs with TGGs/EMorF as an implementation platform	148
5.19	Fragment of our meta-model for HR grammars in Ecore	149
5.20	Combination of features (from [CH06]) for the meta-model based implementation of GBMTs	149
5.21	Example structure based mapping: flattening hierarchical activities	157

6.1	Overview of our weaving based transformation re-use approach	166
6.2	Abstract core weaving meta-model of AMW from [DdF07]	168
6.3	Weaving meta-model for multi-layer system configuration modeling in Ecore	170
6.4	Weaving meta-model from Figure 6.3 in a tree-based representation . . .	171
6.5	Part of a configuration model for a multi-layer system with the application and the threadpool platform layer model from Chapter 3	172
6.6	Part of a configuration model for a multi-layer system with the application and the RPC platform layer model from Chapter 3	172
6.7	T_{LModel} : Layer model transformation into CSP followed by event splitting	174
6.8	Fragment of ADR-to-CSP transformation result for <i>MapDB.getMap()</i> call after the event splitting (<i>C1ID</i> abbreviates <i>CallAction1ID</i>)	175
6.9	Fragment of ADR-to-CSP transformation result for <i>ThreadPool.acquire()</i> activity after the event splitting	175
6.10	Fragment of the ADR-to-CSP transformation result for resource <i>Thread</i> after the event splitting	176
6.11	T_{Conf} : Configuration model transformation into CSP fragments	178
6.12	CM-to-CSP transformation result (b) for resource handling weaving ‘ <i>On getMap invoke acquire</i> ’ (a)	179
6.13	Fragment of the CM-to-CSP transformation result (b) for communication weaving ‘ <i>On getMap callAction1 RPC transmit</i> ’ (a)	180
6.14	Tool support for the weaving based re-use of model transformations . . .	188
7.1	Grammar based definition of the event splitting transformation in CSP . .	199
7.2	Non-changing GBMT rule for the process production from the CSP HR grammar	200
7.3	Extended system configuration meta-model in Ecore	204
7.4	Extended system configuration HR grammar	205
7.5	GBMT rules for requirements 1 and 2 on the CM-to-CSP transformation	206
7.6	Common GBMT rule for requirements 3 – 12 on the CM-to-CSP trans- formation	207
7.7	Type specific GBMT rule set for requirements 3 and 4 on the CM-to-CSP transformation	209
7.8	Type specific GBMT rule set for requirements 3 and 10 on the CM-to-CSP transformation	210
8.1	Flattening hierarchical activities using an idle target production	224

List of Tables

3.1	Overview of requirements on the example ADR-to-CSP transformation . .	43
5.1	Classification of model transformation development approaches, considered in Sections 4.2.2 and 5.6, and of our approach, published in [BSW14]	159
6.1	Classification of the considered re-use approaches and of our approach . .	165
6.2	Overview of requirements on our CSP event splitting transformation . . .	177
6.3	Overview of requirements on our CM-to-CSP transformation	186
6.4	Considered multi-layer system configurations and values of sample properties	195