Consistency-by-Construction Techniques for Software Models and Model Transformations

DISSERTATION
FOR THE DEGREE OF DOCTOR OF NATURAL SCIENCES
(DR. RER. NAT.)

Submitted by
Nebras Nassar, M.Sc.
born May 20, 1984 in Damascus, Syria.

Referees:
Prof. Dr. Gabriele Taentzer, Philipps-Universität Marburg, Germany.
Prof. Dr. Reiko Heckel, University of Leicester, United Kingdom.

Examination Committees:
Prof. Dr. Bernhard Seeger (Chairman and Dean),
Prof. Dr. Gabriele Taentzer,
Prof. Dr. Reiko Heckel,
Prof. Dr. Thorsten Thormählen,
Prof. Dr. Christoph Bockisch

Submitted on September 23, 2019.
Defended on December 18, 2019.
Department of Mathematics and Computer Science
Philipps-Universität Marburg (1180)
Marburg, Germany, 2020.

Nassar, Nebras
Consistency-by-Construction Techniques for Software Models and Model Transformations.
Dissertation, Philipps-Universität Marburg (1180), 2020.

Curriculum vitae

- 2014, Master of Science (Informatics), Philipps-Universität Marburg, Germany.
- 2008, Bachelor in Informatics Engineering (Information Systems and Software Engineering), Tishreen University, Lattakia, Syria.
- Awards and Grants
  - Best Software Science Paper awarded by the European Association of Software Science and Technology (EASST award).
  - The German Academic Exchange Service (DAAD) Scholarship for Master.

Originaldokument gespeichert auf dem Publikationsserver der
Philipps-Universität Marburg
http://archiv.ub.uni-marburg.de

Dieses Werk bzw. Inhalt steht unter einer
Creative Commons
Namensnennung
Keine kommerzielle Nutzung
Weitergabe unter gleichen Bedingungen
3.0 Deutschland Lizenz.

Die vollständige Lizenz finden Sie unter:
http://creativecommons.org/licenses/by-nc-sa/3.0/de/

Für Theodor und Sally
in Liebe

Contents

List of Figures

List of Tables

Abstract

A *model is consistent* with given specifications (specs) if and only if all the specifications are held on the model, i.e., all the specs are true (correct) for the model.

Constructing consistent models (e.g., programs or artifacts) is vital during software development, especially in Model-Driven Engineering (MDE), where models are employed throughout the life cycle of software development phases (analysis, design, implementation, and testing). Models are usually written using domain-specific modeling languages (DSMLs) and specified to describe a domain problem or a system from different perspectives and at several levels of abstraction. If a model conforms to the definition of its DSML (denoted usually by a meta-model and integrity constraints), the model is consistent.

Model transformations are an essential technology for manipulating models, including, e.g., refactoring and code generation in a (semi)automated way. They are often supposed to have a well-defined behavior in the sense that their resulting models are consistent with regard to a set of constraints. Inconsistent models may affect their applicability and thus the automation becomes untrustworthy and error-prone. The consistency of the models and model transformation results contribute to the quality of the overall modeled system.

Although MDE has significantly progressed and become an accepted best practice in many application domains such as automotive and aerospace, there are still several significant challenges that have to be tackled to realize the MDE vision in the industry. Challenges such as handling and resolving inconsistent models (e.g., incomplete models), enabling and enforcing model consistency/correctness during the construction, fostering the trust in and use of model transformations (e.g., by ensuring the resulting models are consistent), developing efficient (automated, standardized and reliable) domain-specific modeling tools, and dealing with large models are continually making the need for more research evident.

In this thesis, we contribute four *automated interactive techniques for ensuring the consistency of models and model transformation results during the construction process.* The first two contributions construct consistent models of a given DSML in an automated and interactive way. The construction can start at a seed model being potentially inconsistent.

Since enhancing a set of transformations to satisfy a set of constraints is a tedious and error-prone task and requires high skills related to the theoretical foundation, we present the other contributions. They ensure model consistency by enhancing the behavior of model transformations through automatically constructing application conditions. The resulting application conditions control the applicability of the transformations to respect a set of constraints. Moreover, we provide several optimizing strategies.

Specifically, we present the following:

First, we present a *model repair technique* for repairing models in an automated and interactive way. Our approach guides the modeler to repair the whole model by resolving all the cardinalities violations and thereby yields a desired, consistent model. Second, we introduce a *model generation technique* to efficiently generate large, consistent, and diverse models. Both techniques are DSML-agnostic, i.e., they can deal with any meta-models. We present meta-techniques to instantiate both approaches to a given DSML; namely, we develop meta-tools to generate the corresponding DSML tools (model repair and generation) for a given meta-model *automatically.* We present the soundness of our techniques and evaluate and discuss their features such as scalability.

Third, we develop a tool based on a correct-by-construction technique for translating OCL constraints into semantically equivalent graph constraints and integrating them as *guaranteeing* application conditions into a transformation rule in a fully automated way. A constraint-guaranteeing application condition ensures that a rule applies successfully to a model if and only if the resulting model after the rule application satisfies the constraint. Fourth, we propose an *optimizing-by-construction technique* for application conditions for transformation rules that need to be *constraint-preserving.* A constraint-preserving application condition ensures that a rule applies successfully to a consistent model (w.r.t. the constraint) if and only if the resulting model after the rule application still satisfies the constraint. We show the soundness of our techniques, develop them as ready-to-use tools, evaluate the efficiency (complexity and performance) of both works, and assess the overall approach in general as well.

All our four techniques are compliant with the *Eclipse Modeling Framework (EMF)*, which is the realization of the OMG standard specification in practice. Thus, the interoperability and the interchangeability of the techniques are ensured. Our techniques not only improve the quality of the modeled system but also increase software productivity by providing meta-tools for generating the DSML tool supports and automating the tasks.

Überblick

Ein Softwaremodell ist mit gegebenen Spezifikationen (Specs) genau dann konsistent, wenn alle Spezifikationen von dem Softwaremodell eingehalten werden, d.h. alle Spezifikationen für das Softwaremodell wahr (korrekt) sind.

Während der Softwareentwicklung ist die Konstruktion von konsistenten Softwaremodellen (z.B. Programmen oder Artefakten) essentiell. Dies gilt besonders im Bereich des Model-Driven Engineering (MDE), in welchem die Softwaremodelle in allen Phasen des Softwareentwicklungsprozesses (Analyse, Design, Implementierung und Test) verwendet werden. Softwaremodelle werden üblicherweise in domänenspezifischen Modellierungssprachen (DSMLs) verfasst und dienen der Beschreibung eines Domänenproblems oder eines Systems aus unterschiedlichen Perspektiven und auf unterschiedlichen Abstraktionsebenen. Wenn das Softwaremodell mit der Definition seiner DSML (gewöhnlich durch ein Meta-Modell und Integritätsbedingungen definiert) konform ist, gilt das Softwaremodell als konsistent.

Modelltransformationen sind eine essentielle Technologie zur (semi)-automatisierten Manipulation von Softwaremodellen, inkl. z.B. des Refactorings und der Codegenerierung. Dabei wird oftmals ein wohldefiniertes Transformationsverhalten vorausgesetzt, in dem Sinne, dass die resultierenden Softwaremodelle in Hinblick auf die Bedingungen konsistent sind. Inkonsistente Softwaremodelle beeinflussen die Anwendbarkeit von Modelltransformationen, wodurch die automatische Ausführung unzuverlässig und fehleranfällig werden kann. Die Konsistenz von Softwaremodellen und den Ergebnissen von Modelltransformationen trägt zur Qualität des gesamten modellierten Systems bei.

Obwohl MDE bemerkenswerte Fortschritte gemacht hat und ein akzeptiertes Verfahren in vielen Anwendungsbereichen wie der Automobilindustrie sowie der Luft- und Raumfahrt darstellt, so gilt es immer noch deutliche Herausforderungen zu bewältigen, um die MDE-Vision in der Industrie umzusetzen. Die Herausforderun-

gen bestehen dabei in dem Umgang und der Auflösung von Inkonsistenzen in Softwaremodellen (z.B. unvollständigen Softwaremodellen), der Sicherstellung und Erhaltung von Modellkonsistenz und Korrektheit während der Modellkonstruktion, der Erhöhung der Zuverlässigkeit von Modelltransformationen (z.B. durch die Sicherstellung der Modellkonsistenz nach Modelltransformationen), der Entwicklung von effizienten (automatisierten, standardisierten und zuverlässigen) domänenspezifischen Modellierungswerkzeugen und dem Umgang mit großen Softwaremodellen, was insgesamt die Notwendigkeit weiterer Forschung zeigt.

In dieser Arbeit werden vier automatisierte und interaktive Techniken zur Sicherstellung der Konsistenz von Softwaremodellen und Modelltransformationsergebnissen innerhalb des Softwareentwicklungsprozesses vorgestellt. Die ersten beiden Beiträge erlauben die Konstruktion von konsistenten Softwaremodellen einer gegebenen DSML in einer automatisierten und interaktiven Weise. Die Konstruktion kann dabei mit einem potentiell inkonsistenten Softwaremodell beginnen.

Da die Erweiterung von Transformationen zur Erfüllung von Bedingungen eine langwierige und fehleranfällige Aufgabe darstellt, welche hohe Fähigkeiten in Bezug auf die theoretischen Grundlagen voraussetzt, ergeben sich die weiteren Beiträge: Die vorgestellten Techniken stellen die Modellkonsistenz nach einer automatischen Erweiterung der Modelltransformation durch zusätzliche Anwendungsbedingungen (engl. Application Conditions - ACs) sicher. Diese resultierenden Anwendungsbedingungen steuern die Anwendbarkeit der Transformationen in Bezug auf eine Menge von Konsistenzbedingungen. Darüber hinaus werden zusätzlich Optimierungsstrategien bereitgestellt.

Im Einzelnen wird folgendes präsentiert:

Als Erstes wird eine automatische und interaktive Technik zur Reparatur von Softwaremodellen präsentiert. Dieser Ansatz leitet Modellierer beim Reparieren des gesamten Modells, indem alle Kardinalitätsverletzungen aufgelöst werden, und führt somit zu dem gewünschten konsistenten Softwaremodell. Zweitens wird eine Technik zur effizienten Generierung von großen, konsistenten und heterogenen Softwaremodellen eingeführt. Beide Techniken sind DSML-unabhängig, d.h. sie können für beliebige Meta-Modelle eingesetzt werden. Es werden Meta-Techniken für die Anwendung beider Ansätze auf einer gegebenen DSML präsentiert, da mittels Meta-Tools entsprechende DSML-Werkzeuge (Modellreparatur und Modellgeneration) auf Basis eines gegebenen Meta-Modells automatisch generierbar sind. Es wird die Korrektheit dieser Techniken sowie die Auswertung und Diskussion der Eigenschaften (z.B. Skalierbarkeit) gezeigt.

Drittens ist ein Werkzeug entwickelt worden, basierend auf einem konstruktiven Ansatz zur Übersetzung von OCL Bedingungen in semantisch äquivalente Graphbedingungen, welches diese als gewährleistende Anwendungsbedingungen vollautomatisch in Transformationsregeln integriert. Eine bedingungsgewährleistende Anwendungsbedingung stellt sicher, dass eine Transformationsregel nur genau dann auf einem beliebigen Softwaremodell ausgeführt werden kann, wenn das resultierende Softwaremodell nach der Regelanwendung die Bedingung erfüllt. Viertens wurde ein konstruktiver Ansatz zur Optimierung von Anwendungsbedingungen für bedingungserhaltende Transformationsregeln entwickelt. Eine bedingungserhaltende Anwendungsbedingung stellt sicher, dass eine Regel erfolgreich auf ein konsistentes Softwaremodell, welches die Bedingung erfüllt, angewandt werden kann, genau dann, wenn das resultierende Softwaremodell nach der Regelanwendung immer noch konsistent mit der Bedingung ist. Es wird die Korrektheit der Techniken, die Einsatzfertigkeit der Werkzeuge, die Evaluation der Effizienz (Komplexität und Leistungsfähigkeit) beider Ansätze sowie die Bewertung des gesamten Ansatzes gezeigt.

Die vier Techniken sind kompatibel zu dem Eclipse Modeling Framework (EMF), welches die Realisierung der OMG-Standard-Spezifikation in der Praxis darstellt. Daher sind die Interoperabilität und die Austauschbarkeit der Techniken sichergestellt. Die vorgestellten Techniken verbessern nicht nur die Qualität des modellierten Systems, sondern erhöhen auch die Produktivität durch die Bereitstellung von Meta-Tools zur Generierung von DSML-Werkzeugen, welche die Aufgabenbearbeitung durch Automatisierung beschleunigen.

Acknowledgements

This thesis would not have been mastered without the help of numerous people who provided guidance, friendship, constructive criticism, or support. Therefore, I want to thank all who in one way or another contributed to the completion of this thesis.

First and foremost, I offer my sincerest gratitude to my supervisor, Prof. Dr. Gabriele Taentzer, for her continuous support and guidance during my study in Marburg as a Master and Ph.D. student. Her knowledge, advice, and trust advanced my research capacity significantly not only in the field of software engineering but also extended my scientific horizons and skills to explore and contribute to the theoretical filed.

I am very thankful to Prof. Dr. Reiko Heckel for providing very helpful feedback that helped me to improve the overall presentation, for the constructive comments, and for immediately agreeing to be my second referee. His work on graph conditions is one of the first and the essential works which constitutes the second part of this work.

I like to thank Prof. Dr. Bernhard Seeger, Prof. Dr. Christoph Bockisch, and Prof. Dr. Thorsten Thormählen for immediately agreeing to be members of my examination committee.

I thank the members of the Oldenburg-Marburg DFG (German Research Foundation) project "Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages" Thank you Prof. Dr. Annegret Habel, Dr. Hendrik Radke, Jan Steffen Becker and Christian Sandmann for fruitful discussions.

I am especially thankful to Dr. Thorsten Arendt and Jens Kosiol for many fruitful hours and evenings for co-developing. Thank you, Dr. Thorsten Arendt and Jan Steffen Becker, who contributed to the implementation of tool *OCL2AC*. A big thank goes to Jens Kosiol, who took a great care in ensuring the soundness of the formal aspects of my work, especially the proof of the last work. I want to thank Dr. Steffen

Vauple and Jens Kosiol for proofreading different parts of my thesis and Dr. Steffen Vauple for helping in translating the abstract from English to German.

I would like to thank Prof. Dr. Andy Schürr and his research group in Darmstadt and Prof. Dr. Timo Kehrer in Berlin for interesting discussions and workshops.

During my time in Marburg, many people provided help and friendship. I like to thank (in no special order) Dr. Steffen Vaupel, Dr. Daniel Strüber, Dr. Thorsten Arendt, Kristopher Born, Stefan John, Jens Kosiol, Felix Rieger, Dennis Priefer, Wolf Rost, Michael Lüttebrandt, Viola Wenz, Arno Kesper, Barbara Dinklage, Prof. Dr. Christoph Bockisch, Stefan Schulz, Tan liang, Prof. Dr. Christian Komusiewicz, Niels Grüttemeier, Frank Sommer, Nils Morawietz, Prof. Dr. Thorsten Thormählen, Jiangbin Gan, Sebastian Lieb, Prof. Dr. H.P Gumm, Mehdi Zarrad, Mona Taheri, Lukas Schiller, and Antonio Garmendia for contributing to my journey. Thanks to my friends Gunnesch, Benavides, Haizmann and Bartlett. Thank you all for making my time in Marburg such a valuable experience.

A heartfelt ”thank you” goes to my parents, Hanna (Johannes in German) and Mona, to my sister Louna and her family, to my brother Areen, to my beautiful and lovely wife Sally, and to my wonderful son Theodor (Theo). Although my whole study journey has been accompanied by worries and fears for my country and for my family in Syria because of the war, their support and faith in science were unlimited, indescribable, and full of hope.

1 Introduction

Software engineers are frequently confronted by continuously increasing complexity, required higher quality, and accelerating productivity of software development. Moreover, one of the main obstacles facing software development is that developing software applications requires a full understanding of both *the domain space* such as a bank system or an automotive system and *the implementation space* such as a specific programming language and platform. The growth of software complexity, the need for better software quality and productivity, and the need for information interchange and interoperability between tools and organizations are the main motivations behind the use of **Model-Driven Engineering (MDE)** [143, 133].

MDE [119, 133] is an approach in software engineering that raises the level of abstraction and focuses on the domain problem rather than the underlying environment. *Model-Driven Development (MDD)*, a field of MDE, employs the modeling approach efficiently by introducing more automation in transforming models describing domain aspects into software implementations [46, 133]. For example, a model can be specified to describe a domain part like a web page of a bank and then can be translated into code (e.g., HTML) using prescribed automation. However, during the development process, an engineer has to specify: (a) which artifacts are to be produced (product dimension) and (b) how to process for producing the artifacts (process dimension) [42].

In MDD, software applications are derived from software models. Therefore, software models have to be specified precisely. Otherwise, the automation may result in software applications full of errors. **A model** is a specification (formal description) of the structure and behavior of a system within a given context and from a specific point of view. **A software model** is a model whose specification is based on a language that has *a well-defined meaning* with each of its constructs [143]. A language consists of the following main ingredients: The *abstract syntax* which is a data structure that holds the core information in a model, and the *concrete syntax* which defines the notation (e.g., a graphical syntax and a textual syntax) with which users can express models [145]. Software models are written using **Domain-Specific**

Modeling Languages (DSMLs) [45] whose abstract syntax is usually defined by a meta-model within a particular domain. **A meta-model** is a special kind of model that describes the structure, elements, properties, and the relations between the elements of the model within a particular domain. A meta-model may contain integrity constraints (i.e., well-formedness constraints), and these constraints may be implicitly defined (like the multiplicity constraints of associations) or explicitly specified using a constraint language [25]. **Constraints** are boolean expressions that check some property of a model [145].

A model is consistent with a meta-model if and only if all the constraints of the meta-model are held on the model, i.e., no constraint of the meta-model is *false* for the model [25, 123]. Thus, a meta-model can also be expressed as the set of all possible consistent models. Developers use a DSML to describe particular applications by constructing consistent models (i.e., models conforming to the meta-model of the DSML). In this thesis, we use the term model to refer to a software model.

A model transformation is the automatic generation of a target model from a source model. Model transformations are the key operations for model manipulation involving, e.g., translation, optimization, synchronization, and the (semi-)automatic construction of the software implementation [130]. **A transformation rule** is a description of how a source model can be transformed into a target model [67, 84]. There are two main kinds of model transformations: endogenous and exogenous transformations [83, 30]. The input model and the output model of an endogenous transformation are of the same modeling language. In contrast, the input model and the output model of an exogenous transformation may be of different languages. In the scope of this work, we consider the endogenous model transformations.

Applying a model transformation to a model may change the consistency state of the model. We call a model transformation **consistency-guaranteeing** if applying it to a model, the resulting model is consistent. We call a model transformation **consistency-preserving** if applying it to a consistent model, the resulting model is still consistent [83, 55].

Ensuring model consistency is crucial during software developments since the consistency of the models affects the quality of the generated artifacts (e.g., code) and the applicability of the tools. Software applications usually expect specific input data that fulfill certain constraints or formats. Otherwise, they often fail to handle the data correctly. For example, the C compiler demands a valid C program to execute it, and the UML tool requires a valid XML file to import it, i.e., a model that fulfills the language specification of UML. Importing an invalid XML file may fail [145].

Additionally, **inconsistencies** affect the applicability of the transformations, and thus, the automation becomes untrustworthy, error-prone, and even impossible. Therefore, the consistency of transformation results affects the quality of the modeled system as well. For example, in safety-critical systems, model transformations are required to fulfill or preserve specific consistency properties [48, 72]. Otherwise, the modeled system may fail.

The ability to *detect and resolve inconsistencies early* during software development is desirable to reduce the development cost, time, and improve the software quality [143].

To empower the modeling process and the interoperability between tools and organizations, the Object Management Group (OMG) [100], which is a computer industry standards consortium, provides several computer standard specifications such as the Unified Modeling Language (UML) and the Object Constraint Language (OCL). UML [100, 133] is a standardized general-purpose modeling language to address the modeling of architecture, objects, and the interactions between objects. OCL [99] is a well-known language used to express and specify constraints and query expressions on models. The modeling development is powered by OMG specifications such as:

- the *Meta object facility (MOF)* [133, 98] for specifying models and meta-models,
- the *XML Metadata Interchange (XMI)* [101] which is a standard interchange mechanism used between various tools, repositories and middle-ware to serialize MOF meta-models and models into XML-based schemes, and
- the *Query/View/Transformation (QVT)* [102] for specifying model transformations to manipulate models.

Models are usually classified into meta-layers according to their abstraction levels [98]. Models at a layer are instances of the model at the next higher layer that represents their meta-model. Briefly, the key *modeling concepts* are Class and Object, and the ability to navigate from an object to its meta-object (Class) and vice versa.

In practice, *the Eclipse Modeling Framework (EMF)* [34] has evolved to a de-facto standard technology for defining models and modeling languages in academia as well as in industrial contexts [135]. EMF provides the Ecore meta-metamodel which implements an essential subset of MOF specification. The structure of Ecore is similar to the UML class diagram. With the help of the Ecore meta-metamodel, one can define meta-models addressing different domains such as an automotive domain, mobile application domain, or web page domain. By instantiating the domain meta-model, one can specify instance models describing particular aspects of the domain, e.g., specifying web pages for a bank or a university. Throughout this thesis, our techniques are based on EMF.

Tools have played a critical role in improving software quality and productivity by assisting tasks in software development processes, facilitating the user interaction to design the desired content, ensuring the well-formedness of content during the construction process and verifying critical properties of the systems [21, 60, 19].

In MDE, tools are essential since they can reduce the accidental complexities associated with understanding the problem, dealing with meta-models, and manipulating their models. Two different kinds of tools according to stakeholders can be distinguished during the development process as shown in Figure 1.1 and described below:

- **modeler tools (DSML tools)** that are used by modelers (model owners) to, e.g., edit or repair models of a specific domain, and

- **developer tools (meta-tools)** that are used to build the modeler tools for a particular domain (DSML tools). A meta-tool usually realizes a generic approach, customizes the approach to the given domain, and builds the DSML tool supports.

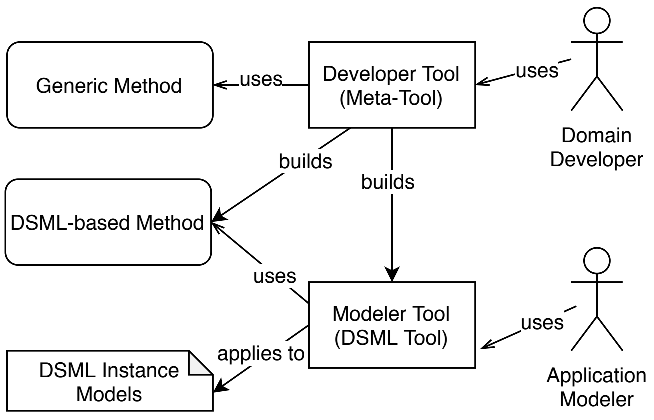


Figure 1.1: A scene of the main ingredients (stakeholders, roles, and tools) in the modeling process.

There are already a lot of tool supports and frameworks supporting the development of DSMLs and their tools such as the Eclipse Modeling Framework (EMF) [34] for defining modeling languages and for building tool supports and applications based on structured data models, and Henshin [3], ATL [64], eMoflon [76] for defining model transformations.

1.1 Challenges and Goals

As modeled systems become more and more complex, consistency problems become more prominent, especially if the model is developed throughout multiple phases or iterations and from different views and concerns. In [46, 136, 69, 124], several challenges dealing with the consistency of model and model transformations, and with developing efficient DSML tools are identified, examples of which are discussed below.

1.1.1 Developing Efficient Techniques for Constructing Consistent Models

During software development, model inconsistency can easily occur due to several reasons such as lack of information, misunderstandings, and the incremental development of models from various aspects.

Resolving and handling incomplete and inconsistent models is a crucial challenge in model-driven engineering (MDE) since it requires high facilities related to the domain. For example, it demands the understanding of the domain structure and its constraints, detecting the inconsistencies in the given instance model, and identifying which repair actions must or could be applied and in which order. Moreover, guiding the modelers in repairing models in a *semi-automated* way is promising for building models efficiently and getting the desired ones, especially if the models have to be constructed throughout several iterations, at different times and from different perspectives. The *interactive* construction of models is promising for producing desired models.

Furthermore, the need for instance models grows with the steady increase of domains and topics to which model-driven engineering (MDE) is applied. In particular, there is a growing need for generating large instances of a given meta-model [70, 118]. *Large and diverse models* are needed in various applications like model transformation testing, benchmark model queries, or validating the suitability of MDE tools to deal with large input models.

Additionally, the development of DSML tools is a general challenge since each DSML needs its own set of tools, e.g., tools for repairing and generating consistent models. Although there are technologies that help developers to build tools for a given DSML, developing and evolving them using *manual* techniques can be expensive. They require high skills related to considered problems, meta-modeling, and tool developments. MDE must be supported by *meta-tools (developer tools)* for building or generating tools for a given DSML [46].

The data *interchangeability* among tools is one of the main problems facing software development. To enable interoperation between tools, the artifacts produced by one tool should be at least accessible from other tools [24]. As stated in [85, 136], exporting a diagram from one tool to another has not been accomplished yet with ease. MDE has to be supported by *compatible and standardized tools* [85, 136] to guarantee the information interchange and interoperability between tools.

Even though MDE can benefit from using generic techniques from other software engineering domains for solving consistency problems, specific features such as *interactivity, compatibility, and scalability* cannot yet be addressed by existing solutions from other software engineering domains because translating models from one form to another introduces more complexity than it removes [136], for instance. Therefore, various research approaches still need to be explored and evolved.

Goal 1: Developing techniques that construct consistent models of a given meta-model not only automatically but also interactively so that the modeler can control the construction process. This semi-automated model construction is promising for producing desired models. The techniques can construct a consistent model from a given model being potentially inconsistent. Moreover, the techniques should be meta-model agnostic, i.e., generic and can be instantiated to any given domain meta-model and deal with its instances. Furthermore, for a given domain (meta-model), meta-tools for automatically generating DSML tools for constructing (repairing and generating) models have to be developed to ease and accelerate the development process. The tools and the tools' outputs should be compatible and standard to support the interchangeability between tools. Additionally, the soundness should be presented, and the techniques should scale.

1.1.2 Developing Efficient Techniques for Ensuring the Consistency of Model Transformations

Applying a model transformation to a consistent model may yield an inconsistent result, and thus, errors may occur. Inconsistent results may affect the applicability of the transformations. Consequently, transformation-based automation becomes untrustworthy, error-prone, and even impossible. For example, when defining modeling languages and their operations (e.g., editing operations) on models, the operations must produce results that belong to the language. Moreover, DSML users may use the operations in a way (not thought by the developers), which leads to a transformation crash or create inconsistent models. Further, in a concurrent and distributed system, each state should fulfill some required invariants such as safety properties, and each refactoring should preserve the model consistency. Making sure that all constraints are considered (implemented) is a challenge and can be hard [145].

In [136, 145], several open challenges are identified dealing with the quality of the modeled system, namely with the consistency of model transformations such as: How to define modeling techniques that enable and enforce model correctness by construction? Newly, the international conference of model transformations (ICMT'19) [1] put forward several challenges such as: How to ensure that model transformations produce expected results? How to foster the trust in and use of model transformations?

Goal 2: For a given set of constraints, and a set of model transformation rules, our goal is to develop techniques to *automatically* enhance the behavior of transformation rules to respect the given set of constraints, i.e., the resulting model after applying a model transformation rule is consistent w.r.t. the set of constraints. Moreover, to foster trust in the use of model transformations, the soundness of the techniques has to be ensured, and the techniques should behave efficiently.

1.2 Contributions

In this section, we present four contributions that aim at tackling the previous challenges and thus achieving our goals.

First of all, our techniques are developed to consider the Eclipse Modeling Framework (EMF) and its constraints. Thus, the interchangeability and interoperability between

[1] http://www.model-transformation.org/

tools are guaranteed. Moreover, the developed approaches are rule-based. Therefore, they can be parameterized and configured to support user interactions.

1.2.1 Automated Interactive Techniques for Constructing Consistent Models of DSMLs

In the first part of the thesis, we present the first two contributions (out of four): The first one is for *repairing EMF models* to be consistent w.r.t. the meta-model of a given DSML in an automated interactive way. The second one is for *generating large and consistent EMF models* w.r.t the meta-model by supporting user configurations. Meta-tools are developed to automatically customize (instantiate) the approaches to any given domain meta-model. DSML tools are provided as ready-to-use tools for modelers. Additionally, the soundness and the scalability of both techniques are presented.

An overview of the first two contributions of the thesis is given in more detail as follow and illustrated in Figure 1.2:

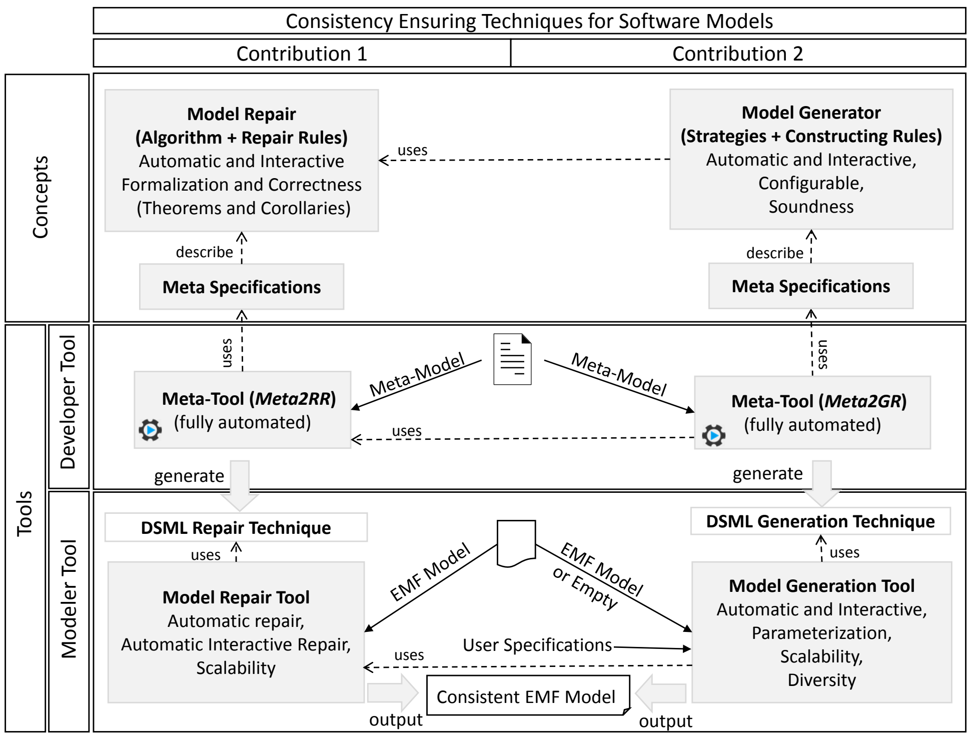

Consistency Ensuring Techniques for Software Models
Contribution 1
Contribution 2
Concepts
Model Repair (Algorithm + Repair Rules) Automatic and Interactive Formalization and Correctness (Theorems and Corollaries)
uses
Model Generator (Strategies + Constructing Rules) Automatic and Interactive, Configurable, Soundness
describe
Meta Specifications
describe
Meta Specifications
Tools
Developer Tool
uses
Meta-Model
Meta-Model
uses
Meta-Tool (Meta2RR) (fully automated)
uses
Meta-Tool (Meta2GR) (fully automated)
generate
generate
Modeler Tool
DSML Repair Technique
uses
DSML Generation Technique
uses
EMF Model
EMF Model or Empty
Model Repair Tool Automatic repair, Automatic Interactive Repair, Scalability
User Specifications
uses
Model Generation Tool Automatic and Interactive, Parameterization, Scalability, Diversity
Consistent EMF Model
output
output

Figure 1.2: Contributions 1 and 2 for ensuring the consistency of software models

- **Contribution 1–Rule-based Model Repair:** In this work, we propose a rule-based approach for repairing models. Given a model, a model repair modifies this model such that it becomes consistent concerning its given meta-model. If the model is already consistent, the model repair does nothing. Our approach distinguishes itself from other approaches that it can repair models not only fully automatically but also guide the modeler in repairing models in a *semi-automated (interactive)* way and thus resolving all the cardinalities violations. This *interactive* model repair is promising for producing desired models. Additionally, we consider the EMF constraints, i.e., the output models are consistent instance models of meta-models with multiplicities conforming to the Eclipse Modeling Framework (EMF).

A generic rule-based algorithm for model repair is developed to repair any instance model of any given domain meta-model, and different repair rules are specified to configure the algorithm w.r.t. the given domain. The repair rules are defined as transformation rules. Meta-specifications (describing how the rules are designed and under which conditions should be derived) are provided to derive the different kinds of repair rules and to instantiate the approach to the given DSML.

A meta-tool (developer tool), called *Meta2RR* is developed as an Eclipse plug-in. It takes a domain meta-model as an input, uses the meta-specifications, and generates the corresponding domain-specific model repair technique in a fully automated way. Once the domain-specific model repair technique is generated, there is no need to use the meta-tool again as long as the meta-model is not changed.

A model repair tool (DSML tool) is developed as an Eclipse plug-in and provided as a ready-to-use tool to repair instance models of the DSML in an automated or interactive manner. It takes an EMF model as an input, uses the generated domain-specific model repair technique, and returns a consistent EMF model.

The algorithm is formalized, and its correctness is presented. It is systematically tested concerning the correctness. Additionally, we discuss the approach in a more general setting and present its features.

Moreover, we show that the approach is scalable in repairing models up to 10 000 elements. The related work is discussed as well.

- **Contribution 2–Rule-based Model Generator:** With the steady use of model-driven engineering (MDE) in practice, the need for the automated generation of instance models grows. In particular, there is a growing need for *large instances* of a given meta-model [70, 118]. Moreover, *interactive* model generation supporting *user configurations* is promising for producing desired models. Furthermore, generating models by *manipulating existing models* not only supports the incremental development of models at different times but also is promising for accelerating the generation process. In this work, we cover all the previous characteristics.

  We present a rule-based, configurable approach for generating consistent EMF models, i.e., consistent instance models of meta-models with multiplicities conforming to the Eclipse Modeling Framework (EMF). The generation process can start at a seed EMF model being potentially inconsistent, and the approach is scalable. Since our approach is rule-based, the generation process can be configured w.r.t. user specifications. Several parameterization strategies are developed and presented to generate different sets of consistent EMF models.

A configurable, rule-based model generator utilizing the EMF model repair is developed for generating consistent EMF models. Meta-specifications are provided to derive the proper generation rules, to instantiate the approach to the given DSML, and to support user specifications.

A meta-tool, called *Meta2GR*, is developed as an Eclipse plug-in. It takes a meta-model as an input, uses the meta-specifications, and returns the corresponding domain-specific generation technique in a fully automated way. Once it is generated for a given DSML, there is no need to use the meta-tool again as long as the meta-model is not changed.

A model generator tool is developed as an Eclipse plug-in. It takes user specifications and EMF model (optional) as inputs, uses the generated domain-specific generation technique, and generates a consistent EMF model.

The soundness of our approach is discussed. After that, we evaluate its scalability by generating consistent EMF models with 10 000 elements (nodes and edges) in about a minute on average. Furthermore, we show that we can generate consistent EMF models with *half a million elements* efficiently. A scalability comparison with the state-of-the-art instance generator is provided showing that our generator is efficient for generating large and consistent EMF models of meta-models without OCL constraints. Moreover, we show the diversity of the generated models. The related work is discussed in detail as well.

1.2.2 Automated Consistency-by-Construction Techniques for Model Transformations

In the second part of the thesis, we present the other two contributions (the third and fourth one): The third contribution is a *correct-by-construction* tool, i.e., the tool functionalities are formally guided by specifications. The tool enhances the behavior of a model transformation rule to *guarantee* a set of OCL constraints *automatically*. In the last contribution, we present a *consistency-by-construction* technique that enhances and optimizes the behavior of model transformations to *preserve* the model consistency with regard to a set of constraints. The correctness of the contribution is presented. The performance of both implemented contributions and the complexity of the outputs are shown. The overall approach is assessed, as well.

An overview of the third and fourth contribution is given in more detail as follow and illustrated in Figure 1.3:

- **Contribution 3–Automated Construction of Guaranteeing Application Conditions from OCL Constraints:** In several application scenarios, model transformation rules are often supposed to have a well-defined behavior in the sense that their resulting models are consistent w.r.t. a set of constraints. In this work, we enhance the behavior of model transformation rules by *automatically constructing constraint-guaranteeing application conditions from OCL constraints.* A rule with a *constraint-guaranteeing* application condition applies successfully to a model if and only if the resulting model after the rule application satisfies the constraint. The theoretical foundations for translating OCL

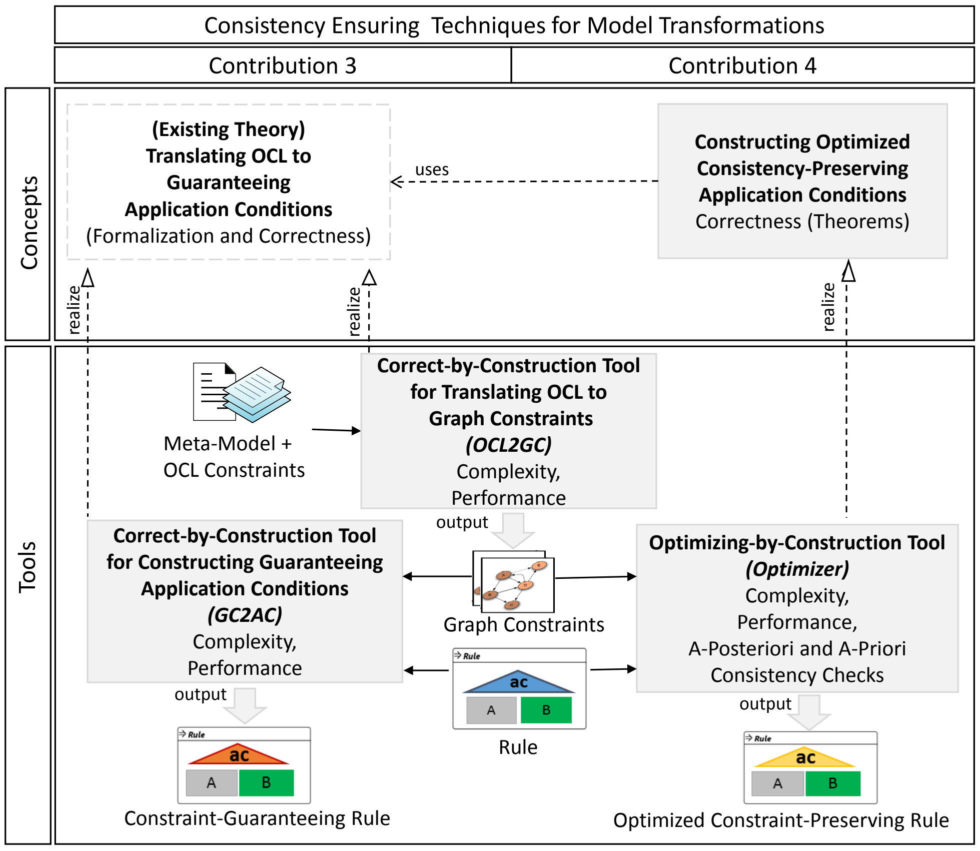

Consistency Ensuring Techniques for Model Transformations
Contribution 3
Contribution 4
Concepts
Tools
(Existing Theory) Translating OCL to Guaranteeing Application Conditions (Formalization and Correctness)
uses
Constructing Optimized Consistency-Preserving Application Conditions Correctness (Theorems)
realize
realize
realize
Meta-Model + OCL Constraints
Correct-by-Construction Tool for Translating OCL to Graph Constraints (OCL2GC) Complexity, Performance
output
Correct-by-Construction Tool for Constructing Guaranteeing Application Conditions (GC2AC) Complexity, Performance
Graph Constraints
Optimizing-by-Construction Tool (Optimizer) Complexity, Performance, A-Posteriori and A-Priori Consistency Checks
Rule
ac
A
B
output
output
Constraint-Guaranteeing Rule
Optimized Constraint-Preserving Rule

Figure 1.3: Contributions 3 and 4 for ensuring the consistency of model transformations

constraints to graph constraints, and for calculating the constraint-guaranteeing application condition do already exist, and their correctness is shown in [114].

However, *manually* enhancing a set of rules to guarantee a set of constraints is a tedious, time consuming, and error-prone task and requires high skills related to the theoretical foundation and the environment. Since we seek to develop reliable techniques, we have developed the solution design of realizing both theoretical foundations [114].

Two tool supports (as Eclipse plug-ins) are developed to *fully automate* the whole process: The first tool support, called *OCL2GC*, takes a meta-model and a set of OCL constraints as inputs and returns a set of semantically equivalent graph constraints as an output. The second tool support, called *GC2AC* takes a Henshin rule and a graph constraint as inputs and returns the Henshin rule with a *constraint-guaranteeing* application condition. Thus, the enhanced rule applies successfully to a *model* if and only if the resulting model after rule application satisfies the given constraint. To the best of our knowledge, we provide the first ready-to-use tool.

Furthermore, we show the feasibility and efficiency of the implemented theories in practice by carrying out experiments to measure the performance of the implemented approach and the complexity of the outputs for both components. The related work is discussed as well.

- **Contribution 4–Constructing Optimized Consistency-Preserving Application Conditions:** We present a new technique for *constructing an optimized constraint-preserving application condition* from a given set of constraints and a rule, and thus, rendering the model transformation rule consistency-preserving. A rule is consistency preserving if all its applications to consistent models yield consistent models, as well.

  Our technique consists of several optimization strategies that simplify guaranteeing application conditions during the construction process. This kind of automatic approximation is conceptually new and quite general in scope.

  We formally show the correctness of our approach. Moreover, the optimizer is implemented as an Eclipse-based tool automating the whole process. The optimizer takes a Henshin rule and a graph constraint as inputs and returns the Henshin rule with a *constraint-preserving* application condition. Thus, the enhanced rule applies successfully to a *consistent model* w.r.t. the constraint if and only if the resulting model after rule application is still consistent w.r.t. the given constraint.

  Furthermore, we experimentally compare the non-optimized application conditions with the optimized ones. The results show a considerable loss in complexity and better run-time performance. Additionally, we conduct a performance experiment comparing the *a-priori consistency* check approach using our techniques with *a-posterior consistency* check approach in general. An extension of the work is discussed, and the related work is presented as well.

1.3 Methodology

The research methodology guiding this thesis is represented by taxonomies of software engineering research proposed in [132, 96]. This work comprises *techniques* supported by *analytic models*. Techniques are defined as inventing new ways to do some tasks, including procedures and implementation techniques, whereas analytic models are defined as developing structural models that permit formal analysis. All the contributions provide techniques to generate DSML tools automatically or to perform the tasks automatically. They are provided as ready-to-use tools. All contributions provide new solutions (however, in contribution 3, we provide a design solution) and present their feasibility and efficiency. All contributions are based on algebraic graph transformations. In Contributions 1, 2, and 4, we give formal proofs or discuss their soundness.

To validate the research results, we use different kinds of validation techniques. Shaw [132] presents several kinds of validation techniques: persuasion, analysis, implementation, evaluation, and experience. Throughout the thesis, persuasion is used to motivate and explain our design choices. In Contributions 1, 2, and 4, we show the soundness. We provide implementations as ready-to-use tools for all contributions

(1-4). All contributions (1-4) are supported by empirical evaluations and experiments to measure its feasibility and efficiency, e.g., performance and complexity.

1.4 Publications

This thesis originates from several years of research which is supported by the German Research Foundation (DFG), Grants HA 2936/4-2 and TA294/13-2 (Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages).

In the following, we list the most important publications in the context of this thesis:

- Nebras Nassar, Hendrik Radke, Thorsten Arendt: Rule-based Repair of EMF Models: An Automated Interactive Approach. In: Theory and Practice of Model Transformation(ICMT 2017), Springer. [93].
- Nebras Nassar, Jens Kosiol and Hendrik Radke: Rule-based Repair of EMF Models: Formalization and Correctness Proof. In: Graph Computation Models (GCM 2017), Electronic. [92].
- Nebras Nassar, Jens Kosiol, Thorsten Arendt, Gabriele Taentzer: OCL2AC: Automatic Translation of OCL Constraints to Graph Constraints and Application Conditions for Transformation Rules. In: The International Conference on Graph Transformation (ICGT 2018), Springer. [89].
- Nebras Nassar, Jens Kosiol, Thorsten Arendt, Gabriele Taentzer: Constructing Optimized Validity-Preserving Application Conditions for Graph Transformation Rules. In: The International Conference on Graph Transformation (ICGT 2019), Springer. [90]. **(EASST Best Paper Award)**
- Nebras Nassar, Jens Kosiol, Timo Kehrer, Gabriele Taentzer: Generating Large EMF Models Efficiently: A Rule-Based, Configurable Approach. In: International Conference on Fundamental Approaches to Software Engineering (FASE 2020)(accepted). [88].
- Nebras Nassar, Jens Kosiol, Thorsten Arendt, Gabriele Taentzer: Constructing Weakest Constraint-Preserving Application Conditions for Model Transformation Rules. In: Journal of Logical and Algebraic Methods in Programming (JLAMP), 2020, (submitted). [91].

The following publications are relevant to the thesis but are not a part of it. In [87], we provide a generic approach and develop a meta-tool, called *NasTool*, to derive model metrics from a given meta-model automatically. The metrics can be used to inquire about the model elements and thus aim at finding the elements which violate a constraint such as the upper bound of a relation. In [71], we develop an approach that updates the rule actions to satisfy a specific form of graph constraints.

- Nebras Nassar, Thorsten Arendt, Gabriele Taentzer: Deriving Model Metrics from Meta Models. In: The Proceedings of Modellierung 2016, Karlsruhe, Germany. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik. [87].
- Jens Kosiol, Lars Fritsche, Nebras Nassar, Andy Schürr, Gabriele Taentzer: Towards Establishing Consistency between Graph Transformation Rules and Atomic Graph Constraints Using Multi-Amalgamation. In: The 24th international workshop on algebraic development techniques (WADT 2018). [71].

We have developed the following Eclipse-based software projects [2]:

- **OCL2AC Project**: It consists of three tool supports:
  1. **OCL2GC** translates OCL constraints into a set of semantically equivalent (nested) graph constraints.
  2. **GC2AC** integrates graph constraints as guaranteeing application conditions into transformation rules.
  3. **Optimizer** constructs optimized validity-preserving application conditions from graph constraints.
- **Model Generator Project:** Generating large EMF models of a given DSML by supporting user specifications.
- **Model Repair Project:** Repairing EMF models of a given DSML automatically or interactively.
- **Model Metrics Project:** Deriving model metrics from a given DSML.

1.5 Outline

The thesis comprises two main parts, and each part contains two chapters: (Part I) Consistency ensuring techniques for software models and (Part II) consistency ensuring techniques for model transformations. The remainder of this thesis is structured as follows:

- **Chapter 2** presents the basic background information needed throughout the thesis. EMF models, EMF constraints, graphs, transformation rules, and OCL are presented. They form the basis for both parts. Additionally, graph constraints and conditions are defined, OCL translation rules are outlined, and the calculation of application conditions is presented. All form the basis of the second part.

[2]https://www.uni-marburg.de/fb12/arbeitsgruppen/swt/nebras-nassar

Part I composes the following two chapters presenting consistency ensuring techniques for software models:

- **Chapter 3** introduces *rule-based model repair.* The work is introduced and motivated by example, the developed rule-based algorithm, the catalog of the different kinds of repair rules and their meta-patterns are outlined, the approach is formalized, presented as Hoare triple and as graph automaton, and the correctness proof of the algorithm and its features are shown, the ready-to-use tool with its functionaries is outlined and systematically tested, the scalability is shown, the discussion of the approach in general settings (e.g., supporting a set of OCL constraints) is provided, and the related work is discussed as well.

- **Chapter 4** presents *rule-based model generator.* The work is motivated and the approach of generating consistent EMF models is presented, four particularization strategies and their meta-specifications are presented using examples, a ready-to-use tool is developed to generate consistent models starting with or without seed models, and the scalability experiment for generating consistent models of varying size up to half a million elements elements are presented using 8 modeling languages taken from different domains, a comparison with the state-of-the-art solver-based tool is carried out, and diversity experiments for generating diverse sets of consistent models are provided, the state-of-the-art of the related work is explored and classified.

Part II composes the following two chapters presenting consistency ensuring techniques for model transformations:

- **Chapter 5** presents *an automated construction of guaranteeing application conditions from OCL constraints.* The work is introduced and motivated by example, the solution design of realizing and automating the theoretical foundations is presented, the ready-to-use tool and its functionalities are shown, the restriction of the technique is studied by considering the OCL constraints of UML, the complexity and the performance of both tool supports are presented using a realistic application case, and the related work is discussed.

- **Chapter 6** presents *an automated construction of optimized consistency-preserving application conditions.* The approach for constructing optimized consistency-preserving application conditions is introduced by example, two theorems are formalized to show the correctness of our approach, the tool is implemented and discussed, the complexity and the performance of our approach are studied using a realistic application case, namely the MagicDraw Statechart meta-model with 11 OCL constraints and 84 editing rules, a performance comparison of *a-priori* consistency approach implemented using our techniques with *a-posteriori* consistency approach realized using validators are carried out, and the related work is presented as well.

- **Chapter 7** summarizes this thesis and gives an outline on possible future work.
- **Chapters A and B** provide comprehensive details used in this thesis.

2 Background

In this chapter, we introduce the main required concepts for the subsequent chapters. We first introduce the basic foundation needed to understand the first two contributions and then present the ones additionally needed to follow the last two contributions. Further helpful information is provided within the chapters to ease the readability.

A suitable formal framework and foundation for this work is the theory of algebraic graph transformation introduced by Ehrig et al. [39]. To use the theory of algebraic graph transformation, we consider EMF models and meta-models as graphs. A meta-model can be regarded as a graph with additional structural information like containment, inheritance, and opposite edges. Besides, a meta-model may contain multiplicities and further constraints. When formalizing meta-modeling by this means, graphs occur at two levels: the type level (meta-models) and the instance level (models). This is reflected by the concept of *typed graphs*, where a fixed *graph TG* serves as an abstract representation of the type structure of a meta-model. Correct typing of instances is formalized by structure-preserving mappings of instance graphs to type graphs.

In the following, we present EMF models, EMF constraints, and model transformations (Henshin), and introduce their formalization. This formalization has been established in Biermann et al. [14]. Then, we introduce a short introduction to the Object Constraint Language (OCL).

After that, we present the notions from the theory of algebraic graph transformation needed to understand the second part of our work additionally: We recall *nested graph constraints* and *conditions* as a means to express properties of graphs and of morphisms in between, we roughly sketch the translation of OCL to nested graph constraints. The formal translation of OCL constraints to graph constraints that we are using is presented in Radke et al. [114]. Since many OCL translation rules are defined w.r.t the OCL expressions, we present here the generic ones. Then, we recall the calculus of an application condition of a rule as presented in Habel and

Pennemann [55]. For more comprehensive details, please consider the works presented in [55, 114] since they mainly form the theoretical background of the second part.

2.1 EMF Models, Constraints and Hierarchy

The Eclipse Modeling Framework (EMF) has evolved into a de-facto standard technology for defining models and modeling languages [135]. It facilities the modeling process by providing tool supports for editing, viewing, and code generation as well as for building tool supports (e.g., Eclipse plug-ins).

The Eclipse Modeling Framework provides an Ecore meta-metamodel to specify domains (meta-models). The structure of the Ecore meta-metamodel is similar to the structure of the UML class diagram. It supports the classification of objects and their attributes, the relationships between the objects and constraints on those objects. In the following, we describe the main subset of the Ecore meta-metamodel for representing meta-models as shown in Figure 2.1:

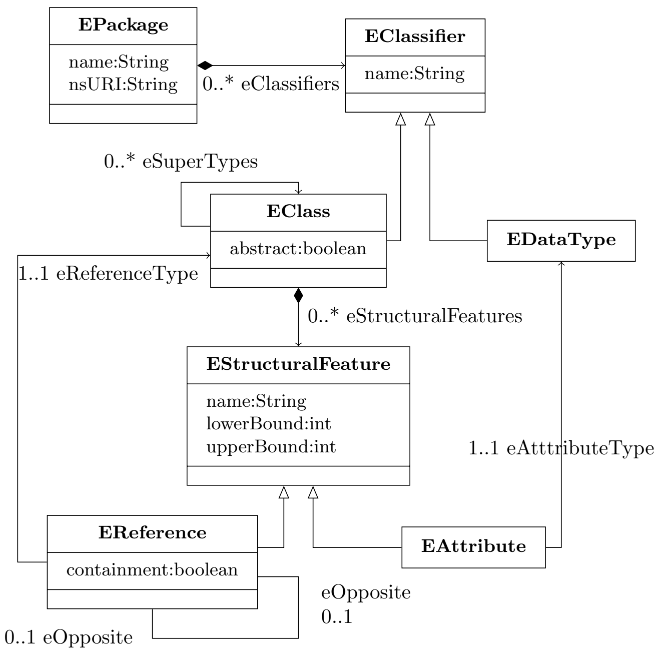


Figure 2.1: Excerpt of the Ecore meta-metamodel

- EClass: This type models classes which are represented as nodes of a graph. A class has a name and can contain attributes and references. A class can be *abstract* or *interface*, i.e., an instance of it cannot be created. A class can refer to other classes as its super-types to represent an inheritance relationship. An inheritance relation is depicted as an arrow with an empty triangle from a class called a subclass to another class called a superclass.

- EReference: It models an association between two classes (nodes). An association has a name and a type. Multiplicities are specified as lower and upper bounds at the end of the reference. EReference can specify three kinds of associations:
  - non-containment: It is a direct association between two classes (nodes), depicted as a regular arrow, which allows navigating from the source node to the target node.
  - containment: It is a stronger type of association representing an ownership relation between classes (nodes) and depicted as a regular arrow with a black diamond as a tail. It represents a part-of relationship between a container class representing the whole and a contained class representing the part. If an instance of a container class is deleted, all its contents have to be eliminated as well.
  - opposite: It is a bidirectional association specified by pairing an association with its opposite and depicted as an edge without arrowheads. Instantiating an opposite association means an instance of each paired association must exist together.
- EAttribute: It models attributes for classes. An attribute has a name and a type. Note, once a node is created in an EMF instance model, the node attributes with their default values are created as well.
- EDataType: It models simple types that are primitive or object types defined in Java, and they are most commonly used as attribute types.

A comprehensive description of the Ecore can be found at [34]. An EMF model unifies three important technologies: Java, XML, and UML, i.e., an EMF model can be created from a UML model, an XML scheme, or annotated Java interfaces. EMF can read and write EMF models in a format that conforms to the standard XML Metadata Interchange (XMI) serialization of the Essential Meta Object Facility (EMOF).

2.1.1 EMF Constraints

The Eclipse Modeling Framework provides several constraints to constitute, manipulate, and persist valid (consistent) state for objects modeled in EMF. The constraints needed *in practice* [14] are listed as follows:

- At-most-one-container: Each model node must not have more than one container.
- No-containment-cycles: Cycles of containment edges must not occur.
- No-parallel-edges: There are no two edges of the same type from the same source to the same target node.

- All-opposite-edges: If edge types $t1$ and $t2$ are opposite to each other: For each edge of type $t1$, there has to be an edge of type $t2$ linking the same nodes in the opposite direction.
- No-lower-bound-violation: For each edge type e, the number of target model nodes being connected to a source node via edges of type e is equal or higher than its lower bound.
- No-upper-bound-violation: For each edge type e, the number of target model nodes being connected to a source node via edges of type e is equal or smaller than its upper bound.
- Rootedness: There is a node, called the root node, such that all other model nodes are directly or transitively contained in it.
- Dangling condition: All adjacent edges of a node to be deleted have to be deleted as well.

2.1.2 EMF Modeling Hierarchy

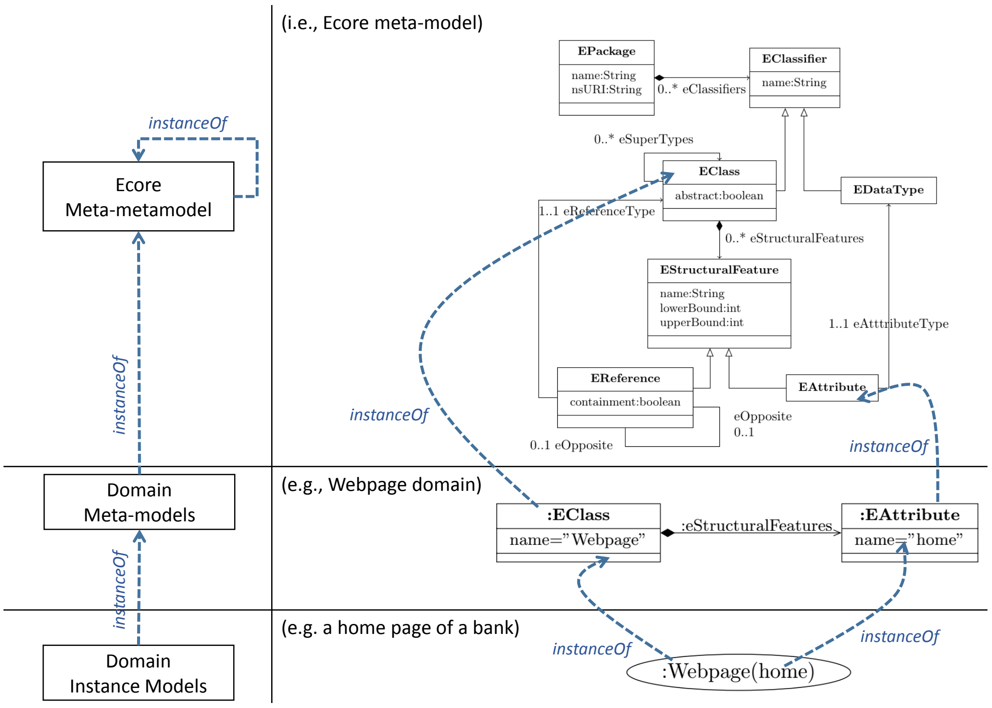


Figure 2.2: EMF modeling hierarchy: An example

The modeling hierarchy in EMF is categorized into meta-layers representing models for describing a system w.r.t different abstraction levels. Figure 2.2 illustrates and describes the layers using an example as follows:

- Meta-metamodel: It is the highest level of abstraction represented by Ecore. The Ecore meta-metamodel is also called the Ecore meta-model because it can describe itself.
- Meta-model: It describes a particular domain, e.g., a webpage meta-model describing a general structure of web pages. Ecore describes the webpage meta-model, i.e., the webpage meta-model is an instance of the Ecore meta-metamodel.
- Instance model: It represents a concrete instance, e.g., an instance model describing the home web page of a bank or a university, or the address web page of a hospital.

Similar examples are already mentioned in several works of literature, e.g., in [16]. Briefly, the main idea is that one can define meta-models addressing different domains such as a web page domain by instantiating the Ecore meta-metamodel. Likewise, by instantiating the domain meta-model, one can specify instance models describing concrete aspects of the domain, e.g., specifying web pages for a bank or a university.

2.2 Graphs and EMF Graphs

This section recalls the background information needed to design consistent EMF graphs and to change them in a consistency-preserving way. It is based on [14].

Definition 1 (Graph and graph morphism). A *graph* $G = (V_G, E_G, s_G, t_G)$ consists of a set V_G of *nodes* (or *vertices*), a set E_G of *edges*, and *source* and *target functions* $s_G, t_G : E_G \to V_G$.

For graphs G and H, a *graph morphism* $f = (f_V, f_E)$ is a pair of functions $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ such that $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$. If f_V and f_E are inclusions, then G is a subgraph of H, short $G \subseteq H$.

Definition 2 (Type graph). A type graph $TG = (T, I, A, C, OE, mult)$ consists of a graph $T = (V_T, E_T, s_T, t_T)$, an *inheritance relation* $I \subseteq V_T \times V_T$, a set $A \subseteq V_T$ of abstract nodes, a set $C \subseteq E_T$ of containment edges, a relation $OE \subseteq E_T \times E_T$ of opposite edges, and a function $mult : E_T \to \mathcal{N} \times (\mathcal{N} \cup \{*\})$ subject to the following conditions: The inheritance relation I is a partial order. The relation OE obeys the axioms: (i) for all $(e_1, e_2) \in OE : s_T(e_1) = t_T(e_2)$ and $s_T(e_2) = t_T(e_1)$ (opposite directions), (ii) for all $e \in E_T : (e, e) \notin OE$ (anti-reflexivity), (iii) for all $(e_1, e_2) \in OE : (e_2, e_1) \in OE$ (symmetry), and (iv) for all $(e_1, e_2), (e_1, e_3) \in OE : e_2 = e_3$ (functionality). For each $(i, j) \in mult(E_T)$ it holds that $i \leqslant j$ and $j \neq 0$ (where $\leqslant$ is the standard order of natural numbers supplemented by $i < *$ for all $i \in \mathcal{N}$).

The following concepts and notations will be used throughout the work and further clarify the correspondence between type graphs and EMF models: For $e \in OE$ we denote its opposite edge by e^{-1}. The inheritance clan of a node $n \in V_T$ is defined by $clan_I(n) = \{m \mid (m,n) \in I\}$. We write $m \leqslant n$ for $m \in clan_I(n)$. We use the set of containment edges C to define a containment relation $cont_{TG} \subseteq V_T \times V_T$, which is given as the transitive closure of the relation $\{(n,m) \mid \exists c \in C : n \leqslant s(c) \wedge \ m \leqslant t(c)\}$ and write $n \sqsupseteq m$ for $(n,m) \in cont_{TG}$. In the following, by $NC := E_T \backslash C$ we refer to the non-containment edges. The function $mult$ maps each edge of E_T to a multiplicity. A value of $*$ indicates an unconstrained number of edges. For $mult(e) = (i,j)$, i is called the lower bound, j is called the upper bound of e, and we define functions $low : E_T \to \mathcal{N}, upp : E_T \to \mathcal{N} \cup *$ which assign the respective bounds to each edge.

Structure preserving maps establish the relationship between TG and instances:

Definition 3 (Typed graphs, typed morphisms). A graph G is called typed over TG if there exists a graph morphism $type_G = (type_{V_G}, type_{E_G}) : G \to TG$ s.t. $type_{V_G}(s_G(e)) \leqslant s_T(type_{E_G}(e))$ and $type_{V_G}(t_G(e)) \leqslant t_T(type_{E_G}(e))$ for all $e \in E_G$; $type_G$ is called typing morphism. A typed graph morphism between graphs G and H, which are both typed over TG, is a graph morphism $f = (f_V, f_E) : G \to H$ such that $type_{V_H} \circ f_V(n) \leqslant type_{V_G}(n)$ for all $n \in V_G$ and $type_{E_H} \circ f_E = type_{E_G}$.

The typing of a graph G over TG with $type_G$ induces a containment relation on G. We write $C_G := \{e \in E_G \mid type_{E_G}(e) \in C\}$ and $cont_G$ is the transitive closure of the relation $\{(s_G(c), t_G(c)) \mid c \in C_G\}$. As above, we denote $(n,m) \in cont_G$ by writing $n \sqsupseteq m$. Analogously, we introduce the sets NC_G and OE_G.

Definition 4 (EMF constraints). To define typed graphs, which also respect the constraints of EMF and to later present our repair and generation algorithms, we now give a list of certain properties typed graphs can possess and express these properties as logical formulas. Let G be a graph typed over TG by typing morphism $type_G$:

- At-most-one-container: No node of G has more than one container:
$$\forall c_1, c_2 \in C_G . t_G(c_1) = t_G(c_2) \Rightarrow c_1 = c_2 \ .$$
- No-containment-cycle: No cycles of containment edges occur in G:
$$\forall n \in V_G . n \not\sqsupseteq n \ .$$
- No-parallel-edges: No parallel edges exist, i.e., there are no two edges of the same type from the same source to the same target node:
$$\forall e, e' \in E_G \Big(\big(type_{E_G}(e) = type_{E_G}(e') \\ \wedge s_G(e) = s_G(e') \ \wedge \ t_G(e) = t_G(e') \big) \Rightarrow e = e' \Big) \ .$$

- All-opposite-edges: Each edge of opposite edge type has its inverse edge:

$$\forall e \in OE_G . \exists e' \in OE_G \Big(s_G(e) = t_G(e') \\ \wedge t_G(e) = s_G(e') \wedge type_{E_G}(e) = type_{E_G}(e')^{-1} \Big) \ .$$

- Concreteness: It instantiates no node of abstract type:

$$\forall n \in V_G . type_{V_G}(n) \notin A \ .$$

- Rootedness: There exists a node r in V_G, called *root node*, such that all nodes of G, except for r, are transitively contained in r:

$$\forall n \in V_G \big(n \neq r \Rightarrow r \sqsupseteq n \big) \ .$$

- No-bound-violation: For every node $n \in V_G$ and every edge type $e_T \in E_T$, for which n can serve as source node, the number of outgoing edges of type e_T from n is between e_T's lower and upper bound:

$$\forall n \in V_G . \forall e_T \in E_T \big(type_{V_G}(n) \leqslant s_T(e_T) \Rightarrow \\ low(e_T) \leqslant \#\{ e \in E_G \,|\, type_{E_G}(e) = e_T \wedge s_G(e) = n \} \leqslant upp(e_T) \big) \ .$$

Definition 5 (EMF model graph). A graph G typed over $TG = (T, I, A, C, OE, mult)$ with typing morphism $type_G$ is an EMF-model graph over TG (or w.r.t. TG) if the conditions At-most-one-container, No-containment-cycle, No-parallel-edges, and All-opposite-edges hold.

2.3 Model Transformations: Henshin

When formalizing transformations of EMF models [34] by the means of the theory of algebraic graph transformation [39, 14], they are specified by transformation rules.

Definition 6 (Transformation rule). A rule $\rho = (p, lac, rac)$ consists of a plain rule p and left and right application conditions lac and rac. The plain rule p consists of three graphs L, K, and R, called left-hand side (LHS), interface, and right-hand side (RHS) with two inclusion morphisms $l : K \hookrightarrow L, r : K \hookrightarrow R$. p is *monotonic* if $l : K \hookrightarrow L$ is an isomorphism. p only deletes if $r : K \hookrightarrow R$ is an isomorphism. The application conditions lac and rac are graph conditions over L and R, respectively.

Given a rule $\rho = ((L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R), lac, rac)$ and an injective morphism $m : L \hookrightarrow G$ with $m \models lac$, called *match*, a *(direct) transformation* $G \Rightarrow_{\rho,m} H$ from G to H via ρ at match m is given by the diagram to the right where both squares are pushouts.

$$\begin{tikzcd}
L \arrow[d, hook, "m \models lac"'] & K \arrow[l, hook', "l"] \arrow[r, hook, "r"'] \arrow[d, hook] & R \arrow[d, hook, "n \models rac"] \\
G & D \arrow[l, hook'] \arrow[r, hook] & H
\end{tikzcd}$$

A rule p is *applicable* at match m if the first pushout square above exists, i.e., if $m \circ l$ has a pushout complement D, and, moreover, the match morphism m satisfies lac and the co-match n satisfies rac.

Note that the first pushout square exists iff the match m fulfills the dangling edge check ensuring that a rule application at this match would not let an edge dangle. Applying the rule, the elements of $m(L \backslash K)$ are deleted. Then, at the chosen image of K in G, a copy of $R \backslash K$ is created. Afterwards, the resulting mapping of the graph R into the new graph is checked to fulfill the right application condition of the rule. In that case, the new graph is the result of the rule application.

Henshin Transformation Language

In our work, we use Henshin [3] due to its formal background, which can be advantageously used to argue for correctness. Henshin is an in-place model transformation language based on graph transformation concepts. In Henshin, transformations of EMF models can be defined by transformation rules. Each rule consists of a left-hand side (LHS), a right-hand side (RHS), and a partial mapping between these object structures specifying which elements are deleted, created or just preserved. Besides, there may be positive application conditions (PACs) or negative application conditions (NACs) requiring or forbidding certain model patterns. The left-hand side of a rule and its application conditions formulate the structural preconditions that must be fulfilled for applying the rule. Accordingly, its right-hand side describes the result (or postconditions). In Henshin specifications, several rule actions (stereotypes), namely, preserve, delete, create, forbid and require are specified to distinguish among elements that should be matched, deleted, created, forbade and required, respectively. Moreover, a Henshin rule may have an arbitrary number of parameters which control the rule application w.r.t. the variable data.

Furthermore, several kinds of transformation units are specified to control the order of rule applications. A transformation unit may compose sub-units and rules. There are different kinds of units, such as:

- Independent unit: It comprises an arbitrary number of sub-units that are checked in a non-deterministic order for applicability. It executes one applicable unit.
- Loop unit: It comprises one sub-unit and executes this as often as possible.
- Conditional unit: It comprises either two or three sub-units specifying the if-unit, the then-unit, and the else-unit. If the if-unit is executed successfully, the then-unit is executed. Otherwise the else-unit is executed.

- Sequential unit: It comprises an arbitrary number of sub-units that are executed in the given order. If a sub-unit is not applicable, it is skipped.

- Priority unit: It comprises an arbitrary number of sub-units that are checked for applicability. It tries to execute the first sub-unit. If this failed, it executes the next one and so on. If a sub-unit is executed successfully, the check and execution of the following sub-units are skipped.

2.4 Object Constraint Language (OCL)

The Object Constraint Language (OCL) [99] is a constraint language used to supplement the specification of object-oriented models. It is equipped with different kinds of models being consistent with the Meta Object Facility (MOF) [103], UML [100], or EMF. OCL can be used to specify invariants, operation contracts, guards of state transitions, or queries on object structures. OCL is a typed language, i.e., each OCL expression evaluates to a type.

The OCL type system supports user-defined types (classes and enumerations), primitive data types (Integer, Real, String and Boolean), and two template types (Tuple and Collection). The type Collection has subtypes Set, Bag, and Sequence to support specific kinds of collections. OCL provides logical operators such as and, or, not and implies, and a universal quantifier forall and an existential quantifier exists. OCL supports navigation along references to objects and attribute values using the *dot notation*. Navigation results may be single-valued, i.e., an object or data value, or Collections.

All predefined types come along with operations such as collection operations union, size, and select. The operations are called by the *arrow notation*. An important sublanguage of OCL is EssentialOCL which is “the minimal OCL required to work with EMOF” [99]. In this work, we deal with OCL invariants (constraints) that must hold for the system being modeled. An OCL invariant is specified in a particular type called the context type. The body of an invariant is a boolean expression (condition). The evaluation of an invariant returns true if all instances of the context type satisfy the body expression.

2.5 Graph Constraints and Conditions

Nested graph constraints formulate properties of graphs whereas *nested graph conditions* express properties of graph morphisms [55], i.e., type and structure-preserving mappings between graphs. Graph conditions are mainly used to restrict the applicability of rules. Constraints and conditions are defined recursively as trees of injective morphisms.

Definition 7 (Graph condition and constraint). Given a graph P, a *(nested) graph condition* over P is defined recursively as follows: `true` is a graph condition over P and $\exists\,(a : P \hookrightarrow C, c)$, where a is an injective morphism and c is a graph condition over C, is one, again. Moreover, Boolean combinations of graph conditions over P are graph conditions over P. A *(nested) graph constraint* is a condition over the empty graph $\varnothing$.

Satisfaction of a graph condition d over P for a morphism $p : P \to G$, denoted as $p \models d$, is defined as follows: Every morphism satisfies `true`. The morphism p satisfies a condition of the form $d = \exists\,(a : P \hookrightarrow C, c)$ if there exists an injective morphism $q : C \hookrightarrow G$ such that $p = q \circ a$ and q satisfies c. For Boolean operators, satisfaction is defined as usual. A graph G satisfies a graph constraint d, denoted as $G \models d$, if the empty morphism to G does so.

$$\begin{tikzcd} \exists\ P \arrow[rr, "a"] \arrow[dr, "p"'] & = & C \arrow[dl, "q"] \triangleleft c \\ & G & \end{tikzcd}$$

Graph constraints are expressively equivalent to a first-order logic on graphs [55, 115]. To ease notation, we use the *compact* form, i.e., we drop the domain of morphisms in constraints and conditions whenever they may be unambiguously inferred and indicate the mapping by the names of nodes. We call constraints of the form $\exists\,C$ *positive* and of the form $\neg\exists\,C$ *negative constraints.*

2.6 OCL Translation Rules

The translation of EssentialOCL invariants to nested constraints is formally presented in [112]. The correctness and completeness of the translation approach are proven. Since a lot of OCL translation rules are specified concerning different OCL expressions, we present here their classifications according to the kind of the OCL expressions as follows:

- Invariant translation (tr_I): Translation for OCL invariants.
- Expression translation (tr_E): Translation for OCL expressions yielding a Boolean value (tr_E).
- Navigation translation (tr_N): Translation for OCL expressions yielding single objects.
- Set translation (tr_S): Translation for OCL expressions yielding collections of objects.

For each translation kind, several concrete translations are specified. For examples, the expression translation (tr_E) composes translations for OCL expressions such as exists(exp), forall(exp), includesAll(exp), excludesAll(exp), notEmpty(), comparison operator $\{<, >, \leqslant, \geqslant, =, <>\}$, oclIsKindOf(T) and OCLIsTypeOf(T) and the set translation (tr_S) includes translations for OCL expressions such as select(exp), union(exp), reject(exp), intersect(exp), collect(exp) and allInstances().

Table 2.1: Excerpt of OCL translation where $T(\dots)$ indicates a recursive call of the translation

OCL constraint (snippet)	**Graph pattern (snippet)**
`context T inv:`	$\forall\ \boxed{\text{self:T}}$
`v.b` $\langle op \rangle$ `c`	$\boxed{\begin{array}{c} \text{v:T} \\ \hline \text{b } \langle \text{op} \rangle \text{ c} \end{array}}$
`v.role`	$\exists\ \boxed{\text{v:T}} \xrightarrow{\text{role}} \boxed{\text{v':T'}}$
$\langle nav \rangle$`->exists(v | `$\langle expr \rangle$`)`	$\exists\ (\boxed{\text{v:T}}, T(\langle nav \rangle)\ \wedge\ T(\langle expr \rangle))$
$\langle nav \rangle$`->forall(v | `$\langle expr \rangle$`)`	$\forall\ (\boxed{\text{v:T}}, T(\langle nav \rangle)\ \Rightarrow\ T(\langle expr \rangle))$
$\langle nav1 \rangle$`->union(`$\langle nav2 \rangle$`)`	$T(\langle nav1 \rangle)\ \vee\ T(\langle nav2 \rangle)$
$\langle nav \rangle$`->size() >= n`	$\exists\ (\boxed{\text{v}_1\text{:T}}\ \dots\ \boxed{\text{v}_n\text{:T}}, \bigwedge_{i=1}^{n} T(\langle nav(\text{v}_i) \rangle))$

Table 2.1 points out the translation of the main constructs. The translation process works recursively along with the abstract syntax structure of OCL expressions. Classes and their attributes are translated to nodes of their respective types, the navigation expressions to edges. Logical operators are translated to their counterparts. Sets are the only kind of collections this translation process can handle. The most important idea for translating set expressions is as follows: A single node can be matched by any object of a set; by further nesting of the condition, this matching can be restricted in such a way that exactly the nodes of the defined set do match: Set operations are translated with their characteristic functions in mind. The whole specification of the translations can be found in Radke [112].

2.7 The Calculus of Application Conditions

Given a transformation rule r and a graph constraint c, a constraint-guaranteeing application condition of this rule with respect to the constraint always exists and can be calculated explicitly [55].

The calculation is done in two steps: First, c is *integrated* into r as *right application condition* in such a way that r is applicable if and only if the result satisfies the constraint. This procedure is called *shifting* (as it is formalized as a shift of the constraint along a morphism). The main idea is to consider all possible ways in which

c could be satisfied after the rule application. This is done by overlapping the elements of the RHS of the rule r with each graph of c in every possible way. This overlapping is done iteratively along with the nesting structure of c. Since for each graph of c there may exist several different ways to overlap with the RHS of the rule, the resulting right application condition *rac* can consist of considerably more graphs. However, the structure of c is preserved. The result of this calculation is yet impractical as one would need to first apply the rule r and then check the right application condition to be fulfilled. Therefore, the second step is needed in which we construct an equivalent left application condition from the originated right application condition *rac*. The idea behind this construction is to apply the rule reversely to the constraint c, again along with the nesting structure. If the inverse rule of r is applicable, the resulting condition is the new left application condition *ac*. If not, r gets equipped with the application condition `false`, as it is not possible to apply r at any instance without violating c. The rule r with its new application condition *ac* has the property that it is applicable at a chosen match if and only if the resulting instance fulfills the integrated constraint c (and the rule without application condition was applicable at all). It is interesting to note that this procedure calculates a *graph condition*, i.e., a property that each *rule match* has to satisfy, in contrast to a *graph constraint*, i.e., a property an *instance graph* has to satisfy. Moreover, this procedure calculates a so-called *c-guaranteeing* application condition, i.e., an application condition that ensures that the original constraint is satisfied after rule application. However, the information that the rule will only be applied in situations where the constraint c already holds, i.e., where the constraint only needs to be preserved and not guaranteed, frequently allows for way simpler conditions. Adding the premise that the constraint was already valid before rule application yields the *preserving application condition.* We elaborate on this in Chapter 6.

Fact 1 (Habel and Pennemann [55]). Given a plain rule $p = (L \hookleftarrow K \hookrightarrow R)$ and a graph constraint c there are constructions $\mathsf{Gua}(p, c)$ and $\mathsf{Pres}(p, c)$ equipping p with an application condition *ac* such that $H \models c$ for every transformation $G \Rightarrow_{\mathsf{Gua}(p,c)} H$ and $H \models c$ for every transformation $G \Rightarrow_{\mathsf{Pres}(p,c)} H$ where $G \models c$.

Part I

Consistency Ensuring Techniques for Software Models

Models are usually developed incrementally, from different perspectives, and at different times. Thus, inconsistencies can easily occur due to misunderstandings and lack of information. For example, edit operations may temporarily create an inconsistent model that must eventually be handled. Automatically resolving inconsistencies and additionally allowing user interactions during the repair process is promising to increase productivity and improve the quality of software development.

Moreover, constructing a large and diverse set of models are needed in various application scenarios like testing and validating the suitability of MDE technologies to deal with large input models.

In this part, we present two works: model repair (see Chapter 3) and model generation (see Chapter 4). Our developed techniques construct consistent models of a given meta-model not only automatically but also interactively. The construction can also start at a seed model being potentially inconsistent. The techniques are meta-model agnostic, i.e., they can deal with any domain meta-model and its instances. We present meta-specifications to specialize them in a given DSML and develop meta-tools (*Meta2RR* and *Meta2GR*) to build the corresponding DSML tools (model repair and generation) automatically. They are designed to be compatible with the Eclipse Modeling Framework (EMF), which is an implementation of the OMG standard specification in practice.

We present and discuss the soundness of our techniques, develop them as ready-to-use tools, show their scalability and characteristics, and discuss the related work as well.

3 Rule-based Repair of EMF Models

This chapter shares material with the ICMT'17 paper "Rule-Based Repair of EMF Models: An Automated Interactive Approach" Nassar et al.[93] and the GCM'17 paper "Rule-based Repair of EMF Models: Formalization and Correctness Proof" Nassar et al.[92].

Managing and resolving inconsistencies in models is crucial in model-driven engineering (MDE). One form of inconsistency resolution is model repair. In this work, we consider models that are based on the Eclipse Modeling Framework (EMF). We propose a rule-based approach to support the modeler in automatically trimming and completing EMF models and thereby resolving their cardinality violations. Although being under repair, the model may be viewed and changed interactively during this repair process. Different kinds of repair rules are designed. Based on the theory of graph transformation, we use transformation rules to declare different repair actions executed during the particular steps of the algorithm. This formal background is used to reason for the correctness of the algorithm and to present conditions under which it always terminates. Possible adaptions of and more general use cases for the algorithm such as supporting a set of OCL constraints are discussed. A meta-tool (*Meta2RR*) for automatically customizing the approach to a given meta-model and a DSML tool (*EMF Model Repair*) for repairing EMF models automatically or interactively are developed based on EMF and the model transformation language Henshin. Additionally, the approach is systematically tested and its scalability of repairing a large number of violations is reported.

3.1 Introduction

Model-driven engineering (MDE) has become increasingly popular in various engineering disciplines and has been accepted as a best practice in many application domains such as automotive and aerospace domains [77]. Domain-specific modeling languages

(DSMLs) promise to increase the productivity and quality of software developments. In MDE, models are the primary artifacts, whereas model transformations are the key operations for model manipulation.

Although model editors are mostly adapted to their underlying DSML, they usually allow editing inconsistent models. The violation of lower and upper bounds and further constraints requiring the existence of model patterns is usually tolerated during editing. This means that model editors often use a relaxed meta-model with fewer constraints than the original language meta-model [66]. The result of an editing process may be an inconsistent model that has to be repaired.

There are a number of model repair approaches in the literature which, however, are either purely interactive or fully automated. Syntactic and rule-based approaches such as [95, 94, 38] usually offer repair rules or repair plans that may be interactively selected. It is up to the user to find a way to a consistent model if there is any. On the contrary, search-based approaches as e.g., [129, 2, 59, 78] run fully automatically yielding consistent models if possible. The resulting model of a search-based repair process, however, might not always be the desired one.

We intend to integrate automatic model repair as seamlessly as possible into the editing process allowing user interaction during the repair. Our approach does not leave the resolution strategy completely to the modeler as in pure rule-based approaches. Instead, it is an automatic and semi-interactive approach that guides the modeler to repair the whole model and thereby yields a meaningful, consistent model. Our approach is designed to repairs models in two modes: (a) *randomly*, i.e., fully automatically without user interactions, or (b) *semi-automatically*, i.e., guiding the modeler to repair the whole model by resolving all the cardinalities violations and thus all inconsistencies are resolved. The modeler can make decisions from the suggested repair options (if possible) during the repair process. Furthermore, the repair process can be stopped and the resulting intermediate model is an EMF model which can be, e.g., displayed by the model editor.

In this work, we consider modeling languages being defined by meta-models based on the Eclipse Modeling Framework (EMF). EMF has evolved to a de-facto standard technology for defining models and modeling languages [135]. We present an approach to model repair that consists of two main tasks, as shown in Figure 3.1:

1. The meta-model of a given language is translated into a rule-based model transformation system containing repair rules that specify different repair actions. These derived repair actions are used to configure a rule-based algorithm developed for repairing models. The domain developer is responsible for performing this task.

2. A potentially inconsistent model is fully repaired in an automated and interactive manner using the generated transformation system which configures the model repair tool according to the given language. An ordered list of rules' applications being used to repair the model is reported as a second output. This task is performed by the modeler using the model repair tool.

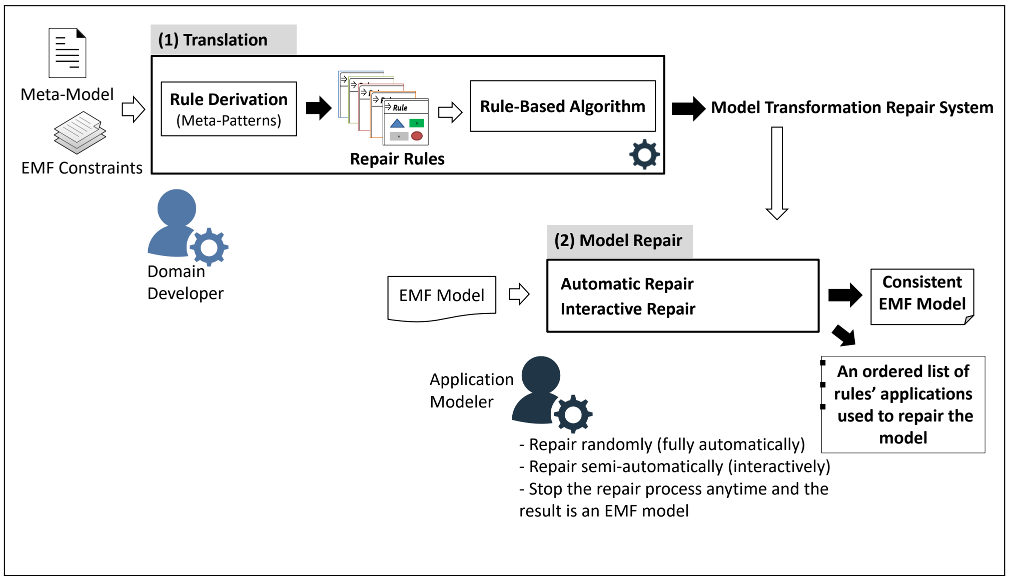

(1) Translation
Meta-Model
EMF Constraints
Rule Derivation
(Meta-Patterns)
Repair Rules
Rule-Based Algorithm
Model Transformation Repair System
Domain Developer
(2) Model Repair
EMF Model
Automatic Repair
Interactive Repair
Consistent EMF Model
An ordered list of rules' applications used to repair the model
Application Modeler
- Repair randomly (fully automatically)
- Repair semi-automatically (interactively)
- Stop the repair process anytime and the result is an EMF model

Figure 3.1: Our approach for model repair

The contributions of this work are the following:

1. A rule-based algorithm for model repair and a catalog of different repair rules.
2. Meta-specifications for deriving the proper rules from any given meta-model.
3. The approach is formalized using the theory of algebraic graph transformation, and its soundness is presented: Two theorems state under which conditions the algorithm terminates and yields consistent models. Two corollaries present explicitly further features, namely, the hippocraticness of algorithm and the validity of the output.
4. An Eclipse-based tool (a meta-tool), called *Meta2RR*, is developed. The domain developer uses this tool to automate the derivation of the model transformation repair system from any given meta-model, i.e., to generate the domain-specific repair technique.
5. A model repair tool (a DSML tool), called *EMF Model Repair*, is developed as an Eclipse plug-in. It uses the generated domain-specific repair technique. Furthermore, the repair process can be stopped anytime, and the resulting models are EMF models. The application modeler uses this tool to repair EMF models randomly (fully automatically) or semi-automatically (interactively).
6. The algorithm is systematically tested concerning correctness, and a scalability experiment is reported. Possible adaptions of the algorithm and its usefulness in more general cases such as supporting a set of OCL constraints are discussed. The related work is presented as well.

In Section 3.3, we present our rule-based repair algorithm, apply it at an example, and give an overview of how a meta-model is translated into a rule-based transformation system with the help of meta-patterns. Additionally, we present and discuss the design of the different kinds of repair rules and transformation units. In Section 3.4.4, the algorithm is formalized and represented with the help of Hoare triples and graph automaton, and its correctness is shown. The developed Eclipse-based tools with their features are shortly presented in Section 3.5. Additionally, we present the correctness test of the technique. In Section 3.6, we report about the scalability experiments. After that, we discuss our approach in more general settings, e.g., supporting OCL constraints in Section 3.7. An overview of related work is given in Section 3.8 while Section 3.9 concludes the work and provides an overview of the future work.

3.2 Running Example

The running example is a simple *Webpage* modeling language for specifying a specific kind of web pages. We present the underlying meta-model first and show an invalid (inconsistent) *Webpage* model in its abstract syntax representation. Thereafter, we discuss briefly how this model is repaired such that it becomes fully consistent.

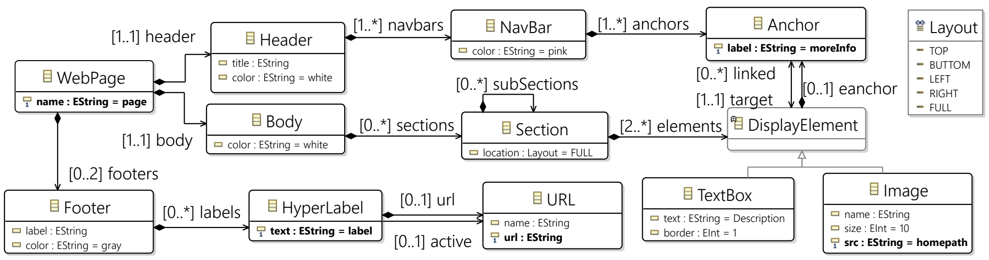


Figure 3.2: *Webpage* meta-model

Our *Webpage* modeling language is defined by the meta-model shown in Figure 3.2; its purpose is to generate web pages, e.g., in PHP. The meta-model elements are explained from left to right: Each web page has a *name* which requires a value *(being page by default)*. A web page contains a *Header*, up to two *Footer*s, and a *Body*. A header includes at least one navigator bar *NavBar* which contains at least one *Anchor*. A header has a *title* and a *color (being white by default)*, whereas a navigator bar has a *color (being pink by default)*. A body has a *color (being white by default)* and includes an arbitrary number of *Section*s. A section has a *location (being Full by default)* and may contain subsections. A section is designed to contain at least two arbitrary *DisplayElement*s of type *Image* or a *TextBox*. A text box has a *text (being Description by default)* and a *border (being 1 by default)*, whereas an image has a *name*, a *size (being 10 by default)* and a *src* which requires a value *(being homepath by default)*. Each display element may contain an *Anchor*. An anchor has a *label* which requires a value *(being moreInfo by default)* and must be connected to one display element. A web page must not contain more than two *Footer*s. A footer has a *label* and a *color (being gray by default)*; it may include an arbitrary number of *HyperLabel*s. A hyper label has a *text* which requires a value *(being label by default)* and may contain one

URL. A url node has a *name* and a *url* which requires a value. A hyper label may be connected to one url node. Note that the lower- and upper-bound invariants of several edge types are restricted just for presenting the power of our developed algorithm.

Figure 3.3 presents the corresponding abstract syntax representation of an invalid *Webpage* model repaired to a valid (consistent) one. The solid-line elements represent the invalid web page model. It consists of a blue header containing a pink navigator bar and three footers: footer *Appointment* contains a hyper label with text *calendar* including a url node with name *calurl* and a *url* attribute with empty value; the label *calendar* is activated. Footer *Address* contains a hyper label with text *floor 2*, and footer *Location* contains a hyper label with text *label3* including a *url* node with name *url3* and a *url* attribute with empty value. The label *label3* is activated.

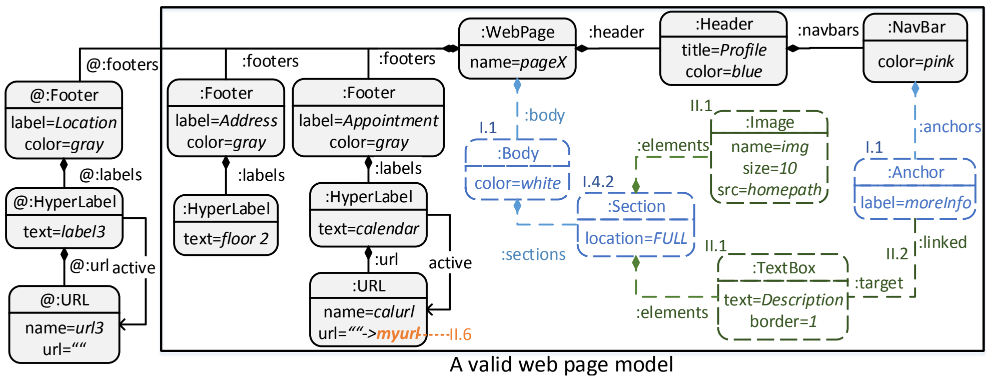


Figure 3.3: The abstract syntax of the invalid *Webpage* being repaired

To repair this invalid *Webpage* model, it is first trimmed and then completed: Since there are too many footers, one *Footer*-node, and its children have to be deleted. The selection of the footer can be done automatically in a non-deterministic way or determined by the user. We select the footer annotated with @. Then, the trimmed web page still contains all solid-line elements shown in Figure 3.3 without those annotated with @. To complete this *Webpage* model, at least the following missing elements (represented as dashed elements in Figure 3.3) have to be created: A *Body*-node and an *Anchor*-node are needed to fulfill the lower bounds of containment types *body* and *anchors*, respectively. An edge of type *target-linked* is needed to fulfill the lower bound of non-containment type *target*. Therefore, a *Section*-node containing two nodes of type *DisplayElement* (e.g., an image and a text box) are demanded to fulfill the lower bounds of containment type *elements* and non-containment type *target*. The *url*-attribute value of the *URL*-node *calurl* has to be set (to, e.g., *myurl*). The user may give suitable information during completion such as the types of *DisplayElements* as well as attribute values of new elements.

3.3 Rule-based Model Repair

Our approach to model repair consists of two main processes:

1. *Translation:* The meta-model of a given language is translated into a rule-based model transformation system (repair rules). The domain developer uses the translation to provide a domain-specific model repair technique.
2. *Model repair:* A potentially invalid (inconsistent) model is repaired yielding a valid (consistent) model. The repair algorithm is automatically configured to use the generated transformation system. The application modeler uses the repair tool to repair application models.

We start with process (2) presenting our repair algorithm in general and at an example. After that, we show how to translate a meta-model to a rule-based transformation system using several meta-patterns. The meta-patterns describe different repair rules, their schemes, and when they have to be generated. Furthermore, we represent the algorithm as transformation units.

3.3.1 A Rule-based Algorithm for Model Repair

As a pre-requisite for our approach to rule-based model repair, there is an instantiable meta-model without OCL constraints. Given a potentially invalid (inconsistent) model, i.e., a model over the given meta-model that may not fulfill all its multiplicities, our repair algorithm is able to produce a valid (consistent) one. The repair process is automatic but may be interrupted and interactively guided by the modeler.

The activity diagram in Figure 3.4 illustrates the overall control flow of our algorithm, which consists of two main parts:

- The left part performs *model trimming* eliminating supernumerous model elements.
- The right part performs *model completion* adding required model elements.

Model trimming: Step (a) in Figure 3.4 removes *all supernumerous edges*, i.e., edges that exceed the upper-bound invariants of each non-containment edge type. Step (b) checks if there is a *supernumerous node*, i.e., a node which exceeds the upper-bound invariants of a containment edge type. If there is none, the model is ready to be completed (Step (e)). Otherwise, this node and its content have to be deleted; this is done in Step (c). It deletes all the incoming and outgoing edges of this node and its content nodes, and then deletes the content nodes in a bottom-up way (Step (c1)); thereafter, it deletes the node itself (Step (c2)). This bottom-up deletion process starts at all the leaves and continues with their containers. Step (d) calls again Step (b) to check if there is another supernumerous node.

Model completion: Once there is no further supernumerous element in the input model, the model can be completed: Step (1) creates *all missing required nodes*, i.e.,

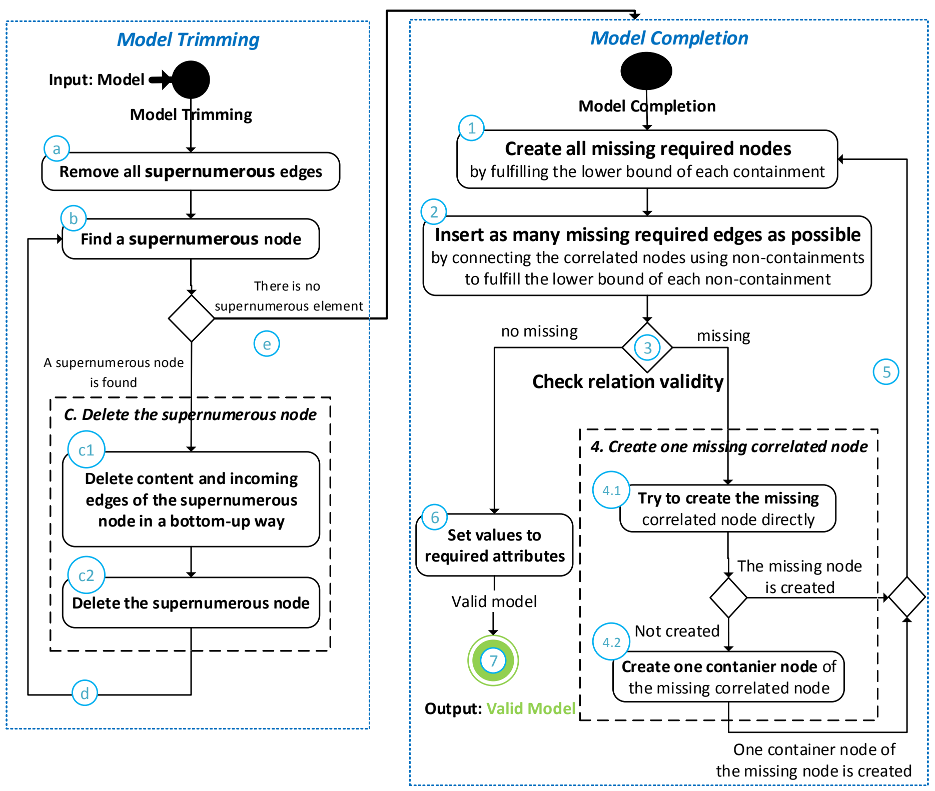

Model Trimming
Input: Model
Model Trimming
a
Remove all supernumerous edges
b
Find a supernumerous node
There is no supernumerous element
e
A supernumerous node is found
C. Delete the supernumerous node
c1
Delete content and incoming edges of the supernumerous node in a bottom-up way
c2
Delete the supernumerous node
d
Model Completion
Model Completion
1
Create all missing required nodes
by fulfilling the lower bound of each containment
2
Insert as many missing required edges as possible
by connecting the correlated nodes using non-containments to fulfill the lower bound of each non-containment
no missing
3
missing
Check relation validity
5
4. Create one missing correlated node
4.1
Try to create the missing correlated node directly
The missing node is created
Not created
4.2
Create one contanier node of the missing correlated node
6
Set values to required attributes
Valid model
7
Output: Valid Model
One container node of the missing node is created

Figure 3.4: Model trimming and completion algorithm

nodes that are needed to fulfill the lower-bound invariant of each containment edge type. Thereafter, we have a skeleton model that contains at least all the required nodes. It may contain additional nodes and edges which have been in the model before.

At this stage, the model may have nodes which are not correlated by required edges. Step (2) tries to insert *all missing required edges* by connecting the existing correlated nodes. These edges are needed to fulfill the lower-bound invariant of each non-containment edge type. This step may stop without having inserted all required edges due to potentially missing correlated nodes, i.e., it may happen that there is no further free node to correlate with.

Step (3) checks the validity of all required non-containment edge types. If all the required edges are already inserted, then we have a valid model w.r.t. all multiplicities of edge types. This also means that all the required nodes have been created. Otherwise, there is still at least one required edge missing.

In this situation, Step (4) tries to add one missing node to fulfill the lower bound of a non-containment edge type. Although the number of missing nodes may be more than one, only one missing node is added in Step (4). If a missing node cannot be created directly, a (transitive) container node of a missing one is created. The added node may function as, e.g., target node for missing required edges. Hence, it may help to solve other inconsistencies in the model. The design decision of adding just one node in Step (4) helps to find a small completion.

Note that the type of a missing node may have several subtypes. In this case, there may be several possibilities to create a missing node choosing one of these subtypes. A further variation point is node types with several containers. This non-determinism may be solved by the user interaction or automatically by randomly picking one. Thereafter, the next iteration of model completion starts with Step (5). Adding a node to the model may require adding further required nodes and edges. Starting a new iteration ensures that all those model elements are generated and that all missing nodes and edges will be created in the end. Once the instance model is valid w.r.t. edge type multiplicities, the values of all the empty required attributes are set in Step (6). In Step (7), the algorithm stops.

An Example Repair Process

We illustrate our algorithm by applying it to the invalid *Webpage* model in Figure 3.3. The invalid model is first trimmed and then completed. This process is described below and illustrated in Figure 3.3. The annotations at the model elements refer to the corresponding iteration and step numbers.

Model trimming: Since the model does not have supernumerous edges, we go directly to Step (b). Here, a supernumerous *Footer*-node is found. Assuming that the *Footer*-node *location* is selected for deletion. In Step (c1), the edge *active* is removed between the nodes *label3* and *url3*. Then, node *url3* itself is deleted. Thereafter, node *label3* can be deleted. Finally, Step (c2) deletes the selected *Footer*-node *Location*. Step (d) calls again Step (b) but there is no further supernumerous node. Hence, the process of model trimming is finished and the output model is ready to be completed (Step (e)).

Model completion: Completion is done in two iterations:

Iteration I: In Step (1), an *Anchor*-node and a *Body*-node are created to fulfill the lower-bound invariants of containment types *anchors* and *body*. In Step (2), a *target*-edge from the created *Anchor*-node is required but cannot be inserted. (3) The required non-containment edge type *target* has not been fulfilled yet. I.e., the node of type *Anchor* has to be connected to a node of type *DisplayElement* but there is none. Step (4.1) tries to create the missing correlated node directly: The creation of a *DisplayElement*-node fails since there is no available *Section*-node (used as container node). Consequently, Step (4.2) is required which creates a *Section*-node inside of the existing *Body*-node. In Step (5), the completion is continued. I.e., a second iteration is required.

Iteration II: In Step (1), two *DisplayElement*-nodes are created inside of the *Section*-node to fulfill the lower-bound invariant of the containment type *elements*. As an example, a node of type *TextBox* and a node of type *Image* are created. In Step (2), a non-containment edge of type *target* is inserted between the existing *Anchor*-node and one of the existing nodes of type *DisplayElement*, e.g., the *TextBox*-node. Step (3) finds out that all the required non-containment edge types are available now. In Step (6) all remaining values for the all required attributes are set. Here, the *url*-attribute value of the *URL*-node *calurl* is set to *myurl*. A valid instance model is produced now (Step (7)).

3.3.2 Deriving a Model Transformation Repair System from a Meta-Model

A pre-requisite of our repair algorithm is the existence of a model transformation system which configures the algorithm w.r.t. a given modeling language. The algorithm is configured with different sets of repair rules in a way that at any point of the algorithm where choices have to be made, they can be random, or interactive. This applies to choices of rules (repair actions), matches (model locations or targets), and attribute values. In other words, identifying the applicable rules and their matches aim at finding the proper repair actions and matches w.r.t the algorithm steps and the given model state. For example, Step 1 uses a kind of rules which creates a node with its containment edge in an existing node if the lower bound of its containment type is not yet reached. Step 1 executes the rules as much as possible and stops once there is no applicable rule. At this end, all the required nodes with all their required child nodes are created. Please note that the different kinds of repair rules are defined to be generic in the sense that they can be used to manage (fulfill) *any* lower or upper bound. Additionally, they consider the EMF model constraints. In other words, each successful application of a rule enhances (or at least preserves) the model consistency.

In the following, we present the different kinds of repair rules. Since these rules have to be derived from each given meta-model, we present their meta-patterns and how to derive them. Thereafter, we present the algorithm steps as transformation units being configured with the derived repair rules.

Repair Rules, their Meta-Patterns, and their Design

This section presents the design of *different kinds of repair rules* used to configure the algorithm steps. Since the number, the type and the design of these kinds of rules should be defined for each given meta-model, we provide the specifications of their meta-patterns. Based on [14], the schemes of the repair rules are designed to consider the EMF constraints, e.g., rules which insert edges are enhanced with application conditions to prevent the creation of parallel edges. However, not all the EMF constraints can be considered as graph application conditions such as the constraints *no-containment-cycle* and *rootedness* (non first-order logic), we, therefore, made the following design decisions:

- The rules do not insert containment edges between existing nodes to avoid the creation of containment cycles.
- The rules add a node with its incoming containment edge in an existing node to prevent producing more than one root, to build the tree-structure of the model and to not violate the constraint *at-most-one-container*. Furthermore, there is no rule which creates a free node, i.e., a node without an incoming containment edge.
- The algorithm deletes a node and its content in a bottom-up way to avoid violating the dangling condition.
- We require that the rules are executed at injactive matches only. Moreover, no rule will be executed if its application could violate the dangling condition.

Additionally, the application conditions of the rules are designed to be generic in the sense that the derived rule sets can be used to manage (fulfill) *any* lower or upper bounds. In other words, the different kinds of repair rules and their generic application conditions are designed to enhance the consistency of any given instance models w.r.t. multiplicities and to preserve the EMF constraints as well. *Each successful application of a rule enhances (or at least preserves) the model consistency and avoids violating other constraints, namely, the upper bounds and the EMF constraints.*

Table 3.1 gives an overview over the different kinds of rules and in which algorithm steps are used. Each algorithm step is configured with the proper kinds of rules, e.g., Step (1) is configured with the rules of kind *Required-node-creation*. In the following, we present a selection of the meta-patterns for deriving different kinds of rules for a given meta-model. All further rules kinds and their meta patterns with examples can be found as a catalog in Section A.1. Note, once a node is created in an EMF instance model, the node attributes with their default values are created as well. Thus, no need to specify them.

Our rules can also be used for different purposes, such as model editing. Recently, the catalog of the rules is already used for generating search operators in the field of search-based model engineering as presented in [22, 23].

- **Set of required-node-creation rules:** For *each* containment type *with* lower bound > 0, a rule is derived which creates a node with its containment edge if the *lower-bound invariant* of the corresponding containment type *is not yet reached*. The container is determined by the input parameter *p*. The user gets the possibility to select the right container. Otherwise, it is selected randomly by the algorithm. Figure 3.5 illustrates the rule scheme used to derive this kind of rules. Note that for each non-abstract class type B' in clan(B) such a rule is derived.

  Figure 3.6 presents an example rule; it creates a required *Body*-node being contained in a *WebPage*-node if there is not already a *Body*-node contained. This concrete rule is created with the help of the information of the domain match, namely the one presented in Figure 3.19.

Table 3.1: Kinds of rules

Kind of the rule	**Description**
Required-node-creation	Create a node with its containment edge if the lower bound of its containment type is not yet reached (Step (1))
Required-edge-insertion	Insert a non-containment edge if the lower bound of the corresponding non-containment type is not yet reached and there is no parallel edge (Step (2))
Required-edge-checking	Check if a required non-containment edge is missing (Step (3))
Additional-node-creation	Add a node with its containment edge if the upper bound of its containment type is not yet reached (Step (4))
Exceeding-edge-removing	Remove a non-containment edge if the upper bound of the corresponding non-containment type is exceeded (Step (a))
Exceeding-node-finding	Find a node that exceeds the upper bound of the corresponding containment type (Step (b))
Node-content-deleting	Delete a (transitively) contained leaf node with its containment edge from a given node (Step (c))
Element-deleting	Delete a given element (i.e., a non-containment edge or a leaf node with its containment edge) directly (Step (c))

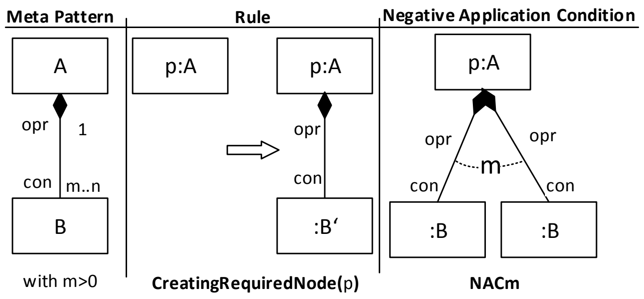


Figure 3.5: Rule scheme for creating a required *B*-node

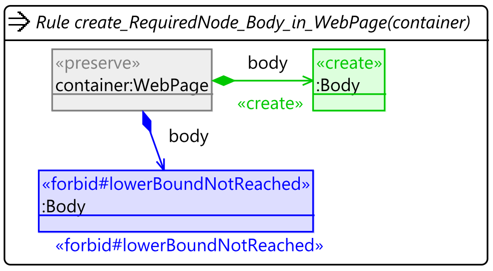

Rule create_RequiredNode_Body_in_WebPage(container)
«preserve»
container:WebPage
body
«create»
:Body
«create»
body
«forbid#lowerBoundNotReached»
:Body
«forbid#lowerBoundNotReached»

Figure 3.6: An example rule for creating a required *Body*-node

- **Set of required-edge-insertion rules:** For *each* non-containment edge type *with* one of its lower bounds m or k being > 0, a rule is generated which inserts a corresponding edge if the composite application condition NAC at the bottom of Figure 3.7 is satisfied. Condition *NACp* forbids to insert an edge between two nodes if there is already one of that type. Conditions *NACm* and *NACl* check if lower bound m has not been reached and the upper bound l has not been exceeded. Then, an edge has to be inserted between the given nodes and this is possible. Similarly, conditions *NACk* and *NACn* may be fulfilled; i.e., the lower bound k has not been reached and the upper bound n has not been exceeded. Note, the NAC in Figure 3.7 is generated in case that there is an opposite non-containment edge type and $0 < k, l, m, n < *$. For all other cases the NAC is adapted correspondingly.

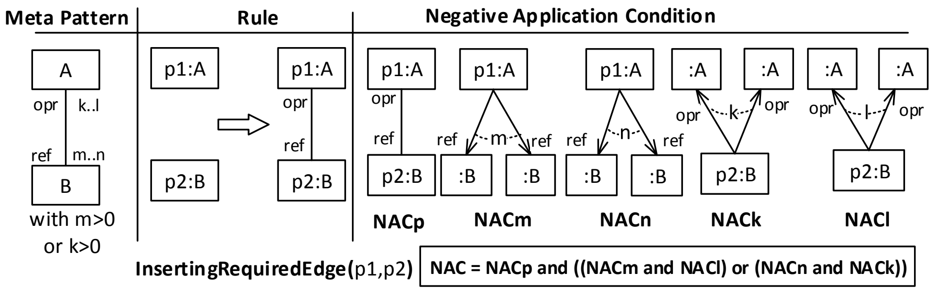


Figure 3.7: Rule scheme for inserting a required non-containment edge

In our example, there is only one required edge type which is the required opposite non-containment type *target*. Figure 3.8 presents the derived rule for inserting an edge of type *target* between two existing nodes: from an *Anchor*-node to a *DisplayElement*-node assuming that the source node is not already connected to a node of type *DisplayElement* by a *target*-edge. The opposite *linked*-edge is automatically inserted in EMF as well.

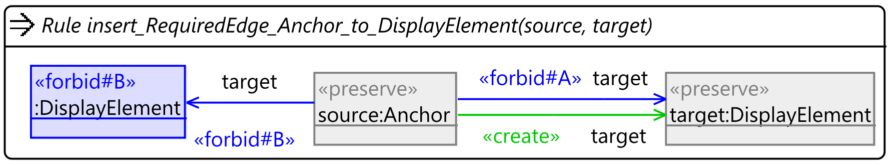


Figure 3.8: A rule for inserting a required non-containment edge of type *target*

- **Set of required-edge-checking rules:** This rule scheme checks if a required edge is missing. If a rule of this scheme can be successfully applied, there is at least one missing required edge. Such a rule scheme is generated for *each* non-containment edge type with $m > 0$. This meta-pattern, as well as the resulting rule scheme are shown in Figure 3.9. If the required non-containment edge type is a loop, another rule scheme is derived (see Sections A.1 and A.2).

  Figure 3.10 presents the rule which checks if a given *Anchor*-node is connected to a node of type *DisplayElement* by a *target*-edge. If this rule is applicable to a given instance model, there is a missing required edge of type *target*.

- **Set of additional-node-creation rules:** For *each* containment type of the given meta-model, a rule is generated which creates one node if *the upper-bound* of its containment type *is not exceeded*. These rules are used to add

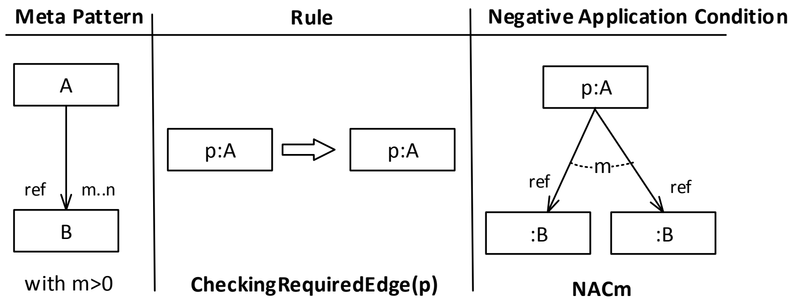


Figure 3.9: Rule scheme for checking the existence of required non-containment edges

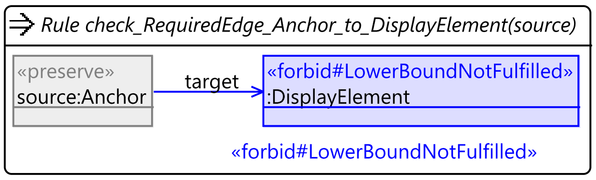

Rule check_RequiredEdge_Anchor_to_DisplayElement(source)
«preserve»
source:Anchor
target
«forbid#LowerBoundNotFulfilled»
:DisplayElement
«forbid#LowerBoundNotFulfilled»

Figure 3.10: A rule for checking the existence of a required edge of type *target*

nodes without violating upper-bound invariants. Note that, if the upper-bound invariant is specified as unlimited (i.e, $n = *$), the corresponding application condition is not generated. Figure 3.11 shows the meta- pattern as well as the resulting rule scheme for creating a node of type B being contained in an A-node. This rule scheme can be applied if $NACn$ is satisfied, i.e., if there are not n B-nodes already contained in this A-node.

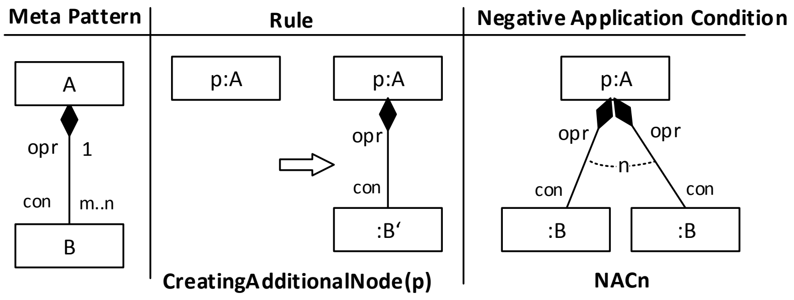
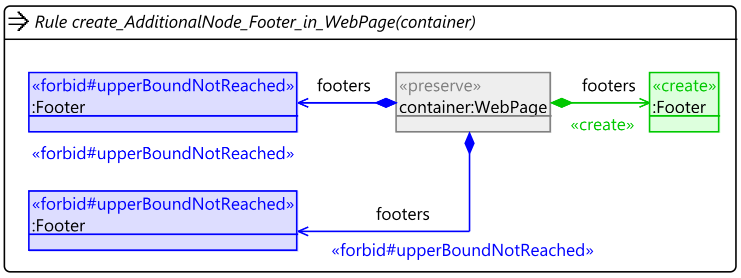


Figure 3.11: Rule scheme for creating an additional B-node

Figure 3.12 presents a rule for creating a *Footer*-node being contained in a *WebPage*-node if there are not already two *Footer*-nodes in the *WebPage*-node. This rule is derived due to edge type *footers*. Similar rules have to be derived for edge types *header*, *body*, *navbars*, *sections*, *subSections*, *elements*, *eanchor*, *footers*, *labels*, and *url*.

- **Set of exceeding-edge-removing rules:** For *each* non-containment type with a limited upper bound of the given meta-model, a rule is generated which re-



Rule create_AdditionalNode_Footer_in_WebPage(container)
«forbid#upperBoundNotReached» :Footer
footers
«preserve» container:WebPage
footers
«create» :Footer
«create»
«forbid#upperBoundNotReached»
«forbid#upperBoundNotReached» :Footer
footers
«forbid#upperBoundNotReached»

Figure 3.12: A rule for creating an additional node of type *Footer*

moves one edge if the *upper-bound* of its non-containment type *is exceeded*.

The rule scheme in Figure 3.13 removes an edge between an *A*-node and a *B*-node if the *PAC* is true, i.e., if there are *n+1 ref*-edges running from this *A*-node to other *B*-nodes in the model *(Cn+1)* or if there are *l+1 opr*-edges running from this *B*-node to other *A*-nodes in the model *(Cl+1)*. Figure 3.14 presents the derived rule for removing an *active*-edge between two existing nodes: from a *HyperLabel*-node to a *URL*-node assuming that the source node is already connected to an *URL*-node (including the *url*-edge) by an *active*-edge.

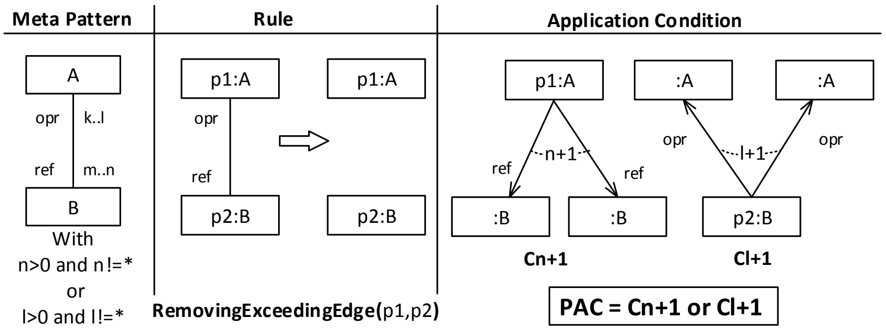
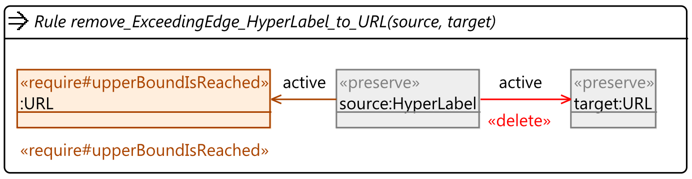


Figure 3.13: Rule scheme for removing an exceeding edge



Rule remove_ExceedingEdge_HyperLabel_to_URL(source, target)
«require#upperBoundIsReached» :URL
active
«preserve» source:HyperLabel
active
«delete»
«preserve» target:URL
«require#upperBoundIsReached»

Figure 3.14: A rule for removing a supernumerous *target*-edge

- **Set of target-node-deleting rules:** For *each* containment type of a given meta-model, this rule scheme deletes a given node from its container. The meta-pattern as well as the resulting rule scheme are shown in Figure 3.15. Each rule is configured with a parameter to delete the given node if possible, i.e., once the given node has no contents and no non-containment edges. Figure 3.16 presents the rule for deleting the *Footer*-node (given as an input parameter) from a *WebPage*-node.

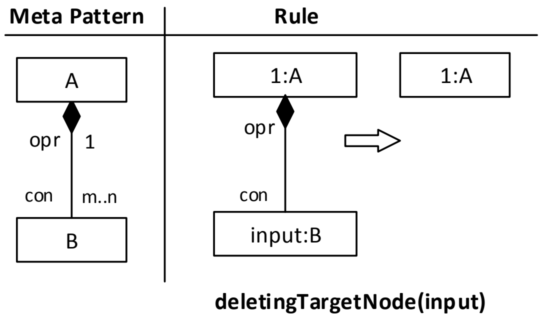


Figure 3.15: Rule scheme for deleting a given node

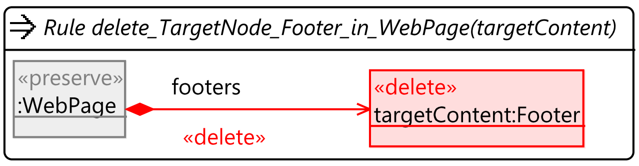


Figure 3.16: A rule for deleting a given *Footer*-node

The algorithm which is being configured with the rule sets not only aims at identifying and repairing the bound violations but also aims at repairing the model (deleting and constructing model elements) in a consistency-preserving way. I.e., the resulting model satisfies not only the bounds but also the other EMF constraints (Def. 5). Thus, the model editor, e.g., can open the resulting model at any time.

The Process of Rule Derivation

To derive the repair rules from any given domain meta-model, we follow the following process as depicted in Figure 3.17:

1. Specifying the meta-patterns as rules typed by the Ecore meta-metamodel.
2. Applying them to the given domain meta-model to retrieve the corresponding rule matches (if exists).
3. Using the domain data of the rule matches and the rule scheme of the corresponding meta-pattern, we specify and derive (repair) rules which can be applied to instance models of the given domain.

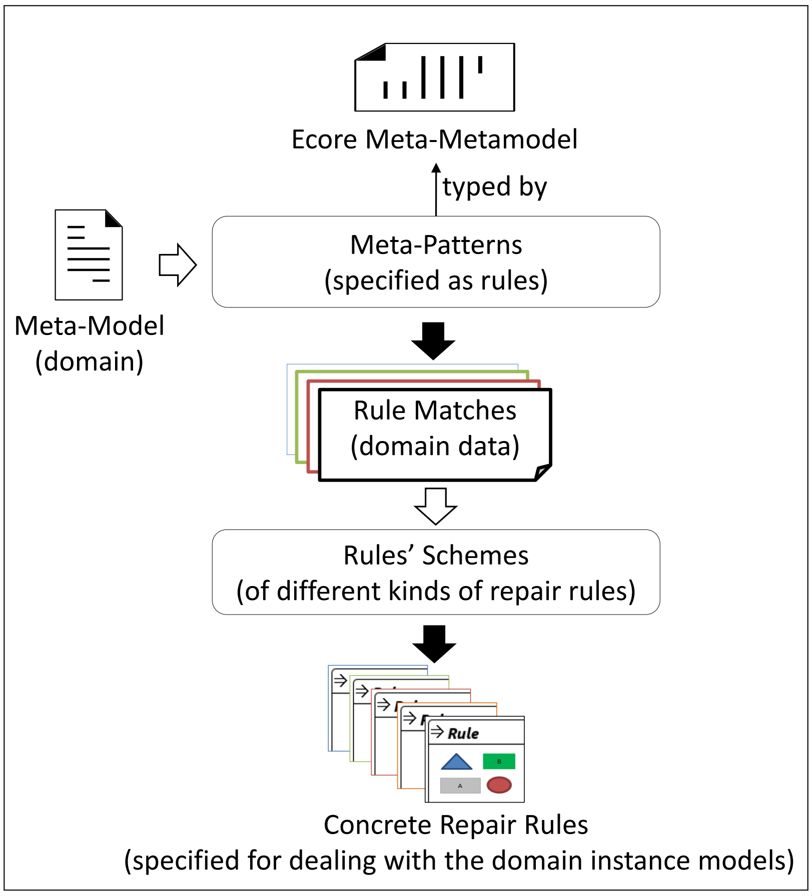


Figure 3.17: The derivation process of rules

Figure 3.18 shows the meta-pattern rules typed by the Ecore meta-metamodel representing the meta-pattern depicted in Figure 3.5. The meta-pattern rules *findChildrenNodes1* and *findChildrenNodes2* are specified to find the non-abstract children types (supporting inheritance) of a container type if the lower bound of the containment edge type that connects the container type with the content type is $>= 1$.

Figure 3.19 shows an example of a match resulting from applying the meta-pattern rule *findChildrenNodes1* to the Webpage meta-model. The match consists of a node type *Webpage* which is connected using a containment edge type *body* to a node type *Body*. The lower bound of the containment edge type *body* is 1. Using the rule scheme depicted in Figure 3.5, the concrete repair rule for creating a required Body-node is generated as depicted in Figure 3.6.

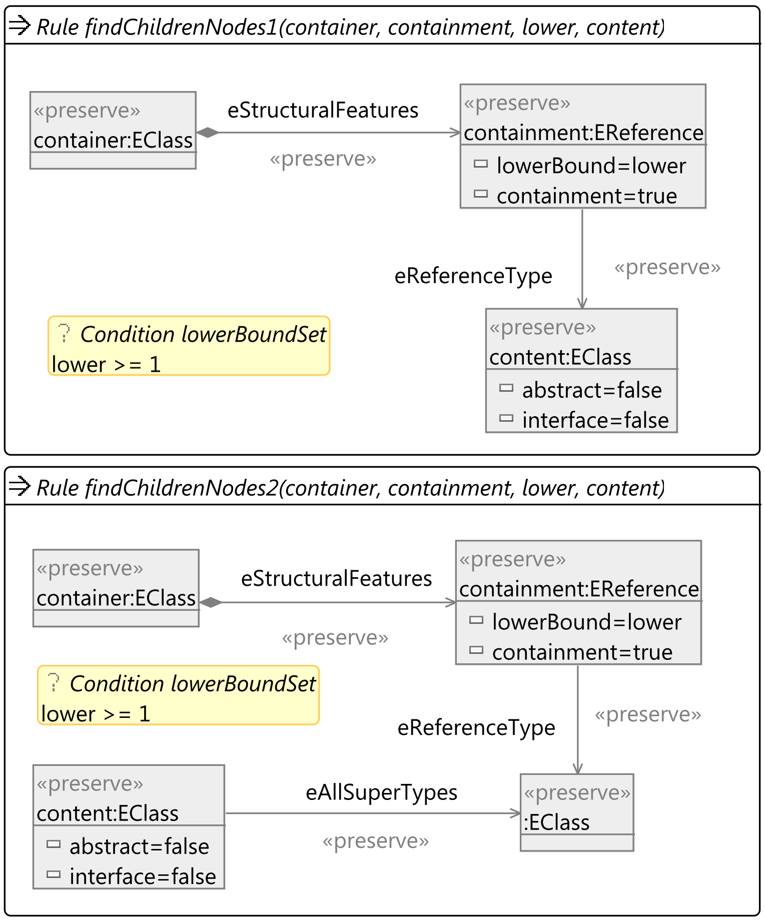


Figure 3.18: Example of meta-patterns specified as rules typed by the Ecore meta-metamodel

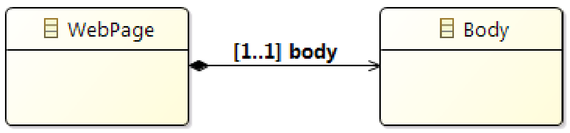


Figure 3.19: An example of a match of the meta-pattern shown in Figure 3.18

Algorithm Steps as Transformation Units

This section presents algorithm steps as unit transformations being configured with the proper set of the derived repair rules. For the representation purpose we use Henshin transformation units. Henshin is a language and tool environment for EMF model transformation [3].

In the following, we present the specifications of some steps. More details about the specification of the algorithm can be found in Section A.3.

Several algorithm steps such as Step (a), Step (1) and Step (2) have to be applied as much as possible in a non-deterministic way. This is specified using a loop unit composing an independent unit being configured with the proper rules. We call such specification as a layer unit. For example, Step (a) of model trimming is specified using a loop unit *Remove_All_ExceedingEdges* composing an independent unit *Remove_One_ExceedingEdge* being configured with the rules of kind *exceeding-edge-removing* as shown in Figure 3.20. The loop unit will be applied as often as its independent unit is applicable, and the independent unit is applicable if at least one of its rules is applicable. Since an exceeding-edge-removing rule is designed to be applicable if and only if there is an exceeding edge, executing this layer unit ensures that all the exceeding edges will be removed.

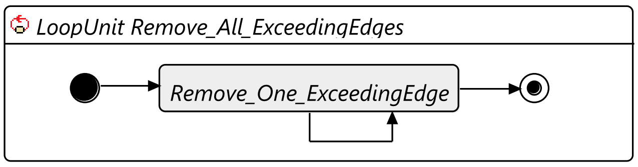


Figure 3.20: Layer transformation unit for Step (a)

Another example, we present the Step (4) of model completion, which is specified with the help of the following transformation units, as shown in Figure 3.21.

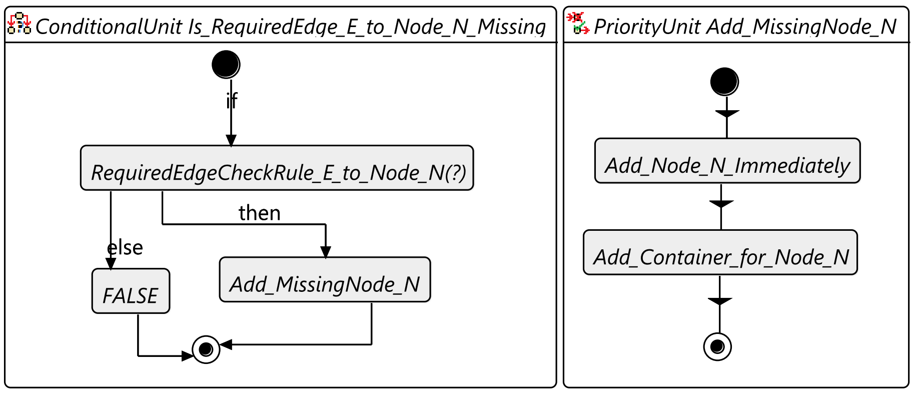

ConditionalUnit Is_RequiredEdge_E_to_Node_N_Missing
if
RequiredEdgeCheckRule_E_to_Node_N(?)
then
else
FALSE
Add_MissingNode_N
PriorityUnit Add_MissingNode_N
Add_Node_N_Immediately
Add_Container_for_Node_N

Figure 3.21: Transformation units for Step (4)

Figure 3.21 consists of a conditional unit *Is_RequiredEdge_E_to_Node_N_Missing* which checks if there is a missing edge of type *E*. The *if-statement* of the conditional unit is configured with a corresponding rule of kind *Required_edge_checking*. The checking rule is applicable if and only if there is a missing edge of type *E* to a node of type *N*. In this case, the *then-statement* is called. Otherwise, the *else-statement* returns false. The *then-statement* is specified as a priority unit *Add_MissingNode_N* composing two ordered sub-units of kind independent unit: (1) an independent unit *Add_Node_N_immediately*. (2) an independent unit *Add_Container_for_Node_N*. These two independent units are configured with the corresponding rules of kind *Additional_node_creation*. Thus, the priority unit adds a node of type *N* immediately in an existing container node without violating the respective upper bound. If it failed, it adds a container node of a container type for the type *N* without violating the respective upper bound. Please note that for each required non-containment edge type of the meta-model, such transformation units are derived as well.

Summary

In the following, we outline the main activities of the whole process of our approach:

- model elements are trimmed to satisfy the negative constraints, i.e., the upper bounds,
- the trimmed model is completed to satisfy the positive constraints (the lower bounds) without violating the negative ones (the upper bounds),
- several kinds of repair rules of different actions and transformation units are defined to deal with an arbitrary instance model violating possibly the negative and positive constraints (the lower and upper bounds). They are designed in a way that each successful application of a rule enhances (or at least preserves) the consistency of the model,
- the repair rules are designed to preserve the EMF constraints by defining not only proper application conditions but also algorithms (control flows) to deal with constraints like the dangling condition and rootedness, and
- rule schemes and meta-patterns are defined to derive the proper repair rules from any given meta-model and to derive the control flows of the algorithm.

3.4 Formalization

3.4.1 Prerequisite

In this section, we provide several definitions and functions needed to show the soundness of our algorithm and its features, present it using Hoare triples and graph automaton, and prove its correctness.

Since different degrees of conformity to the EMF constraints are needed, we present and formalize the following functions and definitions as follows:

- **No-nonCont-ub-violation:** For each node and each type of non-containment edges the number of outgoing edges of this type is smaller or equal to the upper bound of this edge type:

$$\forall n \in V_G . \forall e_T \in NC . \\ \#\{e \in E_G \mid type_{E_G}(e) = e_T \wedge s_G(e) = n\} \leqslant upp(e_T) \ .$$

- **No-Cont-lb-violation:** Each node has (at least) the required number of outgoing containment edges:

$$\forall n \in V_G . \forall c \in C\big(type_{V_G}(n) \leqslant s_T(c) \\ \Rightarrow \#\{e \in E_G \mid type_{E_G}(e) = c \wedge s_G(e) = n\} \geqslant low(c)\big) \ .$$

- **nonCont-lb-violation:** For (at least) one node an outgoing edge of a certain non-containment type is missing:

$$\exists n \in V_G . \exists e_T \in NC\big(type_{V_G}(n) \leqslant s_T(e_T) \\ \wedge \#\{e \in E_G \mid type_{E_G}(e) = e_T \wedge s_G(e) = n\} < low(e_T)\big) \ .$$

- **Cont-ub-violation:** There exists a node with a supernumerous outgoing edges of a certain containment type:

$$\exists n \in V_G . \exists c \in C . \\ \#\{e \in E_G \mid type_{E_G}(e) = c \wedge s_G(e) = n\} > upp(c) \ .$$

- **Missing-edge(e_T):** An edge of non-containment type e_T is missing:

$$\exists n \in V_G\big(type_{V_G}(n) \leqslant s_T(e_T) \wedge \\ \#\{e \in E_G \mid type_{E_G}(e) = e_T \wedge s_G(e) = n\} < low(e_T)\big) \ .$$

- **No-target-exists(e_T):** No node of G can serve as target node for a non-containment edge of type e_T without violation of upper bounds:

$$\forall n \in V_G\Big(type_{V_G}(n) \nleqslant t_T(e_T) \vee \big(e_T \in OE \wedge \\ \#\{e \in E_G \mid type_{E_G}(e) = e_T \wedge t_G(e) = n\} = upp(e_T^{-1})\big)\Big) \ .$$

- **No-target-creation-possible(e_T):** For an edge of non-containment type e_T it is not possible to directly create a node, which can serve as target for an edge of this type:

$$\forall n \in V_G . \forall c_T \in C\big(type_{V_G}(n) \leqslant s_T(c_T) \wedge t_T(e_T) \leqslant t_T(c_T) \\ \Rightarrow \#\{c \in C_G \mid type_{E_G}(c) = c_T \wedge s_G(c) = n\} = upp(c_T)\big) \ .$$

Furthermore, we need to introduce two counters: For a given typed graph G the counter *ContUbViol(G)* gives the number of upper bound violations of containment type edges in G, which can be interpreted as the number of surplus nodes, while *nonContLbViol(G)* counts the number of lower bound violations of edges of non-containment type, i.e., the number of missing edges.

When considering EMF-model graphs as instances of a meta-model, one expects them to be concrete and typically to be rooted. We did not include this in the above definition to be able to define rules more conveniently. But when repairing an instance, we suppose the input to be concretely typed and rooted and our aim is to receive such an EMF-model graph as output since that is common practice in EMF. So, unless stated otherwise, when talking about EMF-model graphs in the following, we always assume them to be concrete and rooted; we use **EMFC** as a shortcut for this.

Definition 8 (Valid EMF-Model Graph (vEMFC)). A concrete and rooted EMF-model graph G is called *valid*, short **vEMFC**, if it additionally satisfies the *No-bound-violation* property.

Usually, meta-models are expected to have at least one non-empty finite instance model or for each non-abstract class of the EMF model one instance which instantiates this class. Since we are going to avoid deletion during the second part of our developed algorithm, we need to introduce a stricter notion additionally. The second part of the algorithm expects instances, which may violate lower, but no upper bounds, to be capable of being complemented into a valid EMF-model graph (without deletion of elements):

Definition 9 (Fully Finitely Instantiable). A meta-model is called *finitely satisfiable* (f.s.) if there exists a non-empty finite instance. It is called *finitely instantiable* (f.i.) if for every non-abstract class there exists a finite EMF instance model, instantiating it. It is called *fully finitely instantiable* (f.f.i.) if it satisfies the property that for every given finite EMF-model graph G which respects upper, but may violate lower bounds there exists a finite EMF-model graph G', s.t. G is a subgraph of G'.

By definition the following implications hold: *f.f.i.* $\Rightarrow$ *f.i.* $\Rightarrow$ *f.s.*

Example 1 (Fully finitely instantiable). The meta-model presented in Figure 3.2 is f.f.i.: Since no containment cycle with all lower bounds $\geqslant 1$ exists, the creation of missing nodes to fulfill all lower bounds of containment type edges will always result in a finite model. Thereafter, a node of type `DisplayElement` will always exist so that all `Anchor`-nodes can be targeted. More generally, let TG be a type graph without cycles over containment edges with all lower bounds > 0 and for every node $n \in (V_T \backslash A)$ the upper bound of at least one incoming containment relation is higher than the lower bound of all non-containment edges for which n can serve as target node. Then TG is f.f.i. This is particularly the case if all (containment) edges have unlimited upper bounds.

Example 2 (Finitely instantiable). Suppose that the multiplicity of `linked` in Figure 3.2 is [1..1], i.e., each `DisplayElement` is linked by exactly one `Anchor` which targets exactly one `DisplayElement`, the upper bounds of `navbars` and `anchors` are set to, e.g., 2 and the `eanchor` edge is dropped out. The result is a meta-model which is f.i., but not f.f.i.: One easily creates valid instance models, but invalid instance models with 5 or more nodes of type `DisplayElement`, which do not violate any upper bound, cannot be complemented into a valid instance since maximally 4 nodes of type `Anchor` can exist in a valid model and therefore the needed `linked` can never be inserted.

Example 3 (Not finitely satisfiable). Suppose the containment type edge `subSections` in Figure 3.2 to have multiplicity [1..∗]. This is a typical example of a meta-model which is not f.s. or f.i.: The lower bound 1 leads to an infinite chain of `subsections`.

3.4.2 Hoare Triple Representation of the Algorithm Steps

In this section, we introduce pre- and postconditions for each algorithm step. We will present those in a way reminding of Hoare triples [62] which describe how the execution of the algorithm changes the state of instance models. By

$$\langle\, \mathit{PreX}, \mathit{Step}(X), \mathit{PostX} \rangle$$

we assert that, given a graph G satisfying *PreX*, the execution of Step (X) will lead to a graph G' satisfying *PostX*. Please note that in most steps, the rules are randomly applied as often as possible and the next step is called as soon as no rule is applicable anymore. Each rule preserves the model consistency w.r.t. EMF constraints and does not introduce an EMF violation.

In the following we recall the description of the algorithm steps and present their respective Hoare triples. The pre- and postconditions are formalized using the functions in Section 3.4 (see Definition 4).

Model trimming: This phase eliminates supernumerous model elements. It consists of the following steps: Step (a) removes *all supernumerous edges*, i.e., non-containment edges that exceed the upper bound of their respective type:

$$\langle\, \mathit{EMFC},\ \text{Step (a)},\ \mathit{EMFC} \wedge \mathit{No\text{-}nonCont\text{-}ub\text{-}violation} \rangle.$$

Step (b) checks if there is *a supernumerous node*, i.e., a node which exceeds the upper bound of a containment edge:

$$\langle\, \mathit{Post}(a),\ \text{Step (b)},\ \mathit{Post}(a) \rangle$$

If there is none, the model is ready to be completed (Step (e)). Otherwise, this node and its content have to be deleted. This is done in Step (c). It deletes all the incoming and outgoing edges of this node and its content nodes, and then deletes the content nodes in a bottom-up way (Step (c1)); thereafter, it deletes the node itself (Step (c2)):

$$\langle\, Post(a) \wedge \textit{Cont-ub-violation},\ \text{Step (c)},\ Post(a) \wedge (ContUbViol(G) > ContUbViol(G'))\,\rangle.$$

Step (d) calls Step (b) again to check if there is another supernumerous node.

Model completion: This phase adds required model elements. It consists of the following steps: Step (1) creates *all missing required nodes*, i.e., nodes that are needed to fulfill the lower bound of a containment edge type:

$$\langle\, EMFC \wedge \textit{No-ub-violation},\ \text{Step (1)},\ EMFC \wedge \textit{No-ub-violation} \wedge \textit{No-cont-lb-violation}\,\rangle.$$

At the end of this stage, the model may contain nodes that are not connected by required non-containment edges.

Step (2) tries to insert *all missing required edges* by adding non-containment edges to the nodes in the model, in order to fulfill the lower bound of each non-containment edge type. This step may stop without having inserted all required edges due to potentially missing target nodes, i.e., it may happen that there is no further free node of a required type. Thus, we get

$$\langle\, Post(1),\ \text{Step (2)},\ Post(2)\,\rangle$$

$$\begin{aligned} Post(2) := {} & \Big(EMFC \wedge \textit{no-bound-violation}\Big) \\ \vee & \Big(Post(1) \wedge \exists e_T \in NC\big(\textit{Missing-edge}(e_T) \wedge \textit{No-target-exists}(e_T)\big) \\ & \wedge NonContLbViol(G) \geqslant NonContLbViol(G')\Big). \end{aligned}$$

Step (3) checks if a lower bound of a non-containment edge type is still violated. If all the required edges are already inserted, then we have a valid EMF-model graph. Otherwise, there is still at least one required edge missing. So, Step (3) checks which of the two possibilities of *Post*(2) is actually true and additionally returns the type e_T of one missing non-containment edge (if true), but does not change the instance.

If an edge was missing, Step (4) tries to create a target node for the missing edge type. Although there may be more than one target node missing, only one is added in Step (4). If it cannot be created directly (Step 4.1), a (transitive) container node of it is created (Step 4.2). So we receive the following pre- and postconditions for these steps, where e_T was returned by Step (3):

$$\begin{aligned} Pre(4.1) := {} & EMFC \wedge \textit{No-ub-violation} \wedge \textit{No-cont-lb-violation} \\ & \wedge \textit{Missing-edge}(e_T) \wedge \textit{No-target-exists}(e_T). \end{aligned}$$

$$\begin{aligned} Post(4.1) := {} & EMFC \wedge \textit{No-ub-violation} \wedge \textit{Missing-edge}(e_T) \wedge \\ & \big(\textit{target-exists}(e_T) \vee (\textit{No-target-creation-possible}(e_T) \wedge \\ & \textit{No-cont-lb-violation})\big). \end{aligned}$$

$$\begin{aligned} Pre(4.2) := {} & EMFC \wedge \textit{No-ub-violation} \wedge \textit{No-cont-lb-violation} \wedge \\ & \textit{Missing-edge}(e_T) \wedge \textit{No-target-creation-possible}(e_T). \end{aligned}$$

$$\begin{aligned} Post(4.2) := {} & EMFC \wedge \textit{No-ub-violation} \wedge \textit{Missing-edge}(e_T) \wedge \\ & \exists n \in V_{G'} \big(n \notin V_G \wedge type_{V_{G'}}(n) \sqsupseteq t_T(e_T)\big). \end{aligned}$$

Note that the type of a missing target node may have several subtypes. In this case, there may be several possibilities to create a missing node, choosing one of these subtypes. A further variation point is node types with several containers. This non-determinism may be solved by user interaction or automatically by randomly picking one.

After Step (4), the next iteration of model completion starts by calling Step (1) again: Adding a node to the model may lead to new required nodes and edges. Starting a new iteration ensures that all those model elements are generated and that all missing nodes and edges will be created in the end. Once the model is a valid EMF model, i.e., EMF constraints and multiplicities are met, the values of all the empty required attributes are set (Step (6)).

3.4.3 The Algorithm as Graph Repair Automaton

We represent the repair algorithm as a program automaton. The states of the automaton are given by graph formulas (or constraints) and the transitions by graph programs. To be able to do this, we first extend the definition of graph programs [106].

Definition 10. Graph programs are inductively defined:

1. Every rule is a program.
2. Every finite set of programs (including the empty one) is a program.
3. Given programs P and Q then $(P; Q)$ and $P\downarrow$ are programs.
4. Given programs P, Q and R, then *if P then Q* else R is a program.

5. Given programs $P_1, \ldots, P_n$ then $\overrightarrow{P_1 \ldots P_n}$ is a program.

We additionally introduce the notation P^k as abbreviation for $P; \ldots; P$ (k times).

The semantics of the newly introduced programs is as follows:

if** P **then** Q **else** R:** Execute the program P. If it was executable, execute program Q. If not, execute R. We write *if* P *then* Q for *if* P *then* Q *else* $\{\}$.

$\overrightarrow{P_1 \ldots P_n}$**:** Try to execute the programs in the given order and stop as soon as one program was executed.

With those newly introduced graph programs, almost all Henshin units [3] can be represented as graph programs. The correspondences are given in Table 3.2.

Graph Program	Henshin Unit
single rule	single rule
set of programs	independent unit
$(P; Q)$	sequential unit
$P \downarrow$	loop unit
if P *then* Q *else* R	conditional unit
$\overrightarrow{P_1 \ldots P_n}$	priority unit
P^k	iteration unit (k iterations of the unit)

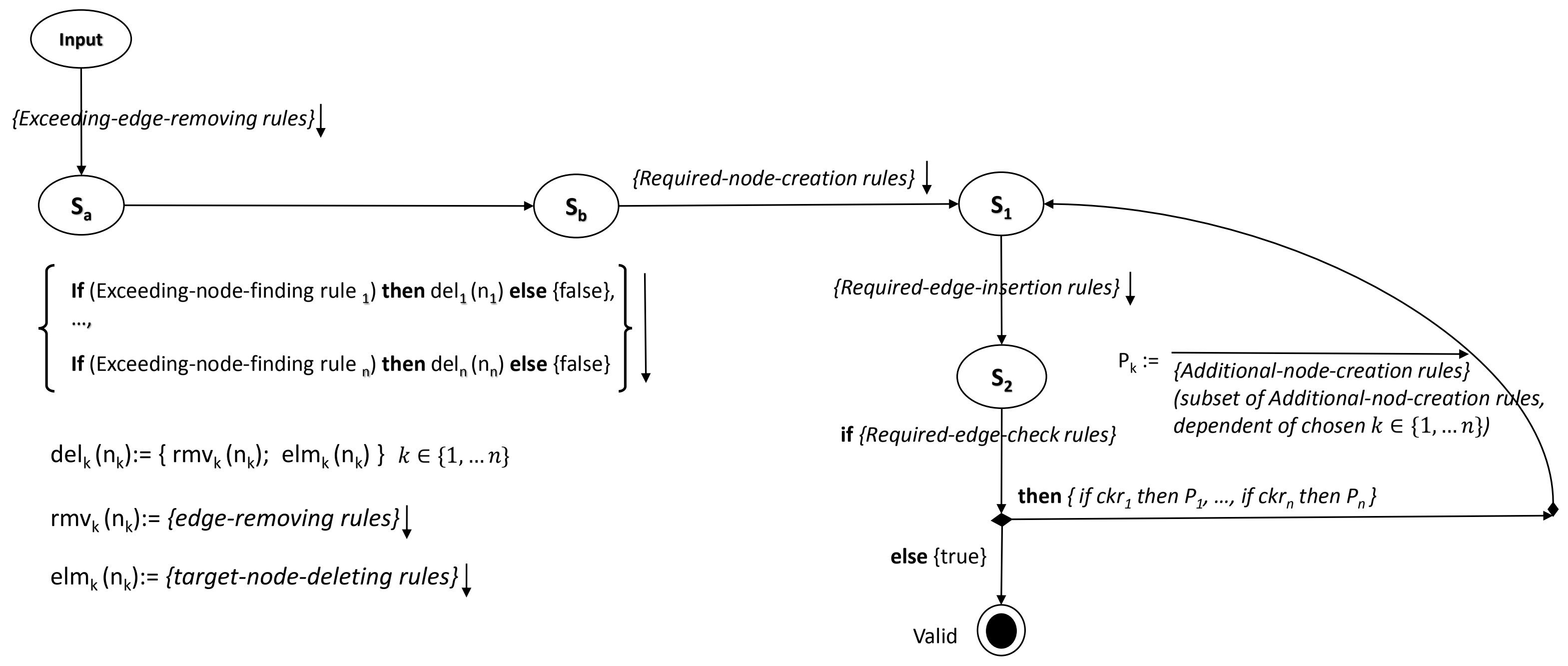
Table 3.2: Correspondence between graph programs and Henshin Units

Since we need to use the *if* ... *then* ... *else* ... statement, we need to additionally extend the concept of transition in automata: A *conditional transition* can lead to two of either states (or even into a new conditional transition), depending whether the *then* or the *else* statement is executed.

We now use the newly introduced graph programs to present (most parts of) our algorithm as a program automaton as shown in Figure 3.22.

The states of the automaton are given by properties of the instance. Transitions are given by graph programs as introduced above. The states are given by the following properties:

- *Input*: Input can be any instance that satisfies the EMF constraints.
- S_a: Instances satisfy all EMF constraints and no upper bound of non-containment edges is violated.
- S_b: Instances satisfy all EMF constraints, and no upper bound of any (containment or non-containment) edge is violated.
- S_1: Instances satisfy all EMF constraints, no upper bound is violated, and no lower bound of containment edges is violated.



Input
{Exceeding-edge-removing rules}↓
S_a
S_b
{Required-node-creation rules}↓
S_1
{Required-edge-insertion rules}↓
S_2
If (Exceeding-node-finding rule $_1$) then $del_1(n_1)$ else {false},
…,
If (Exceeding-node-finding rule $_n$) then $del_n(n_n)$ else {false}
$del_k(n_k)$:= { $rmv_k(n_k)$; $elm_k(n_k)$ } $k \in \{1, \ldots n\}$
$rmv_k(n_k)$:= {edge-removing rules}↓
$elm_k(n_k)$:= {target-node-deleting rules}↓
P_k := {Additional-node-creation rules} (subset of Additional-nod-creation rules, dependent of chosen $k \in \{1, \ldots n\}$)
if {Required-edge-check rules}
then { if ckr_1 then P_1, …, if ckr_n then P_n }
else {true}
Valid

Figure 3.22: Our algorithm as program automaton

- S_2: Instances satisfy all EMF constraints, no upper bound is violated and (no lower bound of any edge is violated, or a target node for a missing non-containment edge is missing).
- *Valid*: Instances satisfy all EMF constraints, no upper bound is violated, and no lower bound is violated.

Note, the *tiny diamonds* are only decision points (not states).

The transitions are defined by the graph programs which are annotated in Figure 3.22. Please note that following remarks:

- The notation $\{set\ of\ rules\} \downarrow$ is implemented as a loop unit containing an independent unit in Henshin; if R denotes the set of rules, it is just $R \downarrow$ as graph program.
- The transition from S_a to S_b is not entirely depicted by graph programs as introduced above. One would additionally need to: (1) introduce parameters into graph programs, e.g., if a superflous node is identified, exactly its children also have to be deleted with it. (2) a graph program to go over each node content of a given node. A such program is needed to completely represent the $rmv_k(n_K)$ and $elm_k(n_k)$.
- The depiction of the outgoing transition from the first diamond then $\{$ if ckr_1 then $P_1, \ldots,$ if ckr_n then P_n $\}$ is a sligth simplification: This *then*-branch of the statement calls an independent unit which contains conditional units. As graph programs we have *if* ckr_k *then* P_k, where ckr_k is one check-rule and P_k contains a subset of the additional-node-creation rules. This subset contains those rules needed to create the missing target node – for which the check-rule checks – or one of its containers.

3.4.4 Correctness Proof of the Algorithm

In this section, we present and proof some properties of the algorithm. In the following, let TG be a fixed type graph over which all the occuring graphs are typed.

Our trimming of models restores integrity with respect to upper bounds:

Theorem 1 (Correctness of Model-Trimming). The *model trimming* algorithm in Section 3.3.1 is correct, i.e., given an EMF-model graph G, the algorithm terminates in a finite number of steps, yielding an EMF-model graph G' that satisfies all upper bounds of TG.

Proof. For a graph G, let $ub(G)$ denote the sum of the number of upper bound violations of (1) containment and (2) non-containment relations. Let G' denote a graph resulting from the application of one of the rules of kinds that constitute the Steps (a) – (c) of the algorithm (see Table 3.1) on graph G. If all upper bounds of G are satisfied, $ub(G) = 0$, Step (a) and (b) do not change the graph, and the trimming algorithm terminates.

Otherwise, an upper bound is unsatisfied. Step (a) removes all supernumerous edges that violate the upper bounds of a non-containment relation, so $ub(G') \leqslant ub(G)$ and *No-nonCont-ub-violation* is certainly true afterwards. Step (b) checks for supernumerous nodes that violate the upper bounds of a containment relation, without changing the graph, so $ub(G) = ub(G')$ and the postcondition of this step equals its precondition. Step (c1) does two things to a supernumerous node v: (1) If v is a container, all content nodes of v (and their edges) are deleted in a bottom-up way. This may remove, but not add supernumerous nodes, so $contUbViol(G') \leqslant contUbViol(G)$ and equally $ub(G') \leqslant ub(G)$. If v is not a container, the graph remains unchanged. (2) Incoming edges of v are removed. In Step (c2), node v itself is removed, so $ub(G') < ub(G)$ and equally $contUbViol(G') < contUbViol(G)$.

Because $ub(G)$ decreases with every iteration, the algorithm terminates with a graph which does not violate any upper bound as result. □ □

For the completion phase of our algorithm to work, we need the meta-model to exhibit certain qualities: It needs to be always possible to create missing target nodes for required edges (guaranteed by full finite instantiability) and the creation of one (of the) possible target node(s) for a required edge may not lead to the need to create a node of the same type again infinitely often. We characterize precisely how to prevent this second problem:

Sometimes, when a node has to be created to serve as target for a required non-containment edge, there exist essentially different ways to do that. This can be due to two causes: There exist different containment paths from the root to the target node in the sense that in one path a node type occurs that does not occur in another path. Or because of inheritance there are different possible node types that can serve as target node. We need to pay closer attention to situations like these if the required edge is an opposite edge.

Definition 11 (Critical Opposite Edge). An opposite edge $e \in OE$ is called *critical* if $low(e) \geqslant 1$ and there either exists more than one possible target node type, i.e., $\#(clan_I(t_T(e)) \cap (V_T \backslash A)) > 1$, or there are containment paths p, p' for a target node type for e s.t. a node type occurs in one path that does not occur in the other.

If e is such a critical opposite edge, we require one of the following two conditions to hold:

(a) The opposite edge e^{-1} has unlimited upper bound, or

(b) (one of) the target node type(s) has at least one incoming containment edge with unlimited upper bound and if there are different possible target node types, they have to be able to serve as sources of exactly the same edge types:

Definition 12 (Acceptance Condition for Critical Opposite Edges). A type graph *TG satisfies the acceptance condition of the critical opposite edge (ACCOE) if every required opposite edge e satisfies condition (a) or condition (b):*

(a) *Condition (a)* on e demands $upp(e^{-1}) = *$, or

(b) *Condition (b)* on e demands (b1) that there exists a containment edge $c \in C$ with $t_T(e) \leqslant t_T(c)$ and $upp(c) = *$. And if $clan_I(t_T(e)) \neq \{t_T(e)\}$ it additionally demands (b2) that for each two nodes $n, n' \in clan_I(t_T(e))$ and every edge $e' \in E_T$

$$n \leqslant s_T(e') \Leftrightarrow n' \leqslant s_T(e') \ . \tag{3.1}$$

Theorem 2 (Correctness of Model-Completion). Let TG be a fully finitely instantiable type graph whose critical opposite edges (if exist) meet the acceptance condition. Then the *model completion* from Section 3.3.1 is correct, i.e., given an EMF-model graph G, which satisfies *No-ub-violation*, the algorithm terminates after a finite number of steps, yielding a valid EMF-model graph G'.

Proof. Let TG be a type graph and G a graph typed by TG s.t. the conditions formulated above hold. We check termination and the satisfaction of the postconditions formulated in Section 3.4.2 for each step and then argue for overall termination.

Step (1). For every containment edge type $c =$ $\begin{tikzcd} \boxed{\text{A}} \arrow[r, tail, "\text{c}", "\text{m..n}"'] & \boxed{\text{B}} \end{tikzcd}$ in TG,

the set of *required-node-creation* rules contains a rule $\boxed{\text{:A}} \Rightarrow$ $\begin{tikzcd} \boxed{\text{:A}} \arrow[r, tail, "\text{:c}"] & \boxed{\text{:B}} \end{tikzcd}$

with a negative application condition *ac* that prevents the rule from being applied at nodes that already meet the lower bound *m*. But as long as it is not met, the rule is applicable. Since $m \neq *$ by definition, those rules can only be applied finitely often with the same match. Therefore the only possibility for Step 1 to not terminate is a containment-cycle with all lower bounds > 0, which is excluded since TG is f.f.i. (f.i. would even be enough). Thus, Step (1) terminates with a graph G' as result where *No-cont-lb-violation* holds.

Step (2). For every non-containment edge type $e = \boxed{\;A\;}\overset{\text{opr}\qquad\text{e}}{\underset{\text{k..l}\qquad\text{m..n}}{\text{————}}}\boxed{\;B\;}$ in TG, the set of *required-edge-insertion* rules contains a rule

$\boxed{\;:A\;}\;\boxed{\;:B\;} \Rightarrow \boxed{\;:A\;}\overset{\text{:opr}\qquad\text{:e}}{\text{————}}\boxed{\;:B\;}$ with an application condition ac ensuring that the rule can not be applied to nodes of types A and B if there is already an edge of type e between them or the lower bound of A or the upper bound of B (if it is an edge of opposite type) are already fulfilled. Because of these application conditions and since no new nodes are created during the execution, Step (2) terminates after finitely many steps as soon as every lower bound of non-containment relations is satisfied, or there is no suitable target node available for any unsatisfied non-containment relation (this is exactly the statement of *Post(2)*). Note, that the result G' of this step of the algorithm is independent of the order of execution and choice of matches in the following sense: The number of missing non-containment edges of a certain type is equal for every possible resulting graph. Only the places, where edges are missing, can be different.

Step (4). If a *required-edge-checking* rule was applicable in Step (3), a target node for a certain non-containment edge type is missing. Since we assume TG to be f.f.i., at least one possible target node can be created without violating any upper bounds. If no target node can be created directly, at least a possible container can. These two possibilities are reflected by *Post(4.1)*, *Pre(4.2)*, and *Post(4.2)*. As soon as such a node is created, Step (4) is finished and Step (5) calls Step (1) again.

Steps (3) and (6). *Required-edge checking* rules do not change the graph and only finitely many checks are performed, so Step (3) always terminates. Also, the setting of required empty attributes is finished after finitely many steps and does not change anything which influences the kind of validity we are discussing.

Overall termination. Let e be a critical edge and let $low(e) = m$ denote its lower bound. We assume TG to meet the (ACCOE). Suppose that e satisfies condition (a) of Def. 12. This implies $upp(e^{-1}) = *$. If it was not possible to create an edge of type e during Step (2) often enough, we have at some point created m appropriate target nodes for an edge of type e. Since each of those target nodes can serve as target for an unlimited number of edges of type e, after this point all edges of type e will be created during Step (2). Now suppose that e satisfies condition (b) of Def. 12. Then it is ensured that at some point a target node for e can always be created directly, because at least one possible container of a target node has an unlimited upper bound: If a target node for e had to be created often enough, the randomness of the creation of target nodes (if no direct container is available) makes for that. After that point, it is always possible to create the target node directly. Additionally, if more than one possible target node exists, Eq. 3.1 states that it makes no difference for the newly arising required containment and non-containment edges, which target is created. In summary, after finitely many iterations of the algorithm, the algorithm becomes deterministic: Every time Step (3) is called, the only missing edges are of types for which there exists only one way to create a target node, or for which it makes no difference which target node is created. Since we assume TG to be f.f.i., and we only create elements that need to exist in a valid model, the algorithm altogether terminates. The resulting graph needs to be a valid EMF-model graph since this is the only situation in which the algorithm stops. □ □

The above proofs give us almost immediately two further results:

Corollary 1 (Validity of Output). If the algorithm terminates, when applied to an EMF-model graph G, the result is a valid EMF-model graph G'.

Proof. The proofs of the previous theorems showed that we apply rules as long as there exist violations of upper or lower bounds. Thus there are only three possible outcomes when starting the algorithm: No termination, break of the algorithm in Step (4), because it is not possible to create a needed target node, or termination with a valid model. □ □

Corollary 2 (Hippocraticness of Algorithm). If G already is a valid EMF-model graph, application of the algorithm to G will result in G again.

Proof. By construction of the rules, as discussed in the proofs of Theorems 1 and 2, the applicability of rules that create or delete graph elements of valid models is prevented by suitable designed application conditions. □ □

3.5 Tooling

In this section, we give an overview of the architecture of the developed tool with the help of the UML component diagram and present its features as well. Furthermore, we present the developed test case specifications used for testing our techniques.

Our implemented techniques consist of two main components depicted in Figure 3.23 and described as follow:

1. **Meta2RR**: This meta-tool (developer tool) automates the derivation of a model transformation repair system from a given meta-model. The input is a domain meta-model typed by Ecore, and the output is an Eclipse plug-in containing the corresponding model transformation system. The generated model transformation system is formulated in Henshin. If the meta-model contains OCL constraints, they are not taken into account yet. The tool uses the meta-patterns which are implemented as Henshin rules typed by the Ecore meta-metamodel to detect and return the corresponding matches (the types) from the domain

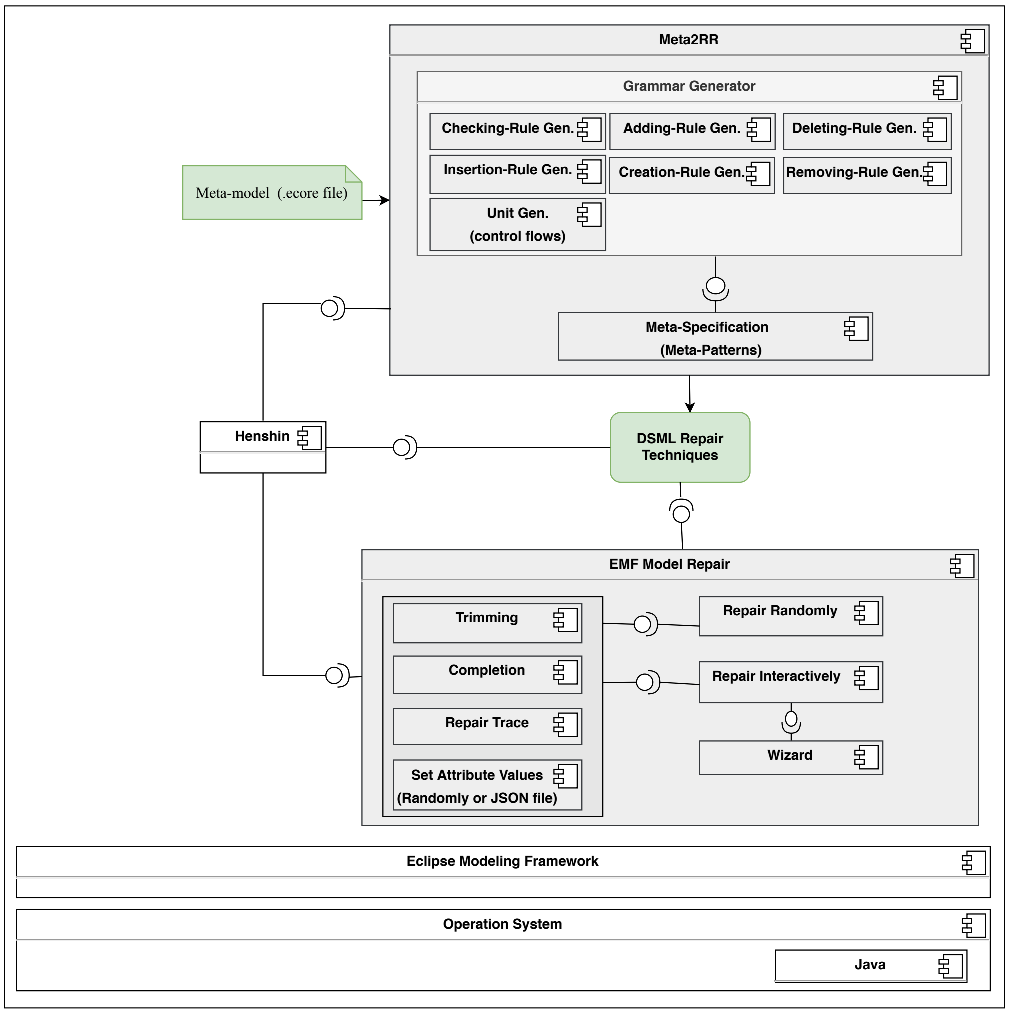


Figure 3.23: An overview of the architecture of our techniques for model repair

meta-model and to generate the corresponding repair rules w.r.t. the domain and the provided schemes, thereafter.

The meta-tool is implemented as an Eclipse plug-in and composes two main components as follow:

- *Grammar Generator*: It is responsible for applying the meta-patterns, finding their matches, and generating the different kinds of repair rules w.r.t. their schemes and wrapping them in proper units, after that. It contains several rules generators as follow:
  - *Creation-Rule Generator*: It is for deducing and generating the required-node-creation rules.
  - *Insertion-rule generator*: It is for deducing and generating the required-edge-insertion rules.

  - *Checking-rule generator*: It is for deducing and generating the checking rules such as required-edge-checking rules.
  - *Adding-Rule Generator*: It is for deducing and generating the rules which add elements without violating the upper bounds such as additional-node-creation rules.
  - *Deleting-rule generator*: It is for deducing and generating the rules for deleting nodes such as target-node-deleting rules
  - *Removing-Rule Generator*: It is for deducing and generating the rules for removing edges such as exceeding-edge-removing rules.
  - *Unit Generator*: It is for generating the corresponding control flows as transformation units being configured with the derived rules.
- *Meta-Specifications*: It contains the specification of the meta-patterns for deducing the different kinds of repair rules. They are designed as Henshin rules typed by the Ecore meta-metamodel.

2. **EMF Model Repair**: This tool (modeler tool) implements the repair algorithm presented in Section 3.3.1. This plug-in has two main inputs: The model transformation system being derived from the given meta-model and an EMF instance model (composing one node of the root type). To guarantee the termination of the algorithm, the meta-model should satisfy the defined preconditions, i.e., fully finitely instantiable. The main output is a consistent EMF instance model which can be opened, e.g., with the model editor.

   This tool is implemented as an Eclipse plug-in and consists of several components as follow:

   - *Trimming*: It implements the first part of the algorithm which deletes model elements to obey the upper bounds.
   - *Completion*: It implements the second part of the algorithm which adds model elements to fulfill the lower bounds.
   - *Set Attribute Values*: It is for assigning values for the attributes of primitive types. Assigning values can be done either randomly or using a set of values given as a JSON file.
   - *Repair Trace*: It records the model changes as a trace model. The trace model contains the information about the order of the rule applications and their names, and the matches to which the repair rules are applied.
   - *Repair Randomly.* It is for repairing models randomly without user interactions in a non-deterministic way.
   - *Repair Interactively*: It is for repairing models semi-automatically considering the user selections. Additionally, It composes the specifications of the wizards (the graphical user interfaces).

   For more information about the tool such as installation can be found at [141].

Features

Our developed tool (EMF model repair) provides the following features and functionalities:

Random or interactive mode. Since the algorithm is configured with the rule sets in a way that at any point of the algorithm where choices have to be made, they can be random, or interactive. This applies to choices of rules (repair actions), matches (model locations or targets), and attribute values. The tool is developed to repair EMF instance models *automatically* in two modes:

- *Randomly:* It repairs the whole model without user interactions.
- *Interactively:* It guides the user to repair the whole model by providing proper suggestions that lead to a consistent model.

Rule-based implementation of automatic random or interactive mode. Within each step of the algorithm, the state of the input model is analyzed by identifying the *applicable* rules which configure an algorithm step. Each of the identified repair rules (actions) aims at repairing the whole model. This is because of the design of the different kinds of rules. They are designed in a way that each successful application of a rule enhances (or at least preserves) the consistency of the model. These identified rules (actions) are suggested to the user who can decide which action should be first executed, or they can be executed randomly. Once an action is selected, the relevant model locations (elements), on which the selected action should be applied, are identified. The user can also choose a location to be repaired first. Once an action is applied, the model state is analyzed again, and so on. At the end of the repair process, the tool yields a consistent EMF instance model.

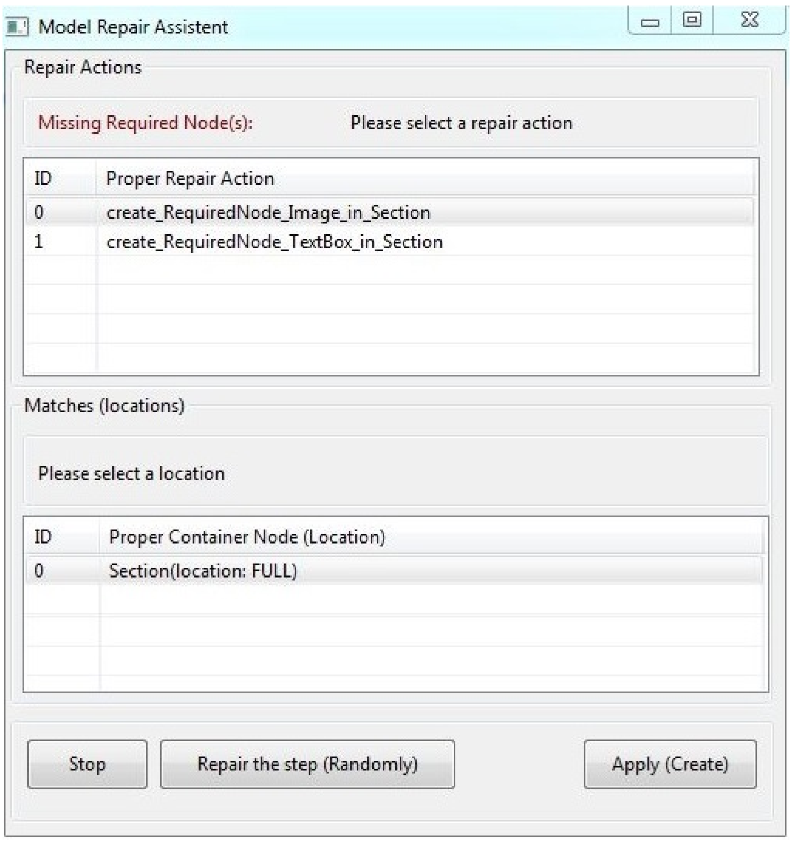
The tool assigns values to the required attributes of primitive types either randomly or by using a set of values provided as a JSON file.

In addition to guiding the user during the repair process by providing a list of proper repair actions which lead to a consistent model, the tool is developed to provide other two functionaries:

- The user can stop the repair process at any time. This would help him/her to design the model in different design times and to manage the needed data that has to be, e.g., deleted. Please note that the intermediate resulting model is an EMF instance model and the repair process can be triggered again to the resulting model at any time.
- After choosing some repair actions, the user can also finish the whole step of the algorithm randomly. This would help him/her to finish the repair process quickly by skipping choosing all uninteresting actions.

Wizard. Figure 3.24 presents a user-friendly repair interface which consists of three parts:

1. the first part provides proper repair actions for the current state of the model,


Model Repair Assistent

Repair Actions

Missing Required Node(s): Please select a repair action

ID	Proper Repair Action
0	create_RequiredNode_Image_in_Section
1	create_RequiredNode_TextBox_in_Section

Matches (locations)

Please select a location

ID	Proper Container Node (Location)
0	Section(location: FULL)

Figure 3.24: Assistant interface for model repair

2. the second part suggests the proper possible locations for a selected action, and
3. in the third part, the user can execute the selected action on the selected location, execute the whole step randomly or stop the repair process.

Ordered list of rules' applications. The tool records semantical model changes as an ordered list of rules' applications with their matches. This information may be beneficial, e.g., to resolve inconsistencies in distributed models. Instead of sending the whole model, this list could be used as a patch to complement changes for other distributed copies of the model. Table 3.6 presents the ordered list resulting from repairing the inconsistent model in Figure 3.3.

Correctness Test

In this following, we design the test case specifications of our algorithm and test it. We focus on testing the correctness of our algorithm for model trimming and completion by providing a vital set of input test models. To systematically test the correctness of our technique including the meta-model translation, and the algorithm of model trimming and completion, we consider the meta-model concepts and all kinds of inconsistencies that may occur in instance models. Thus, the test instance models have to be designed to cover all the algorithm functionalities w.r.t. possible meta-model concepts. To identify this set of test models, we first define its test case specifications. Thereafter, we perform the testing by providing at least one test case example for each test case specification.

Test Specifications

To systematically test our algorithm, we use an approach called the category-partition method [104]. Input parameters and environment conditions (i.e., the characteristics of the system state) that affect the functional behavior have to be identified by finding categories of information that characterize each parameter and environment condition. Each category is partitioned into distinct choices that include all possible kinds of values. All combinations of proper choices of the categories yield all the test case specifications.

Table 3.3 presents the categories we developed to characterize each parameter and environment condition for model trimming (Categories A-C) and then for model completion (Categories D-H).

In general, the Cartesian product of all choices of all categories produces the set of all the test case specifications. The number of choice combinations can be reduced since several choices cannot occur together. E.g., if an input model has all required edges, different forms of missing nodes do not have to be considered. The derived test case specifications help us to design the set of input test models and to choose the meta-model structures that have to be provided for testing the algorithm. Table 3.4 and Table 3.5 show 7 and 8 test case specifications that are needed to systematically test the model trimming and model completion, respectively. Each test case is specified by stating the corresponding requirements on the input instance model and meta-model.

Concrete test cases: For each test case specification, we designed manually a test case consisting of a suitable meta-model and a test model. Then, we repaired the designed test model using our tool and checked the validity of the output model using the *validator* tool integrated into the Eclipse Modeling Framework [135]. Moreover, we record which rules have been applied during the model repair. As a results, all the test models are repaired.

Example of Concrete Test Case

Although for each test case specification, we designed a concrete test case example. We present here a test case example for covering more than one test case specification.

Table 3.3: The category-partition table for the algorithm

Category A	**Number of exceeding nodes**
Choice 1	The input model has no exceeding node.
Choice 2	At least one exceeding node is existed in the input model.
Category B	**Number of exceeding edges**
Choice 1	The input model has no exceeding edge.
Choice 2	At least one exceeding edge is existed in the input model.
Category C	**The content of the exceeding node contains**
Choice 1	No content.
Choice 2	At least one direct child node.
Choice 3	At least one direct edge.
Choice 4	At least one indirect child node.
Choice 5	At least one indirect child node with at least one indirect edge.
Category D	**Number of required nodes**
Choice 1	The input model has all required nodes.
Choice 2	At least one required node is missing in the input model.
Category E	**Number of required edges**
Choice 1	The input model has all required edges.
Choice 2	At least one required edge is missing in the input model.
Category F	**Number of correlated nodes**
Choice 1	For all non-containment edge types, there are enough target nodes to fulfill their lower bounds.
Choice 2	There is at least one non-containment edge type for which there are not enough target nodes to fulfill its lower bound.
Category G	**The creation of missing correlated node**
Choice 1	For a given non-containment edge type whose lower bound is not fulfilled, the missing correlated node can be created directly. I.e., there is a node which can function as container for the missing node.
Choice 2	The missing correlated node has to be created indirectly, i.e., there is not any node which can function as container for the missing node. Hence, at least one further node has to be created before creating the missing node.
Category H	**Number of incoming containment type edges**
Choice 1	There is only one option to create the missing node. I.e., there is only one containment edge type to create at least one further intermediate node.
Choice 2	There are more than one option to create the missing node. I.e., there are more than one containment edge type to create at least one further intermediate node.

Table 3.4: Test case specifications for model trimming

Nr.	**Description**
1	The input model as no exceeding node and no exceeding edge.
2	The input model has no exceeding node but it has at least one exceeding edge.
3	The input model has no exceeding edge but it has at least one exceeding node and the exceeding node has no content.
4	The input model has no exceeding edge but it has at least one exceeding node and the exceeding node has at least one direct child node.
5	The input model has no exceeding edge but it has at least one exceeding node and the exceeding node has at least one edge.
6	The input model has no exceeding edge but it has at least one exceeding node and the exceeding node has at least one indirect child node.
7	The input model has no exceeding edge but it has at least one exceeding node and the exceeding node has at least one indirect child node with at least one edge.

Table 3.5: Test case specifications for model completion

Nr.	**Description**
1	The input model has all required nodes and edges.
2	At least one required node is missing while all required edges do exist.
3	The input model has all required nodes but at least one required edge is missing and there are enough target nodes.
4	The input model has all required nodes but at least one required edge is missing. There are not enough target nodes to connect with. The missing node can be created directly using only one containment type.
5	The input model has all required nodes but at least one required edge is missing. There are not enough target nodes to connect with. The missing node can be created directly using more than one containment type.
6	The input model has all required nodes but at least one required edge is missing. There are not enough target nodes to connect with. The missing node has to be created indirectly using only one containment type.
7	The input model has all required nodes but at least one required edge is missing. There are not enough target nodes to connect with. The missing node has to be created indirectly using more than one containment type.
8	At least one required node and at least one required edge are missing in the input model.

Input meta-model: The *Webpage* meta-model depicted in Figure 3.2.

Input model: We use the input model depicted in Figure 3.3 (the solid-line elements) as an input test model that covers several test case specifications, namely the test case specification 7 for model trimming and the test case specification 2 for model completion. Here, an exceeding node of type *Footer* do exist while two required nodes are missing: one of type *Body* and one of type *Anchor*. Please note that the resulting model state after applying Step (1) of the first iteration of model completion can be considered as a test case example for the test case specification 6 for model completion.

Expected model repair: A consistent model, which is similar to the one, is illustrated in Figure 3.3. An exceeding node of type *Footer* with its contents shall be deleted. The required nodes of type *Body* and *Anchor* should be created. The required edge of type *target* should be fulfilled and thus the container node of the missing correlated node of type *Section* should be added and two nodes of type *DisplayElement* shall be created as well.

Output model: Running the algorithm, the model is trimmed and completed as depicted in Figure 3.3. A *Footer*-node with its contents is deleted. A *Body*-node and an *Anchor*-node are created. A *Section*-node with two children is created: a *TextBox*-node and an *Image*-node. The *target*-link points to a *TextBox*-node. Attribute values are set according to default values. The expected actions have taken place.

Table 3.6: The ordered list of rules' applications

Nr.	Name of the successful rule application	On Match
1	Check_ExceedingNode_Footer_in_Webpage	:Webpage(pagex)
2	Remove_Edge_active_to_URL	:HyperLabel(label3)
3	Delete_URL_in_HyperLabel_in_Footer	:Footer(location)
4	Delete_HyperLable_in_Footer	:Footer(location)
5	Delete_Footer_in_Webpage	:Footer(location)
6	Create_RequiredNode_Body_in_Webpage	:Webpage(pagex)
7	Create_RequiredNode_Anchor_in_NavBar	:NavBar(pink)
8	Check_RequiredEdge_target_to_DispalyElement	:Anchor(moreinfo)
9	Add_AdditionalNode_Section_in_Body	:Body(white)
10	Create_RequiredNode_TextBox_in_Section	:Section(full)
11	Create_RequiredNode_Image_in_Section	:Section(full)
12	Insert_RequiredEdge_target_to_DisplayElement	:Anchor(moreinfo), :TextBox

Output list of rules' applications: Table 3.6 presents the rules which are successfully applied to repair the input model using our tool:

- The first rule dedicates that there is an exceeding node of type *Footer* in a node of type *Webpage*.
- The rules 2-5 delete a selected *Footer*-node with its contents in bottom-top way. The *Footer*-node can be selected randomly or by the user, and then the selected node is passed as a rule parameter.
- The rules 7 and 8 create the missing required nodes, a *Body*-node and an *Anchor*-node in the *Webpage*-node and the *NavBar*-node, respectively.

- The rule 8 dedicates that there is a missing required edge of type *target* to a node of type *DisplayElement*.
- The rule 9 adds a *Section*-node in a *Body*-node.
- The rules 10-11 create the missing required nodes in the added *Section*-node.
- The rule 12 inserts the missing required edge *target* from the *Anchor*-node to an *DisplayElement*-node.

3.6 Evaluation

In this section, we present an experiment showing the performance and the scalability of our tool being applied to inconsistent models of varying sizes composing a large number of violations. This experiment intends to answer the following research questions (RQs):

RQs: How much time does the tool need to repair large models composing a large number of violations? How many actions are applied to repair the whole models?

3.6.1 Scalability Experiment

To answer RQs, we carried out the following experiment:

Experiment set-up. We consider different size categories of input models, namely 1 000, 3 000, 5 000, 8 000 and 10 000 elements (nodes and edges). For each size category, we generated 100 inconsistent models.

The inconsistent models are generated by randomly adding elements (nodes and edges) to root nodes. This is done by randomly applying rules which adds elements without considering the multiplicities. These rules are derived for each meta-model type, and each one is designed to add one element in a rule application. After generating random inconsistent models, we checked the number of violations using the diagnostic of the EMF validator [34]. The number of violations was ranged from 145 to 1511 violations on average.

We applied our tool to the input models and calculated the times needed to repair the models, the number of trimmed and created elements, and the number of applied repair actions. We used the webpage meta-model, whose size is 44 elements (13 nodes, 14 edges, 17 attributes) and it has 18 bounded cardinalities.

The evaluation was performed with a desktop PC, Intel Core i7, 16 GB RAM, Windows 7 x64, Eclipse Oxygen with the default settings, e.g., the heap size is limited to 1 GB, Henshin 1.4.

Table 3.7: Average run time (in seconds) of repairing 500 inconsistent models of varying size up to 10 000 elements and each model contains up to 1511 violations

Size Categories of Inconsistent Models	**1000**	**3000**	**5000**	**8000**	**10 000**
Number of Inconsistent Models	100	100	100	100	100
Number of Violations	145	442	737	1207	1511
Number of Applied Repair Actions	590	1789	2983	4885	6116
Number of Trimmed Elements	301	927	1562	2559	3204
Number of Created Elements	289	862	1421	2326	2912
Repairing Time in Seconds	0.32	2.01	7.07	28.54	47.27

Table 3.8: Summary of the meta-models

Meta-Model	**Size**	**Nodes**	**Edges**	**Attributes**	**Bounded Card.**
GraphML	39	14	13	12	23
Statechart	51	16	28	7	25
LaTeX	71	31	39	1	55
Bugzilla	63	16	8	39	34
Car Rental	30	8	8	14	14
Web Model	33	13	12	8	9
CoreWareHouse	33	15	14	4	12
Class Model	34	12	17	5	16
Average	44	16	17	11	24

Experiment results. Table 3.7 shows the result of our evaluation. Our tool repairs 145 to 1511 violations in models of size 1000 to 10 000 elements in 0.3 to 47 seconds on average. The number of the applied repair actions are ranged from 590 to 6116 repair actions. All the input models are repaired, and the results show that our technique is fast enough to be usable in practice. Further experiments presenting that our EMF repair technique is scalable is reported in Chapter 4 in which the EMF repair is used to generate large consistent models.

3.6.2 Threats to Validity

Although we considered 500 inconsistent models of different sizes being generated randomly, they are instances of one meta-model. However, we are pretty convinced that the number of violations and the number of applied repair actions affect the performance. Therefore, we provided large models with a large number of violations and calculated the number of actions needed to repair the models.

However, we additionally applied the tool to 8 meta-models taken from the literature, projects and use cases, namely the Statechart of the Magicdraw [66], web model [16], car rental and class model [5], Bugzilla, Latex, Warehouse and, Graph Modeling Language (GML) [6]. Table 3.8 presents the information about the used meta-models.

Our tool repaired 2592 violations in models of size 10 000 in 51 seconds on average. The used meta-models compose most of the meta-model concepts such as containment and non-containment relations with different properties (e.g., loop and opposite), classes, abstract classes, interfaces and enumerations, and supporting inheritance and all multiplicity patterns.

3.7 Discussion

In this section, we discuss the usefulness of the approach in more general settings.

3.7.1 User Interaction

Rule systems have the advantage to allow incremental processes that can be controlled by user interaction. During each step of the algorithm, it is easily possible to enable the user to choose which of the applicable rules to execute at which match. This includes the resolution of the kind of non-determinism that was already mentioned at the end of Section 3.3.1. In this way, the user is enabled to repair an instance in a more meaningful fashion.

In our developed tool, we restrict user interaction to choices that can be made during a step of the algorithm but do not allow changes of the overall control flow. The steps are run through in the designated order and, the rules of each step have to be executed as long as there are applicable rules. The applicable rules and their matches serve as repair choices. The suggested choices of each step aim at repairing the whole model. This semi-interactivity guides the user towards a consistent instance and ensures the integrity of the approach: If user interaction is just used to reduce the amount of non-determinism in the repair algorithm, the correctness (and termination) proofs for the algorithm still hold.

The control flow of the algorithm can be stopped at any time, and the resulting intermediate model is an EMF instance model that can be opened and displayed by the EMF editor. If arbitrary model editing steps are allowed (out of the repair process and out of the model editor), it may happen that new inconsistencies are introduced, and the algorithm does not terminate anymore.

3.7.2 Moving Instead of Trimming

During the first part of our algorithm, we resolve inconsistencies w.r.t. upper bounds. We do this by deletion of supernumerous elements. This is a general solution. In some cases, moving the elements into other containers (sources) without violating their upper bounds would be possible. But this does not always work: For example, the consistency requirements for rules which move cycle-capable containment edges are quite strict [14]. So it may not be possible to generally derive a rule that would be useful in a concrete situation. And more importantly, because of finite upper bounds it is possible that there is a maximal number of objects of a certain node type which are allowed to exists in EMF-model graphs. Thus, no graph that contains more nodes for one type n than allowed can be repaired without deleting.

But as long as this is considered, the moving of supernumerous elements into other already existing containers (sources) or newly created ones (if possible) does not change the validity of Theorem 1 and therefore neither that of Theorem 2. However, in Sections A.1 and A.2, we present the schemes of moving rules for non cycle-capable containment and non-containment edges and their meta patterns as well as concrete examples.

3.7.3 Finitely Instantiable Versus Fully Finitely Instantiable Meta-Models

While deleting during the trimming part of the algorithm, we intended not to delete during the second part at the cost of not being able to repair every instance of finitely instantiable type graphs TG: Deletion would have to take place in Step (4) of the algorithm if no target node can be created. In this case, the bounds of all containment edges would be satisfied, but the lower bound of at least one non-containment edge violated. This means that multiplicities of non-containment edges would not only demand or forbid the existence of non-containment edges but additionally impose constraints on the number of certain types of nodes. Here, it may be more advisable to adapt the multiplicities of the meta-model than to loose information by deletion. Furthermore, there are f.i. meta-models for which only a very small number of consistent models exist. Then, repairing instances means the deletion of almost everything and the creation of one of the few consistent models.

For a type graph TG, which is not fully finitely instantiable, the algorithm may break during Step 4. In this case, getting the match of the check rule used in Step 3 aims at identifying for which edge it was not possible to create a target node. This information could be useful, e.g., to refactor the multiplicities of the corresponding type graph so that a given model can be completed, or to maybe delete the source node of that required edge.

An additional alternative is the creation of a further root node. There exist instances of finitely instantiable meta-models, which can be repaired – without deletion during the second part – by creating more than one, but finitely many root nodes.

To summarize, when applying the considered approach to finitely, but not fully finitely instantiable meta-models, the following possibilities exist:

(a) Instance gets repaired.

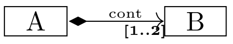
(b) Algorithm breaks in Step 4 because a needed target node cannot be created. A solution might be achieved by creating finitely many new nodes of root type, or there is no termination, even when creating new nodes of root type.

(c) No termination.

Characterizing these situations by features of the type graph or the instance which is to be repaired is a future work.

3.7.4 Supporting a Set of OCL Constraints

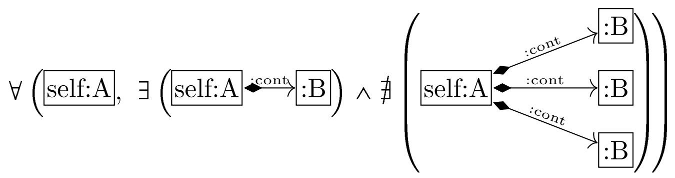
There are some of OCL constraints which can be straightforward supported by our approach. These OCL constraints are semantically equivalent to the lower and upper bound constraints, i.e., they can be represented as graph constraints of form $\forall(P, \exists C_1)$ or $\forall(P, \nexists C_2)$ — P, C_1 and C_2 are graphs such that P is a sub-graph of C_1 and of C_2.

For example, the multiplicity bound **[1..2]** of a containment edge type 

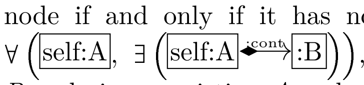
and the following OCL constraint:

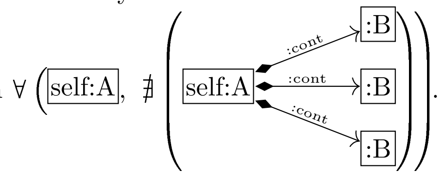
context *A* ***inv****: self.cont → notEmpty() and self.cont → size()<3*

are semantically equivalent to each other and they can be represented as graph constraints as follows.



It means that there is at least one B-node in an A-node, and there are at most two B-nodes in an A-node. The first part represents the lower bound constraint of value 1 and the OCl expression *self.cont → notEmpty()* whereas the second part represents the upper bound constraint of value 2 and the OCL expression *self.cont → size()<3*.

Our algorithm can similarly be configured with rules being derived from the graph patterns as follows: From a graph pattern of form $\forall(P, \exists C_1)$, rules of kind *required-node-creation* can be derived to configure the Step (1) of the algorithm and from a graph pattern of form $\forall(P, \nexists C_2)$ rules of kinds *finding-exceeding-node*, *exceeding-node-deleting* and *additional-node-creation* can be derived to configure the Step (b), the Step (c) and the Step (4) of the algorithm, respectively. Regarding to our example, a *required-node-creation* rule, which creates a B-node in an existing A-node if and only if it has no B-node, will be derived from the graph pattern 

, and an *exceeding-node-deleting* rule, which deletes a B-node in an existing A-node if and only if it has more than two B-nodes, will be derived from the graph pattern 

.

However, not all constraints of form $\forall(P, \exists C_1)$ or $\forall(P, \nexists C_2)$ can be directly supported. They need to obey some criteria such that there are no conflicts among them, e.g., if there is a conjunction between a positive and a negative graph constraint of form $\forall(P, \exists(C_1) \wedge \nexists(C_2))$, C_1 should be a sub-graph of C_2. Figure 3.25 presents the core concept of our future work for supporting a set of OCL constraints.

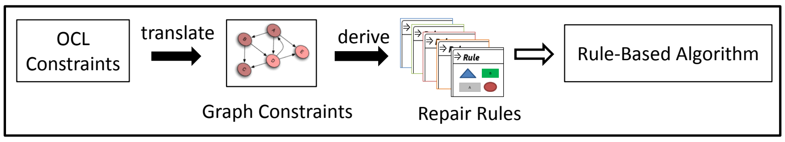


Figure 3.25: EMF Repair supporting a set of OCL constraints

Moreover, the repair algorithm can straightforwardly support negative constraints as done previously. I.e., we derive rules which delete supernumerous elements w.r.t. the negative constraints, and thereafter, we update the rules which create model elements in a way that they do not violate the negative constraints. Here, we can use the tool *OCL2AC* (see Chapter 5) to update the rules automatically. Furthermore, translating OCL constraints to graph patterns [12, 113] and further to application conditions of rules is promising to achieve an automated interactive model repair approach for meta-models with OCL constraints.

3.8 Related Work

In the following, we consider related work w.r.t. model repair and rule generation. We relate our approach to repair approaches first. After that, the rule generation is considered.

Model repair is a long-term research challenge. According to the recent survey on model repair [79], the number and wide variety of existing approaches make a detailed comparison with all of them infeasible. However, the existing model repair techniques can be mainly categorized into syntactic and rule-based approaches on the one hand, and search-based and logic-based approaches, on the other hand. Table 3.9 shows an overview of different main features of the approaches. Note, most of the existing repair techniques based on a solver or a generator for generating valid models. A general overview of them can be found in Section 4.6, Table 4.6.

Syntactic and rule-based approaches. In [95, 94], the authors provide a syntactic approach for generating interactive repairs from full first-order logic formulas that constrain UML documents. The user can choose from automatically generated repair actions when an inconsistency occurs. Similarly, Egyed et al. [37, 38] describe a rule-based technique for automatically generating a set of concrete changes for fixing inconsistencies at different locations in UML models and providing information about the impact of each change on all consistency rules. The designer is not required to introduce new model elements, i.e., the approach ignores the creation of new model elements as choices for fixing inconsistencies. In [111], Rabbi et al. propose a formal approach (with prototype tooling) to support software designers in completing partial models. Their approach is based on rule-based model rewriting, which supports both addition and deletion of model elements.

In all these approaches, inconsistencies can be considered by the user one after the other; possible negative side effects are not taken into consideration. It is up to the user to find a way to a valid model (if any). Moreover, they are not shown to be fully consistent.

Taentzer et al. [140] present a prototype based on graph transformation theory for change-preserving model repair. From given edit operations taken from the edit history of two versions of consistent models and consistency-preserving operations, they calculate repair operations. These repair operations can be detected to repair the model resulting from applying an editing operation. The performance of detecting repair operations is shown to be fast. They consider only the conformance to the meta-model, and there is no available tool to perform the whole process automatically.

Search-based and logic-based approaches. A search-based repair engine starts with an inconsistent model state and tries to find a sequence of change actions that leads to a valid model. Another approach is model finding, using a constraint solver to calculate a valid model. Many approaches such as [129, 2, 59, 78] provide support for automatic inconsistency resolution from a logical perspective. All these approaches provide automatic model completion; however, they may easily run into scalability problems (as stated in [79]). Since the input model is always translated into a logical model, the performance depends on the model size. Badger [110] is a search-based tool which uses a regression planning algorithm to find resolution plans. It can take a variety of parameters to let the search process be configured by the user to a certain extent. The meta-model has to be manually specified as a set of logic facts. The authors argue that their generation of resolution plans is reasonably fast for resolving several inconsistencies (70 inconsistencies) of a consistency type in large models. However, they do not show the time needed to apply resolution plans to resolve a large number of violations of different types so that the whole model is valid. Moreover, there is no correctness proof. Although being rule-based, the refinement approach in [117] basically translates a set of rules with complex application conditions to a logical model. Schneider et al. [121] provide a strategy relying on an existing graph generation algorithm for graph constraints implemented in AutoGraph. The input graph is represented as a constraint and then it is given to the AutoGraph with other constraints to generate a consistent graph. However, if the input graph is inconsistent, no consistent graph can be found. Moreover, they provide another strategy similar to the one presented by Taentzer et al. [140]. But, they do not require the user to specify consistency preserving operations since they derive repairs using constraint solving techniques directly from the graph constraints. In both strategies, no implementation and evaluation are available now. Additionally, the AutoGraph cannot support EMF constraints since some of them are not first order.

Our approach is designed for integrating the best of both worlds: It is a rule-based approach; therefore, it is easy to allow user interaction during the repair process (in contrast to search-based and logic-based approaches), and it does not leave the resolution strategy completely to the modeler as in pure rule-based approaches. Instead, it guides the modeler in an automatic interactive way to repair the whole model. Our approach yields valid EMF instance models, which can be opened by the model editor. How all the EMF constraints are specified and how valid EMF instance models are constructed are not clearly discussed by most existing approaches. Additionally, we provide the correctness proof of our approach. On the downside, our approach cannot yet handle OCL constraints being covered by most of the approaches mentioned above. It is promising to translate OCL constraints to graph patterns [12, 113] functioning as application conditions of rules and thereby extending the automated interactive model repair approach. Besides, in Section 3.7.4, we discussed an idea about how to support a set of OCL constraints.

Rule generation. In [66], model transformation rules are generated from a given meta-model as well. The main difference to our work is that consistency-preserving rules are generated there while we generate repair rules allowing temporarily inconsistent models w.r.t the multiplicities. Hence, rules are generated for different purposes: There, consistency-preserving rules are generated from restricted meta-models to recognize consistent edit steps, while we generate repair rules to configure our model repair algorithm yielding consistent EMF models as results. Our rules are more flexible (less restricted), can be applied to inconsistent models and derived from unrestricted meta-models.

Table 3.9: Overview of some different main features of the repair approaches.

Approach	Repair whole model	Interactive	Scalable	EMF-compliant	wf-contraints	
					Multiplicities	OCL constraints
Constraint solvers	+	−	−	○/?	+	+
Search-based	+	−	○	○/?	+	+
Rule-based	−	+	?	○	+	○
Our approach	+	+	+	+	+	−

3.9 Conclusion

In this work, we presented a rule-based approach for repairing EMF instance models in two modes: (1) *randomly*, i.e., fully automatically without user interactions or (2) *semi-automatically (interactively)*, i.e., guiding the user to repair the whole model and thereby resolving all the cardinalities violations. The resulting models are consistent EMF models, i.e., consistent instance models of meta-models with multiplicities conforming to the Eclipse Modeling Framework (EMF). A rule-based algorithm of model trimming and completion is developed, and different sets of model transformation rules (repair actions) are designed in a proper way to configure the algorithm according to the given language. To automatically derive them, a catalog of several meta-patterns is specified as well. Using the theory of algebraic graph transformation as a formal background, we proved the correctness of the algorithm and gave conditions under which the algorithm is guaranteed to terminate. Additionally, we discussed its features and usefulness in more general settings, such as supporting a set of OCL constraints. Two Eclipse plug-ins are developed: (1) *Meta2RR (meta-tool)* to translate meta-models to model transformation repair systems automatically, and (2) *EMF Model Repair (modeler tool)* to repair corresponding EMF instance models. Using the EMF repair tool, the user can repair EMF models not only randomly and semi-automatically but also stop the repair process at any time. The resulting model is an EMF instance model which can be opened, e.g., by the model editor. Furthermore, a systematic test is performed to show the correctness of our technique and we presented that models of size 10 000 elements with 1511 violations are repaired in 47 seconds on average.

We plan to extend our approach to support a certain kind of OCL constraints (e.g., the negative constraints) as well. Moreover, translating OCL constraints to graph patterns [12, 113] and further to application conditions of rules is promising to achieve an automated interactive model repair approach for meta-models with OCL constraints. Additionally, we want to use the work to generate models w.r.t. given (user) specifications for testing purposes. Furthermore, we intend to automatically generate editing operations from a given meta-model for different purposes by using and combining the different kinds of derived repair rules.

4 Rule-based Generator of Consistent EMF Models

This chapter shares material with the accepted paper at FASE'20 "Generating Large EMF Models Efficiently: A Rule-Based, Configurable Approach" Nassar et al [88].

Efficiently generating large instance models or large sets of diverse instance models of a given meta-model is a highly important task in model-driven engineering. Depending on a chosen application scenario, a model generator has to fulfill different requirements: As a modeling language is usually defined by a meta-model, all generated models are expected to *conform to their meta-models*. For performance tests of model-driven engineering techniques, the efficient generation of *large* models should be supported. When generating several models, the resulting set of models should show some *diversity*. *Interactive model generation* may help in producing relevant models. In this work, we present a rule-based, configurable approach to automate model generation which addresses the stated requirements. Our model generator produces consistent instance models of meta-models with multiplicities conforming to the Eclipse Modeling Framework (EMF). An evaluation of the model generator shows that large EMF models (with half a million elements) can be produced. Since the model generation is rule-based, it can be configured beforehand or during the generation process to produce sets of models that are diverse to a certain extent. Moreover, an evaluation of our tool shows a speed-up of several orders of magnitude compared to other state-of-the-art instance generation tools for EMF models.

4.1 Introduction

The need for the automated generation of instance models grows with the steady increase of domains and topics to which model-driven engineering (MDE) is applied. In particular, there is a growing need for large instances of a given meta-model [70, 118]. They are needed in various applications like model transformation testing [8, 44], benchmarking model queries and transformations [137, 11], model-driven search and optimization [150, 15], or validating the suitability of MDE tools to deal with large input models [122, 1, 47, 31]. As most of the available MDE tools are based on the Eclipse Modeling Framework (EMF) [135], instances should be conformant to EMF.

Depending on the chosen application scenario, a model generator has to fulfill different requirements: As a modeling language is usually defined by a meta-model, all generated models are expected to *conform to their meta-models.* For performance tests of model-driven engineering techniques, the efficient generation of *large* models should be supported. When several models are generated, they should show some diversity. *Interactive model generation* may help in producing relevant models. While there are several tools and approaches to instance model generation in the literature, e.g. [81, 86, 125, 128, 138], we are not aware of any tool satisfying all the requirements stated above. Two extreme approaches are the following: The approach in [86] is very fast but does not address any modeling framework and provides very few guarantees concerning the properties of the generated output models. As EMF has developed to the de-facto standard for modeling in MDE, respecting the EMF constraints is crucial to guarantee the usability of the resulting models in practice for processing them by other tools, e.g., for opening them in standard editors. On the contrary, solver-based approaches such as [81, 128, 138] provide high guarantees by generating instance models that even conform to additional well-formedness constraints (expressed in, e.g., OCL [99]), but they suffer from severe scalability issues.

We suggest finding a good trade-off between having a scalable generation process for models and generating well-formed models. In this work, we propose a rule-based approach to the generation of models which has the following distinguishing features:

- To guarantee interchangeability, generated models conform to the standards of EMF. In particular, this means that the containment structure of a generated model forms a tree.
- Generated models exhibit a basic consistency in the sense that they conform to the structure and the multiplicities specified by the meta-model.
- The generation of models can be configured to obtain models that are diverse to a certain extent.
- The implementation is efficient in the sense that instance models with several hundred thousand elements can be generated.
- The approach is meta-model agnostic and customizable to a given domain-specific modeling language (DSML) in a fully automated way.
- The generation can be started by *manipulating existing models.* This facility not only supports the incremental development of models at different times but also is promising for accelerating the generation process.
- It is possible to generate models in a batch mode or interactively to somewhat guide the generation process towards relevant models. User interaction includes the setting of seed models as well as interactively choosing between alternative generation strategies.

Our rule-based approach to model generation consists of two main tasks:

1. The meta-model of a given modeling language is translated into a rule-based model transformation system (MTS) containing rules for model generation.
2. These rules are consecutively applied to generate instance models. This generation process may be further configured by the user. Especially, a potentially inconsistent model may be used as a seed for generating consistent models.

Our approach is implemented in two Eclipse plug-ins:

1. A meta-tool, called *Meta2GR*, automatically derives the MTS from a given meta-model.
2. A second plug-in, called *EMF Model Generator*, is automatically configured with the resulting MTS. A modeler uses the configured model generator, which takes additional user specifications and an optional seed EMF model as inputs and generates a consistent EMF model.

We argue for the soundness of our approach and evaluate its scalability by generating large, consistent EMF models (up to half a million elements). Furthermore, we show how to generate a set of models that are diverse to some extent.

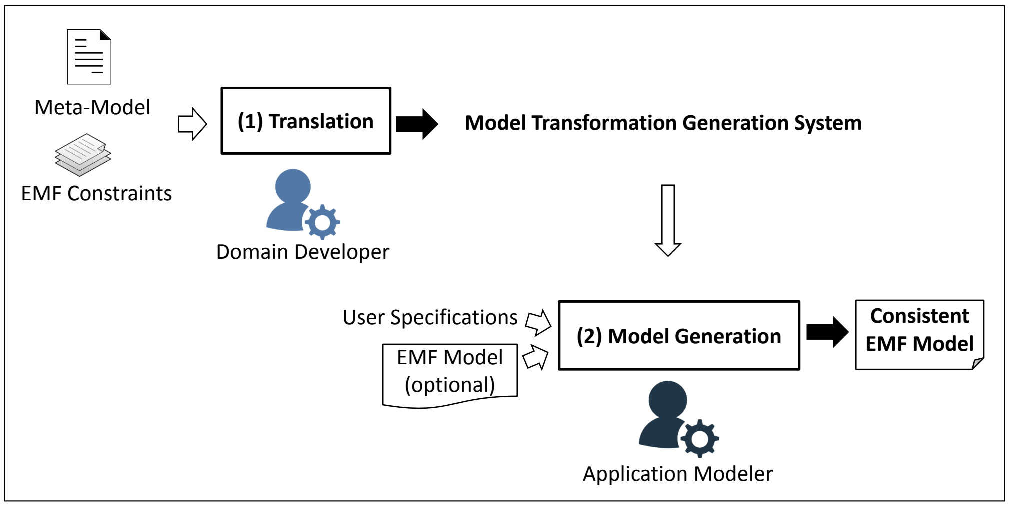


Figure 4.1: Model generation tasks

The contributions of this work are the following:

1. A rule-based model generator, different kinds of generation rules, and several configuration strategies for supporting user specifications.

2. Meta-specifications to derive and configure the model generator to a given meta-model and to configure the user specifications.
3. An Eclipse-based tool (a meta-tool), called *Meta2GR*, is developed. The domain developer uses this tool to automate the derivation of the model transformation system from any given meta-model, i.e., to generate the domain-specific generation technique.
4. A model generation tool (a DSML tool), called *EMF Model Generator*, is developed as an Eclipse plug-in. It takes user specifications and an EMF model (optional) as inputs, uses the generated domain-specific generation technique, and generates a consistent EMF model. The application modeler usually uses this tool.
5. We discuss the soundness of our approach and then evaluate its scalability by generating consistent EMF models with half a million elements (nodes and edges). Furthermore, we perform a scalability comparison with the state-of-the-art instance generator. Moreover, we show the diversity of the generated models. The related work is discussed as well.

The rest of this chapter is structured as follows: We first introduce our running example in Section 4.2. Section 4.3 develops our approach showing the generation strategies, the generated rules, and their meta-patterns, and discusses its soundness as well. In Section 4.4, we present our tooling that is subsequently evaluated in Section 4.5 showing its scalability and the diversity of the generated models. We discuss related work in detail in Section 4.6. Section 4.7 concludes and points to some future work.

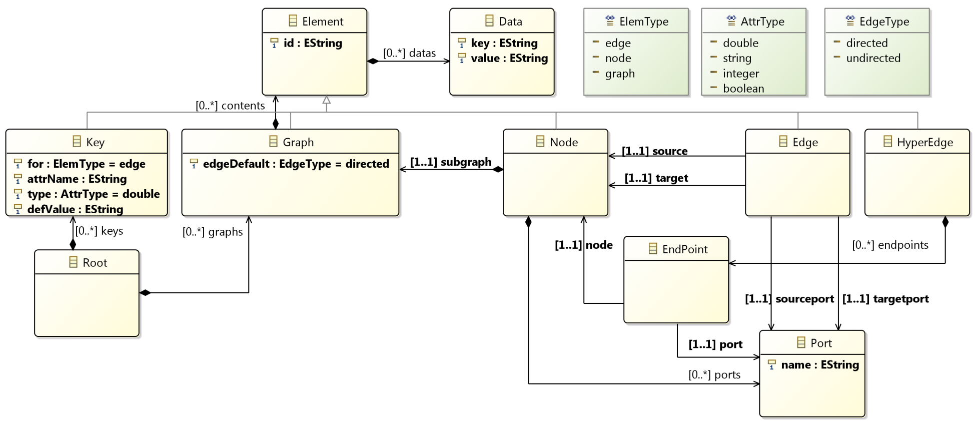


Figure 4.2: Excerpt of the meta-model of Graph ML

4.2 Running Example

This section presents our running example. As a running example we use a simplified version of the meta-model of GraphML as provided by [6] shown in Figure 4.2. GraphML [17] is a file format for different kinds of graphs (directed, hierarchical, hypergraphs, etc.) and separates the graph structure from additional data. We use this example to illustrate how our rule-based approach generates instance models of this meta-model.

4.3 Rule-based Model Generation

In this section, we first present our basic approach to the generation of consistent EMF models. We then give an overview of the kinds of derived rules, introduce four parametrization strategies, and show the possibility of user-interaction. We end with discussing formal guarantees and limitations.

4.3.1 Overall Approach

Our overall approach to instance generation is depicted in Fig. 4.3. The fundamental idea behind our approach is to base model generation as far as possible on rule-based model repair using the tool EMF Repair [93]. All rules needed to perform model generation steps are automatically derived from the given meta-model by the meta-tool *Meta2GR*. If a non-empty seed model is given, the model generation process starts with checking it for upper bound violations and potentially trimming it using EMF Repair (*model trimming*). Thereafter, the EMF model is extended with object nodes and references without violating upper bounds using the rules derived by *Meta2GR* (*model increase*). The resulting model shall meet user specifications w.r.t. its size which will be discussed in more detail in Section 4.3.3 below. In the next step, the EMF model is completed to a consistent EMF model, again using EMF Repair (*model completion*). As this repair process adds elements only, the user specifications are still met by the resulting model. Moreover, the result is guaranteed to be a consistent EMF model [92]. EMF Repair is also used to set attribute values, either randomly or using user input which is provided in a JSON-file.

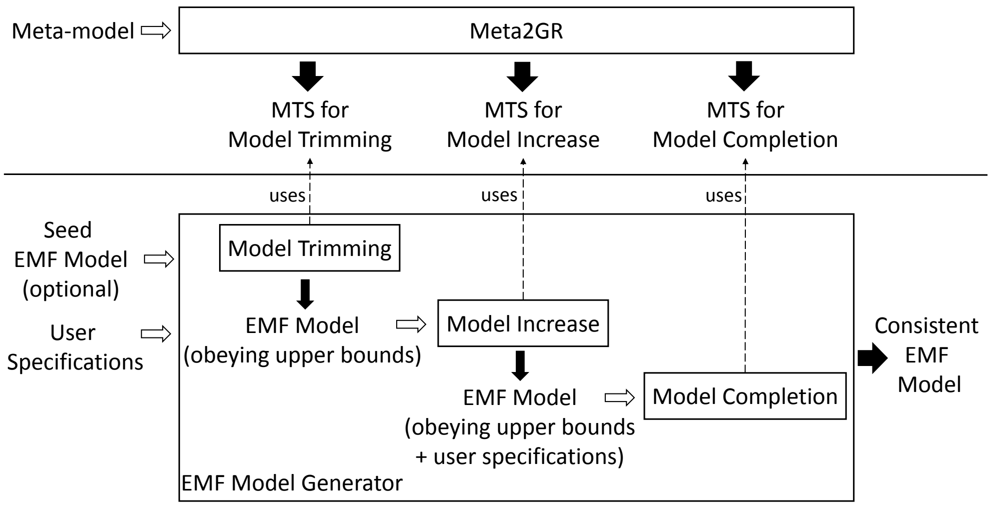


Figure 4.3: Rule-based EMF Model Generator

4.3.2 Generated Rules

Table 4.1: Overview of used kinds of rules

Role	**Kind**	**Semantics**
Node creation	Additional-node-creation rules	Insert a node of a certain type inside of one of its direct containers
	Transitive-node-creation rules	Add a node of a certain type inside of one of its transitive containers
Edge insertion	Additional-edge-insertion rules	Insert an edge of non-containment type between two nodes
Check	Additional-edge-checking rules	Check if possible source and target nodes for an edge of a certain type exist

Given a meta-model, we first derive certain kinds of rules that are then used in the different programs for the generation of an EMF model. Table 4.1 gives an overview of the kinds of derived rules. The derived rules perform three different roles: (i) creation of nodes, (ii) insertion of non-containment edges, and (iii) checking for the existence of source or target nodes for an edge of a certain type. All rules that create elements, i.e., the rules with the role (i) or (ii) are generated with NACs that ensure that the according rules are applicable if and only if they will not introduce an upper bound violation. Moreover, they are designed in such a way that they are not able to introduce violations of the EMF constraints such that accordance with these constraints is preserved throughout their application. Please see Section 3.3.2 to get more information about how the rules are designed to deal with and preserve the EMF constraints.

Node creation (i) is performed by two different kinds of rules: For each containment edge type in a given meta-model an *additional-node-creation rule* is derived. A general scheme for this kind of rule is depicted in Figure 4.4. Such a rule matches the source node of this edge and creates an edge of the chosen type together with an instance of the contained node (if other nodes inherit from the target node, a rule is derived for each of them as well). It is equipped with a NAC that prevents the application of the rule at a node that already has the maximal number of outgoing edges of the respective type. The rule *add_in_Node_a_Port* in Figure 4.8 is an example of a concrete instance derived for the containment edge type ports. It does not have a NAC since the upper bound of ports is unlimited.

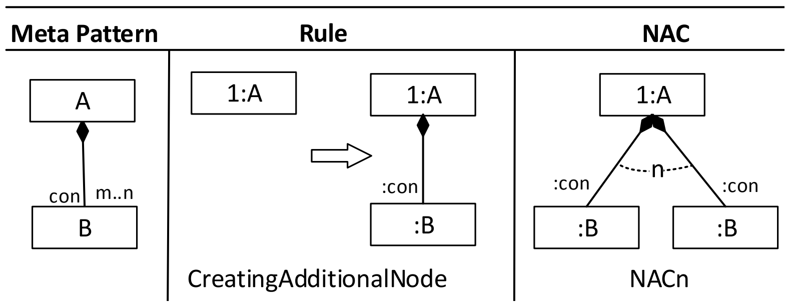


Figure 4.4: Rule scheme for *additional-node-creation rules*

Transitive-node-creation rules are similar. An example scheme for length 2 for this kind of rule is depicted in Figure 4.5. For every concrete node type in the meta-model, every possible incoming path over containment edges is computed such that each containment type occurs maximally once. For each such path, a rule is derived that matches the node where this path starts and creates the rest of this path. Again, the rules are equipped with a NAC ensuring that no upper bound violation can be introduced. The lower part of Figure 4.8 depicts all transitive-node-creation rules that are derived for the type port. Only one of the rules has to be equipped with a NAC as only the edge type subgraph has an upper bound (of 1).

To insert edges (ii), *additional-edge-insertion rules* are generated. The general scheme for this kind of rules is depicted in Figure 4.6. For each non-containment edge type, a rule is derived that matches the source and the target node of this edge and creates an edge of the according type. Again, a NAC prevents that an upper bound is violated (NACn) and a second NAC prevents that parallel edges are introduced (NACp). If the edge is an EOpposite edge, the opposite edge is created as well and its upper bound considered accordingly (NACl). A concrete example for the edge type targetport is

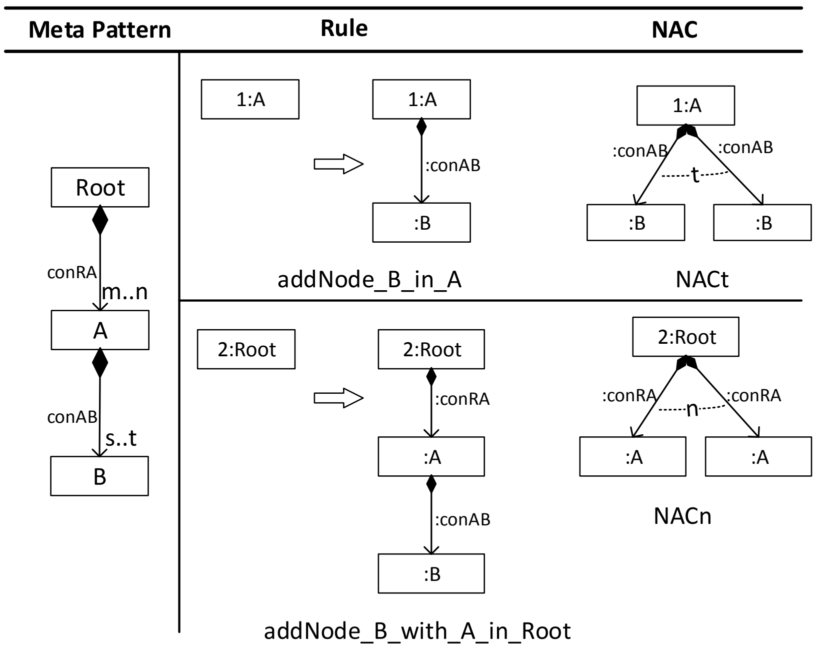


Figure 4.5: Rule scheme for *transitive-node-creation rules* (of length 2)

the rule *insert_additionalEdge_targetport* as depicted in Figure 4.9.

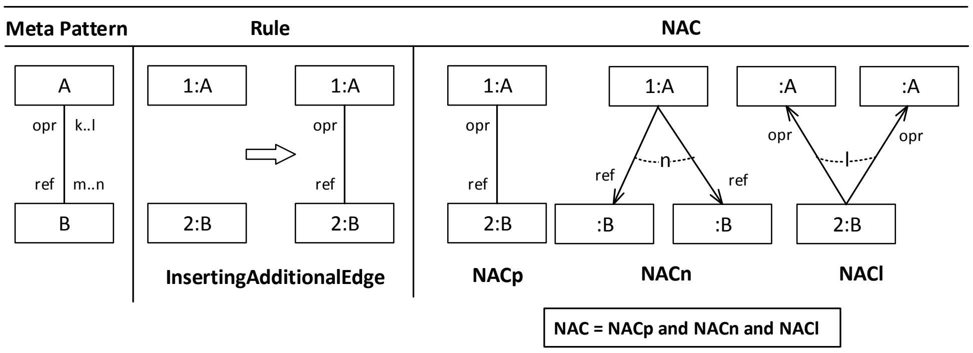


Figure 4.6: Rule scheme for *additional-edge-insertion rules*

In some places of our algorithm, it will be necessary to check if source- or target nodes for an edge of a certain type have existed and able to serve as proper source- or target nodes without violating the upper bounds of the certain edge type (iii). This is done using *additional-edge-checking rules* which are derived for each edge of non-containment type. The general scheme is depicted in Figure 4.7. These rules are applicable if and only if there is a source node where the upper bound of the non-containment is not yet reached. Thus, if there is no match for rules of this kind, no source node for an edge of this type is available. The same kind of rule is derived for the target node type as well. A concrete example for the edge type targetport is the rule *check_proper_sourceNode...* in Figure 4.9.

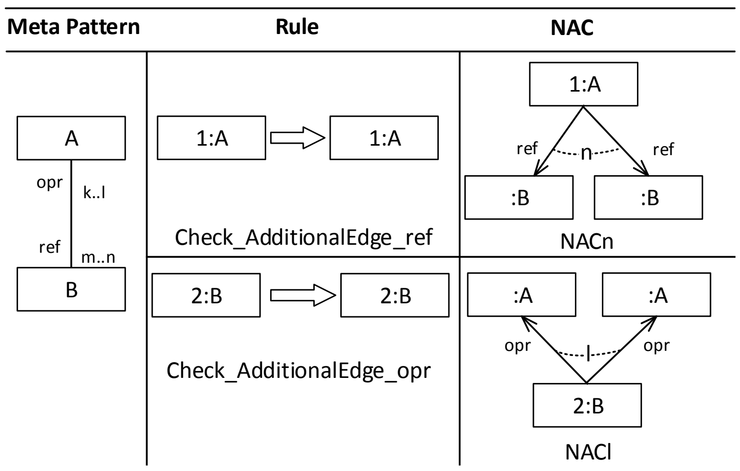


Figure 4.7: Rule scheme for *additional-edge-checking rules*

4.3.3 Generation Strategies: Parameterization

Since we use a rule-based approach, the model generator can be parameterized w.r.t. given specifications. In the following, we develop four strategies for generating models w.r.t. user specifications. The resulting EMF models conform to upper bounds, EMF and meet the user specification but may violate lower bounds. They are then used as input for the model repair algorithm to receive a consistent EMF model. The user can:

1. specify the number of elements that is minimally to be created,
2. specify a node type and the number of nodes of this type that is minimally to be created,
3. specify an edge type and the number of edges of this type that is minimally to be created, and
4. combine the above-mentioned strategies sequentially in arbitrary order.

In each case, the generation is initialized by creating a node of the root type.

Adding Elements of Arbitrary Types

In this strategy, the user can specify how many elements (nodes and edges) are minimally to be created. The idea behind this strategy is to randomly execute a set of rules for adding nodes and inserting edges of arbitrary types without violating the corresponding upper bounds and the EMF constraints. Hence, in this strategy all rules of kinds *additional-node-creation* and *additional-edge-insertion* are collected into one independent unit. This independent unit is then applied as often as the user specification requires.

While the independent unit in Henshin is implemented using a uniform distribution, in general this strategy could be performed using other distributions, e.g., by leveraging a stochastic controller [148].

Adding Nodes of a Specific Type

In this strategy, the user can specify a node type and the number of nodes of the specified type that should minimally be created. This strategy is implemented as an independent unit containing all *transitive-node-creation* rules for the specified node type. This unit is applied as often as the user has specified. An example unit for the node type Port is given in Figure 4.8.

A second way facilitates priority units: One always tries to create a node of the fixed type inside an already existing container via the *additional-node-creation* rules (assembled into an independent unit) and if that is not possible, *transitive-node-creation* rules of increasing length are tried to be applied. Thus, this strategy puts the *transitive-node-creation rules*, sorted into independent units by length, into a priority unit, sorted by increasing length. Compared to the first variant of this strategy, here the number of nodes of other types that are created as well is minimized but also the diversity of the output when applying the same strategy several times is decreased.

Adding Edges of a Specific Type

In this strategy, the user can specify an edge type (non-containment) and the number of edges of the specified type that should minimally be created in the output model. This strategy is similar to the strategy above. Thus its basis is a unit that just contains the *additional-edge insertion rule* for the specified type. However, if this rule is not applicable, a source or a target node (or both) for an additional edge of that type are missing. The *additional-edge-checking rules* for this edge type are used to detect this. Then, according *transitive-node-creation rules* for the type of the missing node can be used to create the missing source and/or target node(s). Summarizing, this strategy consists of a priority unit where the first comprised unit is just the *additional-edge-insertion rule.* Its second comprised unit is a sequential unit with two conditional units where the conditional units check for source or target nodes to be missing, respectively, and create according nodes if that is needed.

Figure 4.9 presents this strategy at the example of the targetport-edge. The according priority unit *add_edge_targetport* consists of two levels: The first one contains the rule *insert_additionalEdge...*. The second level is specified as a sequential unit *add_proper_source_target_Node...*: The conditional units add a node of type

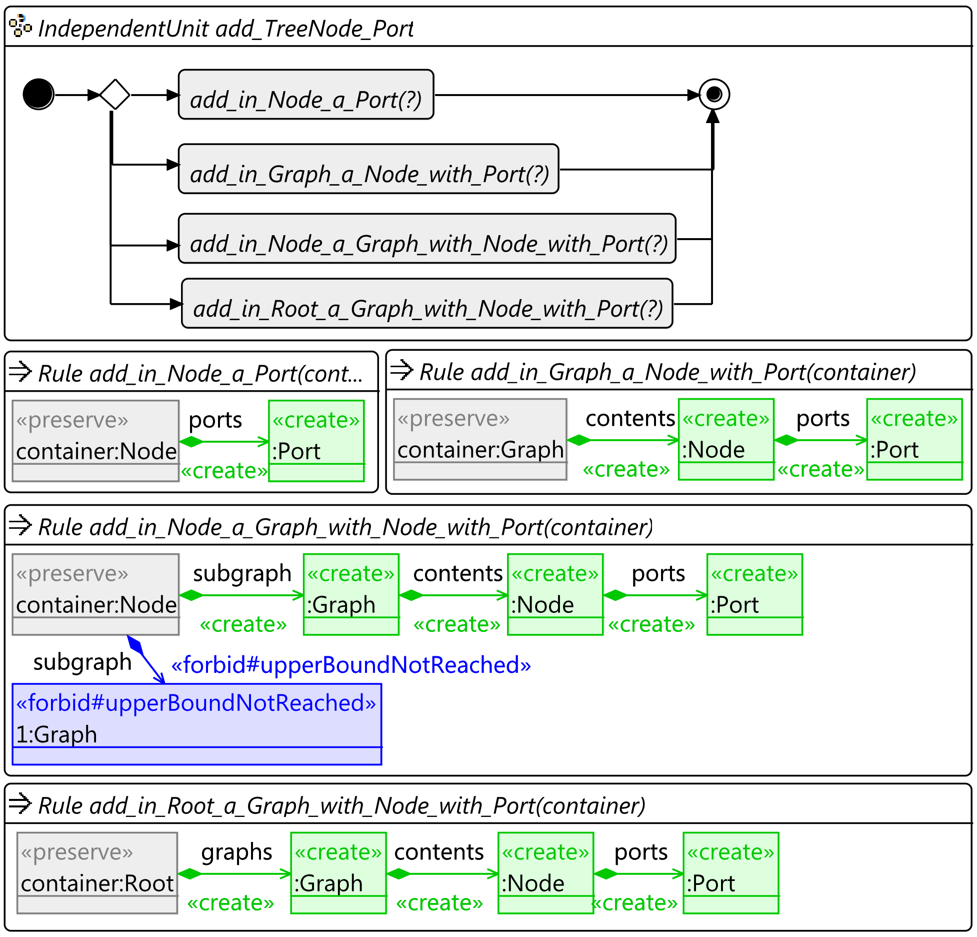

IndependentUnit add_TreeNode_Port
add_in_Node_a_Port(?)
add_in_Graph_a_Node_with_Port(?)
add_in_Node_a_Graph_with_Node_with_Port(?)
add_in_Root_a_Graph_with_Node_with_Port(?)
Rule add_in_Node_a_Port(cont...
«preserve» container:Node
ports «create»
«create» :Port
Rule add_in_Graph_a_Node_with_Port(container)
«preserve» container:Graph
contents «create»
«create» :Node
ports «create»
«create» :Port
Rule add_in_Node_a_Graph_with_Node_with_Port(container)
«preserve» container:Node
subgraph «create»
«create» :Graph
contents «create»
«create» :Node
ports «create»
«create» :Port
subgraph «forbid#upperBoundNotReached»
«forbid#upperBoundNotReached» 1:Graph
Rule add_in_Root_a_Graph_with_Node_with_Port(container)
«preserve» container:Root
graphs «create»
«create» :Graph
contents «create»
«create» :Node
ports «create»
«create» :Port

Figure 4.8: Independent unit for randomly creating a containment tree containing a fixed number of nodes of type Port

Edge (source) if there is no proper one and a node of type Port (target) if there is no proper one. The conditional unit *check_add_proper_sourceNode...* uses the rule *check_proper_sourceNode...* in the if-statement. The then-statement is set to true whereas the else-statement is configured with a priority unit *add_treeNode_Edge* which adds an Edge-node respecting upper bounds and the EMF constraints. The conditional unit adding a missing target node is built up analogously.

Sequential Combination of Strategies

As our approach allows for an arbitrary seed model as input, the result stemming from one application can be used as input for a second application. This allows for unlimited sequential combinations of strategies.

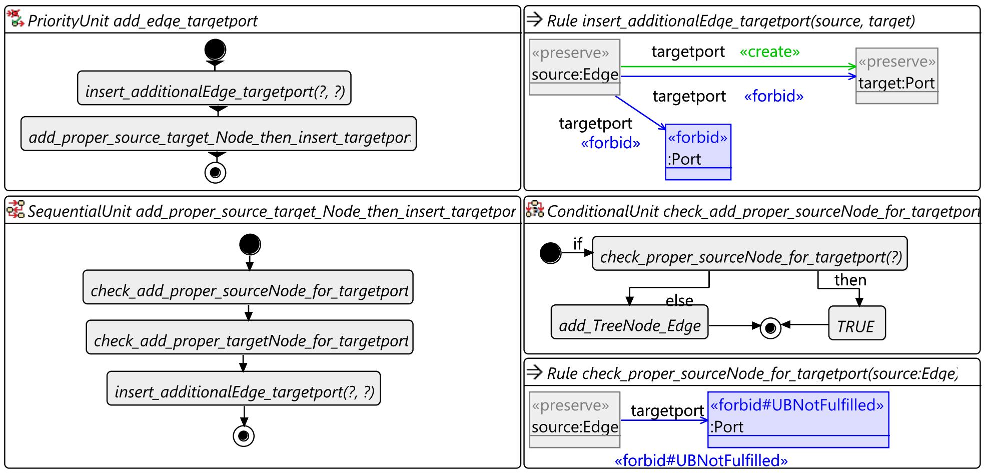

PriorityUnit add_edge_targetport
insert_additionalEdge_targetport(?, ?)
add_proper_source_target_Node_then_insert_targetpor
Rule insert_additionalEdge_targetport(source, target)
«preserve» source:Edge
targetport «create»
«preserve» target:Port
targetport «forbid»
targetport «forbid»
«forbid» :Port
SequentialUnit add_proper_source_target_Node_then_insert_targetpor
check_add_proper_sourceNode_for_targetport
check_add_proper_targetNode_for_targetport
insert_additionalEdge_targetport(?, ?)
ConditionalUnit check_add_proper_sourceNode_for_targetport
if
check_proper_sourceNode_for_targetport(?)
then
else
add_TreeNode_Edge
TRUE
Rule check_proper_sourceNode_for_targetport(source:Edge,
«preserve» source:Edge
targetport
«forbid#UBNotFulfilled» :Port
«forbid#UBNotFulfilled»

Figure 4.9: Units for inserting a fixed number of edges of type targetport

4.3.4 User Interaction

Since our approach is rule-based, it is also possible to allow for user interaction. Instead of random rule applications at random matches, the available rules and matches can be presented to the user for selecting at which match a rule has to be applied and how many times. That is promising for generating different tree structures of various weights. While it may not desirable to completely generate large models in such a way, a hybrid strategy can be applied to utilize the selection process, e.g., by employing heuristic data. EMF Repair already supports this kind of user interaction.

4.3.5 Limitations and Formal Guarantees

Limitations

A user may only specify the *minimum* number of desired elements; the specification of a maximum number is not yet supported within our approach. Although the generation process applies the respective rules exactly as often as specified during the model increase phase, some of the rules create more than one element and additional model elements may be created to repair violations of lower bounds during the consecutive model repair. Moreover, we cannot guarantee that the user specification is fully met since necessary rules may not be applicable as often as specified and backtracking is not used. Even if the specification could be met in principle, it may happen that the specific selection, order, and matches of rules do not succeed as they are randomly chosen in the current version of the approach. By counting created elements, it can always be decided whether a user specification has been met, and thus, the user can be informed. In our experiments (in Section 4.5), every generated output meets the selected specifications. Thus, while more research is needed to precisely evaluate the severity of our limitations, the performed experiments are positive evidence that these limitations are rather small even for reasonably complex meta-models.

Formal Guarantees

In case of termination, our approach guarantees a consistent EMF model as output: All generation rules conform to a design that is proven to preserve EMF constraints in [14]. Moreover, applications of these rules cannot introduce violations of upper bounds as they are equipped with corresponding NACs. So each strategy mentioned above is guaranteed to result in an instance model that conforms to EMF and does not violate any upper bounds. Moreover, it is ensured by the finite number of rule calls specified in each strategy that the increase phase terminates. Thus, suitable input for the model completion process of EMF Repair [93] is ensured after finitely many steps. For model completion, termination was proven in the case of f.f.i. meta-models while correctness was proven in all cases in [92]. If the user specification is met after a model has been increased, it is met after model completion as well since no deletion takes place during model completion. Even an increased model that does not meet the user specification is an EMF model and hence a suitable input for EMF Repair. Thus, it can be completed and returned to the user as a consistent EMF model. The given user specification, however, is only partly satisfied in this case.

4.4 Tooling

We have developed two Eclipse plug-ins that are available for download.[1] In the following, we give an overview of the architecture of the developed tool with the help of the UML component diagram.

Our implemented techniques consist of two main components depicted in Figure 4.10 and described as follow:

1. **Meta2GR:** The first plug-in is a meta-tool, called *Meta2GR*. It takes a domain meta-model as an input and generates an Eclipse plug-in containing a model transformation system (MTS) (the rules and the control units) implemented in Henshin. This is done by applying the meta-patterns depicted in Figures 4.4-4.7, which are specified as rules typed over the Ecore meta-metamodel, to the given domain meta-model. Using the data of the matches, the domain rules w.r.t. their schemes are created, thereafter. Once the MTS is generated, there is no need to use the meta-tool again as long as the meta-model is not changed. The tool composes two main components:

   - *Grammar Generator*: It is responsible for applying the meta-patterns, finding their matches, and generating the different kinds of generation rules

[1] https://github.com/RuleBasedApproach/EMFModelGenerator/wiki

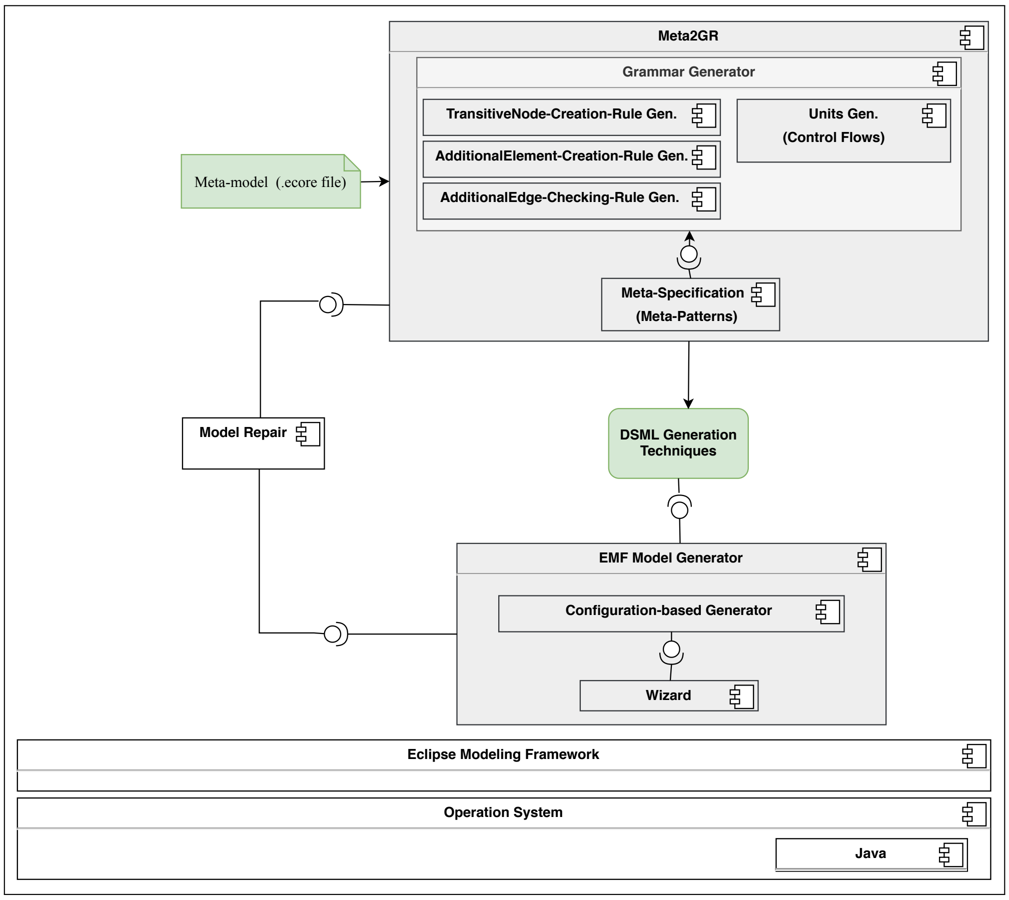

Meta2GR
Grammar Generator
TransitiveNode-Creation-Rule Gen.
AdditionalElement-Creation-Rule Gen.
AdditionalEdge-Checking-Rule Gen.
Units Gen.
(Control Flows)
Meta-model (.ecore file)
Meta-Specification
(Meta-Patterns)
Model Repair
DSML Generation Techniques
EMF Model Generator
Configuration-based Generator
Wizard
Eclipse Modeling Framework
Operation System
Java

Figure 4.10: An overview of the architecture of our techniques for model generation

w.r.t. their schemes and wrapping them in proper units, thereafter. This component has several generators as follow:

- *Transitive-node-creation-rule Generator*: It is for deducing and generating the rules which add a node of a certain type inside of one of its transitive containers without violating the corresponding upper bound and the EMF constraints.
- *Additional-element-creation-rule Generator*: It is for deducing and generating additional-node-creation and additional-edge-insertion rules.
- *Additional-edge-checking-rule Generator*: It is for deducing and generating the rules which check if the possible source and target nodes exist for an edge of a certain type.
- *Unit Generator*: It is for generating the corresponding control flows as transformation units being configured with the generated rules.

• *Meta-Specification*: It contains the specification of the meta-patterns for deducing the different kinds of the generation rules. They are designed as Henshin rules typed by the Ecore meta-metamodel.

2. **EMF Model Generator:** The second Eclipse plug-in is a modeler tool which uses the derived MTS to generate instance models. It provides the following functionalities:

   - Generating consistent EMF models. The inputs are the user specifications, e.g., the minimal number for adding elements of a selected type. The outputs are consistent EMF models.
   - Generating consistent EMF models incrementally. Starting from a given (consistent) EMF model, the user can generate other consistent EMF models. This might be helpful, e.g., to generate a huge consistent EMF model starting from a large consistent EMF model, and thus there is no need to generate the whole model from scratch.
   - Generating EMF models that may contain violations. This is done by applying rules designed not to consider the multiplicity bounds, i.e., excluding the corresponding application conditions and keeping the ones that respect the EMF constraints.

   This tool consists of the following main components:

   - *Configuration-based Generator*: It implements the strategies for generating models w.r.t. user parametrizations.
   - *Wizard*: It composes the specifications of the graphical user interfaces used to receive the inputs of the user.

4.5 Evaluation

In this section, we evaluate our approach to instance generation. In particular, we want to answer the following research questions (RQs):

RQ 1.1: How fast can instance models of varying sizes be generated in our approach?
RQ 1.2: How does this relate to the scalability of the state-of-the-art instance generation tool?

RQ 2: How diverse are instance models generated using Strategy (1)? Does the use of parametrization help to increase diversity compared to this?

All experiments were performed on a desktop PC, Intel Core i7, 16 GB RAM, Windows 7 x64 using Eclipse Oxygen. Additionally, our Eclipse-based tool was configured to use the default settings of Eclipse, e.g., the heap size was limited to 1 GB. All the evaluation artifacts are available for download.[3]

4.5.1 Scalability Experiments

To answer RQ 1.1, we conducted three scalability experiments. We used 8 meta-models taken from the literature and projects, namely the Statechart meta-model of

Magicdraw [66], web model [16], car rental and class model [5], Bugzilla, Latex, Warehouse, and GraphML (GML).[2] The average size of the meta-models is 44 elements (16 nodes, 17 edges, 11 attributes) and the number of multiplicity bounds is 24 on average. Table 3.8 presents information about the meta-models. The overhead for generating the needed transformation rules and units was, on average, less than 5 seconds, and we will thus focus on the run-time of the model generation in the sequel.

Experiment 1. In the first experiment, we randomly generated consistent EMF models of varying sizes up to 10 000 elements (counting nodes and edges) for each meta-model using Strategy (1) (in Section 4.3.3). For each size category, we generated 10 consistent EMF models and calculated the average run-time. Table 4.2 presents the results of this experiment. Considering all the meta-models and generated models of varying sizes, our tool always generates a consistent EMF model with at least 10 000 elements. Generation times were fastest for the Bugzilla meta-model and slowest for the GraphML one. To assess how robust the times are, we measured the time for generating a seed and for the subsequent repair separately. For each one, we also computed the corrected standard deviation (which is presented for model size 10 000 only). Generating the seed is generally faster than the subsequent repair, except for the StateChart and Warehouse meta-models. If the standard deviation is rather high, this tends to be the case for both, the seed generation and the repair (as for GraphML, Web Model, and Class Model). A closer inspection of the meta-models shows that higher run-times, as well as higher deviations of run-times, are caused by larger meta-model sizes (and hence larger sizes of derived MTSs) and higher numbers of interrelated multiplicity constraints.

Table 4.2: Average run-time (in seconds) for generating consistent EMF models of varying sizes for 8 meta-models (MM) using Strategy (1); for size 10 000, run-time is split into the generation of seed and subsequent repair where the corrected standard deviation is added in brackets, respectively.

MM\Model Size	**1 000**	**3 000**	**5 000**	**8 000**	**10 000**
Bugzilla	0.05	0.1	0.1	0.1	0.08 (0.006) + 0.04 (0.01)
Car Rental	0.27	5	17.9	72.3	65.5 (7.2) + 78.1 (4)
Class Model	0.16	1.7	9.4	61.5	13.2 (14.2) + 85 (113.8)
CoreWarehouse	0.81	4.5	18.9	67.9	0.4 (0.02) + 131 (10.9)
GraphML	0.4	2.6	16.7	79.2	39.3 (56) + 168.1 (119.6)
Latex	1.27	1.3	1.3	1.5	0.7 (0.01) + 0.8 (0.03)
StateChart	0.55	1.7	5.5	18.7	35.8 (3.9) + 1 (0.3)
Web Model	0.16	1.4	5.1	14.6	18.7 (18.8) + 6.2 (2.6)

Experiment 2. The second experiment is dedicated to generating huge models for a complex meta-model which would lead to complex model repair processes. The meta-model GraphML is right for this purpose as its number of lower bounds being non-zero is above the average. Fulfilling these bounds renders model repair into a

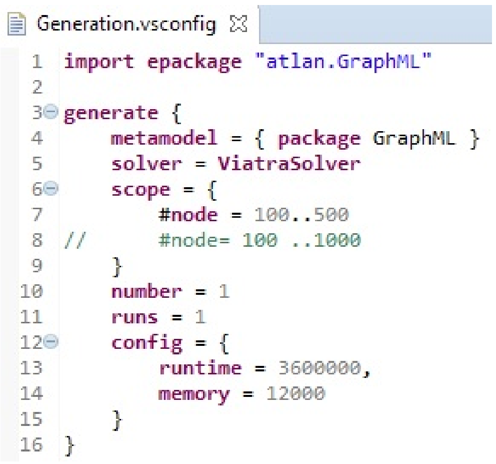
[2]The last four meta-models are extracted from http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Zoos

complex process. We expect the generation of models to become faster when using Strategy (3), i.e., when specifying a minimal number of edge occurrences of a certain type. In this case, nodes are introduced together with incident edges; this generation behavior should reduce the number of repairs needed to take place for fixing lower bound violations. Models of an average size of between 200 000 and 500 000 elements are generated in 6 to 32.5 minutes on average. Each generation process was repeated five times. The standard deviation was between 1.4 to 6.5 minutes, i.e., the run-times for the generation of these huge models are pretty stable. Table 4.3 presents the experiment results. Moreover, to give an impression of the tool performance for simple meta-models, we applied it to the Bugzilla meta-model. It is considered as simple since it consists of unrestricted containment edges only. The tool needed 1.2 minutes only to generate a consistent EMF model with a minimum of 500 000 elements.

Table 4.3: Average run-time and standard deviation (in minutes) for generating consistent EMF models of varying huge sizes for the GraphML meta-model using Strategy (3). The standard deviations are presented in brackets.

Model Size	**200 000**	**300 000**	**400 000**	**half a million**
Average Time (Min.)	6 (1.4)	11.4 (2.6)	23.3 (5.7)	32.5 (6.5)

Experiment 3. To answer RQ 1.2, we conducted a similar experiment with the VIATRA Solver (the state-of-the-art of the generators for EMF models). In [125], Semeráth et al. report that after an extensive warm-up phase the tool was able to generate models of size of about 500 objects in a few seconds, and models with about 4 000 objects in 20 minutes on average (median). We installed their Eclipse-based tool



```
import epackage "atlan.GraphML"

generate {
    metamodel = { package GraphML }
    solver = ViatraSolver
    scope = {
        #node = 100..500
//      #node= 100 ..1000
    }
    number = 1
    runs = 1
    config = {
        runtime = 3600000,
        memory = 12000
    }
}
```

Figure 4.11: Configuration of VIATRA Solver

as described at their web page [54] on January 23, 2019 and performed an experiment on the hardware mentioned above using Xtext SDK 2.1 [147] and VIATRA SDK 2.1 [144].

Figure 4.11 shows the configuration of their generator: the VIATRA solver is selected (on line 5), the size of the model is assigned to an interval of values 100 to 500 nodes (on line 7), the run time is limited to one hour and the memory is set to 12 000 MB (on line 14). As initial tests, we tried to generate models of size 10 to 100 nodes. After many runs, the tool was able to generate a small model of size about 20 objects for some of the meta-models in a few seconds or minutes. After that, we changed the scope to be 100 to 500 nodes and executed the tool to generate instance models for the GraphML meta-model. In several tests, after one hour of the execution, their tool could not generate any instance (although it explored roughly 2 500 states of the state space). This result is evidence that our tool generates consistent models considerably larger and faster than what the VIATRA can perform.

4.5.2 Diversity Experiments

To assess the diversity of the output of our tool we conducted two experiments.

Diversity by Strategy (1)

We randomly generated sets of 10 instance models of the meta-model of GraphML for sizes of about 100, 1 000, 2 000, and 10 000 elements using Strategy 1). For each set, we computed the difference for each pair of instance models using a simple metric based on their respective local structure: For each instance model M, we computed the in- and outdegrees in_n and out_n of every node n resulting in sets of tuples

$$D_M = \{(in_n, out_n) \mid n \text{ is node of } M\} \ .$$

Given two such sets of tuples stemming from different instance models M and N, we computed $|D_M \bigtriangleup D_N|$, i.e., the cardinality of the symmetric difference of the respective sets. We call this metric the *Number of Distinct In-/Out-Degrees* (NoDIOD). It can be seen as a simplification of the metric used in [127], by only taking into account one very abstract kind of shape and ignoring typing information and larger possible structures.

Table 4.4: Diversity of randomly generated instance models for different sizes

NoDIOD\Size	100	1 000	2 000	10 000
Average	15.3	35.5	44.7	59.6
Standard Deviation	3.4	6.7	5.6	6.5

The results are presented in Table 4.4. While for two instance models of size 100 there exist 15 nodes on average connected in a way that is specific to one of the graphs, this number only grows to roughly 60 for instance models with 10 000 elements. While this shows that resulting models are different enough from each other, they contain a lot of nodes having an identical structure w.r.t. their connections. In particular, the proportion of unique patterns w.r.t. the size of the instance models strongly decreases with growing size. A more sophisticated metric would be needed to state the real extent of their diversity as typing information and larger structures are ignored by our metric.

Diversity by Strategy (2)

To test if the further parametrization of our algorithm is fit to introduce diversity, we conducted the following experiment. We randomly created a first set of ten instance models of size of about 2 000 elements of the meta-model of GraphML using Strategy 1). As a comparison, we created sets of instances using Strategy 2): For each node type, we created a set of ten instance models generated by specifying that this node type should minimally occur 500 times. Since we expected this to already affect the distribution of occurring elements, for each of the resulting sets we just calculated the Shannon index [131], an established diversity measure. This is, we considered the nodes of each of the sets as individuals and the node types as species and computed the Shannon index according to the formula

$$-\sum_{i=1}^{9} \frac{n_i}{N} \cdot \lg \frac{n_i}{N}$$

where N is the total number of nodes, i ranges over the non-abstract node types and n_i is the according number of nodes of that type. The results are presented

Table 4.5: Diversity of randomly generated instances parametrized by node type

	Strategy (1)	**Element**	**Key**	**Graph**	**Edge**
Shannon index	3	2.12	0.82	0.76	0.94
	HyperEdge	**Node**	**Port**	**EndPoint**	**Data**
Shannon index	0.92	0.99	1.57	1.48	2.06

in Table 4.5. The result shows that the distribution of elements among the types is significantly different between several of the sets which is a new kind of diversity compared to that considered above. Note that, the types of occurring elements in large instances resulting from Strategy (1) show nearly uniform distribution as the maximal possible Shannon index is $\lg 9 \approx 3.17$.

To assess that even the sets with similar Shannon indexes differ from one another, we checked for the types actually occurring in each set and compared them. The results are depicted in Figure 4.12. For example, 66 % of the nodes are of type HyperEdge if HyperEdge (H.E.) is chosen as type parameter, and 68 % of the nodes are of type Edge if Edge (E.) is chosen as parameter, even though both sets of models exhibit almost the same Shannon index.

To answer RQ 2, instance models resulting from our first generation strategy contain patterns distinguishing them from each other, even when only considering a small pattern without typing information. Hence, the generated output is diverse even if

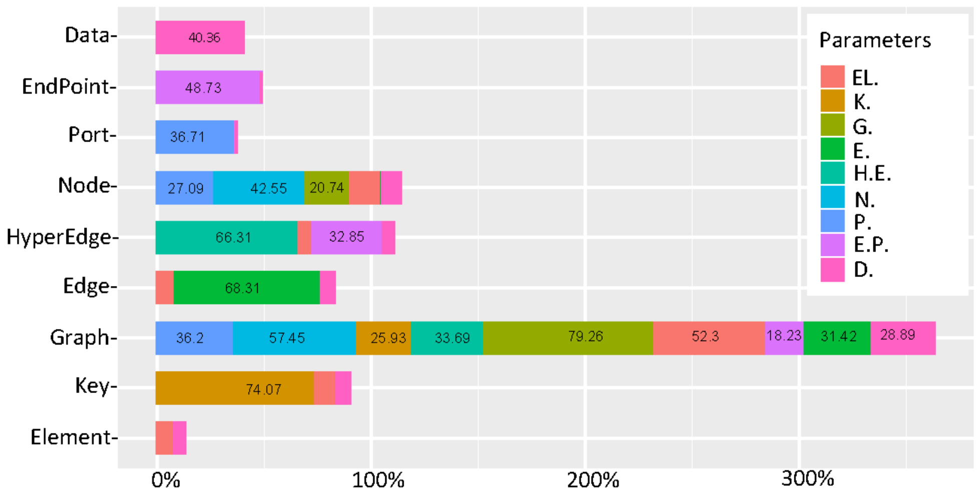


Figure 4.12: Relative number of occurrences (x-axis) of node types (y-axis) in all the instance models generated using Strategy (2); results obtained for different parameter settings are encoded in colors and each color indicates one instance model. For example, 79.26% nodes of type Graph and 20.74% nodes of type Node are created in an instance model for parameter Graph (G.).

the effect diminishes with the growing size of instances. Moreover, choosing different node kinds as a parameter leads to significantly different distributions of the types of occurring elements. Hence, parametrization is fit to introduce a new kind of diversity compared to the one observed for large instance models resulting from Strategy (1), which exhibit a nearly uniform distribution of occurring types. However, the set generated by Strategy (1) shows diversity compared to the other sets generated by Strategy (2).

4.5.3 Threats to Validity

In our evaluation, we selected 8 meta-models. Evaluation results might differ when choosing others. We are confident, however, that our results are representative as we selected meta-models from diverse backgrounds, with reasonable sizes, and with varying numbers and forms of multiplicities. The used metric to measure diversity completely abstracts from details of the underlying graph structures of generated instance models. On the one hand, abstracting from such details typically underrates diversity rather than overrating it. On the other hand, we have to acknowledge that the form of diversity we show in our experiments is limited to the distribution of types.

4.6 Related Work

In this section, we first present a classification scheme for existing model generation approaches, which is shown in Figure 4.13. *Language-* and *application-specific* approaches will be only considered very briefly due to their principle limitations. Next, we consider generic *solver-based* and a *tableaux-based* approach, which share with our approach that they can be applied to arbitrary meta-models and used for a variety of different application scenarios. Generic *rule-based* approaches are the most closely related to ours and will be considered in detail. We conclude our review by summarizing the limitations of existing generic approaches which motivate our work.

Language- and Application-Specific Approaches

Language-specific approaches are presented, e.g., in [75, 138, 81, 82]. Laurent et al. [75] present a manipulation-based approach employing multi-objective genetic algorithms for the generation of process models. Solver-based approaches have been proposed by Svendsen et al. [138], McQuillan and Power [82], and McGill et al. [81] targeting domain-specifc languages referred to as metrics, object-role and train modeling language, respectively.

Application-specific approaches mostly concentrate on the scenario of model transformation testing with the aim of generating sets of models serving as model transformation test suites exposing certain coverage criteria. This has been first addressed by Fleurey et al. [43] and later been extended and implemented in [18, 74, 146]. Xiao et al. [57] propose a rule-based approach which randomly generates large test models for stress testing of model transformations in industrial applications.

Generic Solver-based Approaches

Solver-based approaches indirectly generate models by (i) translating a meta-model into a logical formula, (ii) using an off-the-shelf solver to find possible solutions, and (iii) translating back the found solutions into instances of the meta-model. In most cases, solver-based approaches are capable of generating models which respect further well-formedness constraints such as OCL constraints [51] since these can be translated into the logical formula, too. The approaches presented in [128, 138, 81, 82] use Alloy [63] for this purpose. Although we see no general limitation for them to be applied to arbitrary meta-models, the translations to Alloy presented in [138, 81, 82] target dedicated domain-specific languages (see Section 4.6), and the language-independent translation presented by Sen et al. [128] is, in contrast to our rule generation, not fully automated. Most importantly, however, the scalability of using an off-the-shelf solver is limited to very small models [125].

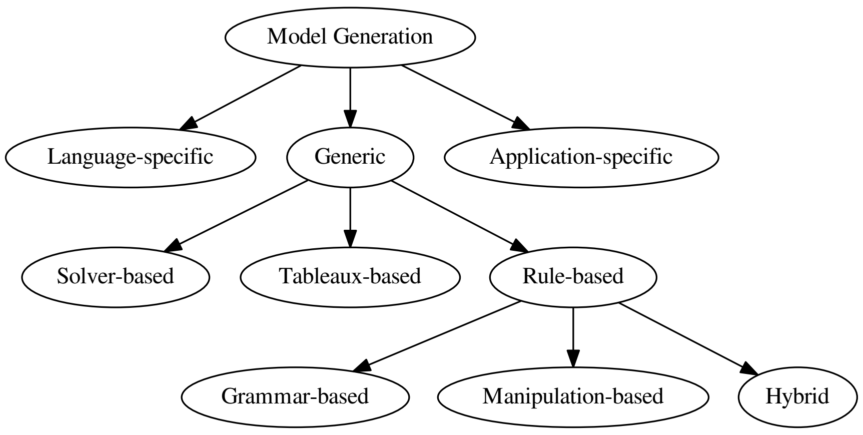


Figure 4.13: Classification of existing approaches to model generation.

A Generic Tableaux-based Approach

Schneider et al. [120] present an approach and tool for generation of symbolic instances of attributed typed graphs fulfilling a given set of first order constraints. The approach is based on a correct and refutationally complete tableaux calculus for graph constraints. The resulting minimal graphs encode (infinitely) many instances fulfilling the set of constraints. While this is highly desirable to achieve an overview of the structure of possible instances, retrieving large graphs from the symbolic instances is not directly supported. Moreover, the work does not aim at EMF, and it is also not possible to add the EMF constraints as some of those are not first order. Additionally, the work cannot generate a valid graph starting from an invalid input graph.

Generic Rule-based Approaches

Grammar-based approaches. Ehrig et al. [41] present an approach for converting type graphs with restricted multiplicity constraints into instance-generating graph grammars. Starting from the empty model, instance generation proceeds in three phases; (i) the addition of nodes, (ii) the addition of edges (with nodes if necessary) to respect lower bounds, and (iii) the addition of additional edges without violating upper bounds. Taentzer [139] generalizes the approach from restricted to arbitrary multiplicity constraints. However, the approach is presented for plain graphs, i.e., there is no consideration of containment edge types and other EMF constraints. Moreover, there is no discussion on how to parametrize the approach and how to support user interactions. Finally, there is no implementation and thus, no experiment showing the scalability of the approach.

Radke et al. [114] present a translation from OCL constraints to graph constraints which can be integrated as application conditions into a given set of transformation rules [89]. The resulting rules guarantee validity w.r.t. the OCL constraints. The work is motivated by application to instance generation. However, no dedicated algorithm is presented. Instead, each rule-based approach could use this technique to adapt its rules to respect further well-formedness constraints additionally. As a drawback, the resulting application conditions can render rules inapplicable.

Another grammar-based approach is presented by Mougenot et al. [86]. By reducing models to their containment structure, a tree grammar is derived from that meta-model projection. The approach uses the so-called Boltzmann method to create tree structures uniformly, where uniformity means that for a tree of a given size (in number of nodes), the method is capable of uniformly generating all tree structures of that size. Similarly, the tool *EMF random instantiator* [50] presented by Gómez and the AtlanMod Team (no paper presenting the tool is found) considers only the containment edges. While both approaches are highly efficient, reducing models to their containment structure is a severe oversimplification in practice.

Additionally, both frameworks *RandomEMF* presented by Scheidgen [118] and *EMG* presented by S. Popoola et al. [109], aid users to manually specify a generator that automatically generates models, and thus, the developing time and effort can be reduced. However, ensuring that the generated models conform to the meta-model and the generated models satisfy the required constraints is the responsibility of the developer of the model generator. Moreover, there is no discussion about how to design the generator. Besides, there are no mature experiments evaluating the scalability of the tools.

Manipulation-based approaches. A manipulation-based approach known as the SiDiff model generator (SMG) has been proposed by Pietsch et al. [108, 107]. It takes an existing model as input and manipulates it by applying model editing operations, configured by a stochastic controller. Pietsch et al. propose to adopt the approach presented by Kehrer et al. [66] which generates a complete set of consistency-preserving edit operations from a given meta-model. However, it supports only a restricted class of meta-models, and the edit operations can be applied to valid models only. Moreover, the stochastic controller has been designed to generate sequences of models which mimic realistic model histories [148]. Thus, the generated models are, on purpose, very similar to each other lacking diversity.

Hybrid approaches. A hybrid approach is implemented within the VIATRA Solver [125]: Rules are used to generate an instance model from a seed model or from scratch, and a solver is used to guarantee validity with respect to additional well-formedness constraints. During the generation process, a partial model is extended using rules, and this partial model is continuously evaluated w.r.t. the validity of the constraints using a 3-valued logic [126]. By under-approximation, the search space is pruned as soon as a partial model cannot be refined into a valid model. The evaluation of the constraints can be done using a specifically developed solver or an off-the-shelf one. The resulting instance models fulfill the additional constraints and conform to

EMF. The VIATRA Solver supports well-formedness constraints in a particular graph pattern language; support of OCL is not yet implemented. Moreover, the VIATRA Solver has been investigated successfully for diverse output. However, while experimental results indicate that the approach is 1–2 orders of magnitude better than existing approaches using Alloy [125], the authors also mention that the scalability of their approach is not yet sufficient.

We summarize the related work through selected generic approaches from all categories in Table 4.6 w.r.t. important characteristics. First, we indicate whether the approach is implemented in a tool (column 1). Second, we are interested in manipulating an existing seed model (column 2), e.g., for the sake of generating model evolution scenarios. Here, ∘ indicates that only special kinds of seeds are possible. Third, concerning the consistency level of generated output models, we are interested in the conformance with EMF (column 3) and additional well-formedness constraints, including multiplicities (column 4). Here, + indicates partly and ++ full support of multiplicity constraints, whereas +++ means support of more general well-formedness constraints. Fourth, we are interested in the properties of the generation algorithm itself, which should be configurable (column 5), offer interaction possibilities (column 6), and be scalable (column 7) in order to support the generation of diverse and large instances, respectively.

Table 4.6: Summary of selected generic approaches to model generation w.r.t. important characteristics we aim at in this work.

			Input	Output		Algorithm		
Category	Approach	implementation	existing seed	EMF	well-formedness constraints	configuration	interaction	scalable
Solver	Sen et al. [128]	+	−	∘	+ + +	−	−	−
Tableaux	Schneider et al. [120, 121]	+	∘	−	+ + +	−	−	?
Rule-based	Taentzer [139]	−	−	−	++	∘	+	?
	Mougenot et al. [86]	∘	−	∘	+	∘	−	+
	Pietsch et al. [107]	+	∘	+	+	+	+	∘
Hybrid	Semeráth et al. [125]	+	∘	+	+ + +	+	−	∘
Rule-based	Our approach	+	+	+	++	+	+	+

None of the generic approaches to model generation fully meets all criteria. Given a meta-model with multiplicities as the only well-formedness constraints, we are heading towards a model generator that supports all quality attributes.

4.7 Conclusion

We developed a rule-based approach for generating consistent models w.r.t. arbitrary multiplicities and EMF constraints. Since we use a rule-based approach, our generator is configurable to support user specifications and to allow user interaction. Several parameterization strategies are presented to generate different sets of consistent EMF models. Two Eclipse plug-ins have been developed: *Meta2GR* automatically translates the meta-model of a given DSML to an MTS and the *EMF Model Generator* uses the derived MTS to generate consistent EMF models. We evaluated the scalability of our approach by generating large instances of several meta-models of different domains and showed that models with 10 000 elements can be generated in about a minute on average. Furthermore, our tool can generate consistent EMF models of 500 000 elements in less than 2 minutes for a meta-model with largely unrelated multiplicity constraints and in about 30 minutes for a meta-model with closely interrelated ones. A scalability comparison with the state-of-the-art instance generator is provided showing that our generator is efficient for generating large and consistent EMF models from meta-models without OCL constraints. Moreover, we showed that a certain form of diversity between the generated models can be achieved by configuration. The related work is discussed in detail as well.

As future work, we intend to support a set of OCL constraints (at least negative constraints). Translating OCL constraints to graph patterns [114, 89], and further to application conditions of rules is promising to develop an automated interactive approach for generating consistent EMF models conforming meta-models with (OCL) constraints. Furthermore, we want to support further configurations and generate realistic models by leveraging a stochastic controller [148] and extend our experiments to generate huge models starting from existing large models as seed models.

Part II

Consistency Ensuring Techniques for Model Transformations

In Model-Driven Engineering (MDE), model transformations are the key operations for manipulating models, including, e.g., a repair transformation as described in Chapter 3, refactoring, and code generation. Applying a transformation to a model may change its consistency with respect to a set of constraints, and thus, errors may occur. In several application scenarios, models have to fulfill a set of constraints or their consistencies (at least basic ones) have to be preserved during the transformation process. For example, applying editing operations to a model has to satisfy the constraints required by the model editor to view it. Moreover, each state of a concurrent and distributed system should fulfill some required invariants such as safety properties, and each refactoring should preserve the model consistency.

Manually enhancing a set of rules to guarantee or to preserve a set of constraints is a tedious, time-consuming, and error-prone task. Furthermore, it requires high skills related to the theoretical foundation and the environment. Optimizing the results is another challenge as well.

In this part, we present two works: In Chapter 5, we introduce a correct-by-construction technique for translating OCL constraints into semantically equivalent graph constraints and integrating them as guaranteeing application conditions into a transformation rule. Our techniques realize an existing theory and automate the whole process. In Chapter 6, we present an optimizing-by-construction technique for application conditions for transformation rules that need to be constraint-preserving. This automatic construction of optimized preserving application condition is conceptually new. Moreover, we show the soundness of the technique.

We develop all the techniques as ready-to-use tools based on the Eclipse Modeling Framework. Further, we evaluate the efficiency (complexity and performance) of both works, assess the overall approach in general, and discuss the related work.

5 Automated Construction of Guaranteeing Application Conditions from OCL Constraints

This chapter shares material with the ICGT'18 paper "OCL2AC: Automatic Translation of OCL Constraints to Graph Constraints and Application Conditions for Transformation Rules" Nassar et al.[89].

Model transformations are often supposed to have a well-defined behavior in the sense that their resulting models are consistent w.r.t. a set of constraints. Based on existing theory, we develop an automated technique that is able to adapt a given rule-based model transformation such that resulting models *guarantee* a given constraint set. This technique is designed for models of the Eclipse Modeling Framework and based on graph constraints and graph transformations. Two main *correct-by-construction* functionalities are developed: First, OCL constraints are translated into semantically equivalent graph constraints. Secondly, graph constraints can further be integrated as application conditions into transformation rules. The resulting rule is applicable to an EMF model only if its application does not violate the original constraints. This technique is thus able to guarantee the consistency of a model transformation w.r.t a given set of constraints; its correctness is shown in the literature. Our technique is implemented as Eclipse plug-in, called *OCL2AC*, and enhances Henshin transformation rules in a fully automated way. In the evaluation, we show that our technique is feasible and effective in practice, i.e., both components work reasonably fast, and the complexity of the resulting output is far better than could be expected from theory.

5.1 Introduction

Model transformations are key artifacts of Model-Driven Engineering (MDE). They are used for various MDE-activities including translation, optimization, and synchronization of models [130]. Resulting models should belong to the transformation's target language, which means that they have to satisfy all the corresponding language constraints. This means that the developer has the task to design transformation rules such that they behave well w.r.t. language constraints. This task can easily become time-consuming and error-prone. Moreover, it requires high skills related to the theoretical foundation and the environment. Furthermore, a straight forward way to test the well-behavedness of a model transformation is to systematically apply it to selected test models and to analyze the transformation results. Since testing can become tedious, we are interested in a static and automatic way to guarantee constraints.

Based on existing theory [55, 114], we present a static, automated technique which is able to automatically adapt a given rule-based model transformation such that resulting models satisfy a given set of constraints. Use-cases for this technique are abundant, including instance generation [114], ensuring that refactored models do not show certain model smells (anymore), and generating model editing rules from meta-models to enable high-level model version management [66]. More examples of how advancing constraints into application conditions of transformation rules may be used can be found in [28].

Our technique builds upon the following basis (see Chapter 2 for more details): The de facto standard for defining modeling languages in practice are the Eclipse Modeling Framework (EMF) [34] for specifying meta-models and the Object Constraint Language (OCL) [99] for expressing additional constraints. Recent empirical findings suggest that OCL is especially fit to express complex constraints (compared to Java) [149]. Graph transformation [39] has been shown to be a versatile foundation for rule-based model transformation [40] focusing on the models' underlying graph structure. To reason about graph properties, Habel and Pennemann [55] have developed (nested) graph constraints being equivalent to first-order formulas on graphs. A construction of application conditions for transformation rules out of constraints was first developed for graphs by Heckel and Wagner [58] and then generalized in [55].

The first component of our technique is able to translate a reasonable subset of OCL constraints to nested graph constraints using the formally defined OCL translation in [114] as a conceptual basis. The second component of our technique integrates a graph constraint as an application condition into a transformation rule. The resulting application condition guarantees that the EMF model resulting from the successful application of the enhanced rule satisfies the original constraint. We call it a *constraint-guaranteeing* application condition. This integration does not change the rule' actions. Note, our technique does not yet check in advance whether the given transformation rule with its application condition does already guarantee the constraint or not.

Our technique is implemented as Eclipse plug-in, called *OCL2AC*, being based on EMF and the model transformation language Henshin.

Note that our OCL translator is novel: Instead of checking satisfiability, it enhances transformation rules such that their applications can result in valid models only. To that extent, our technique is static; moreover, it not just checks constraint satisfaction but also aims the user to improve transformation rules. Specifically, we make the following contributions:

- Solution design for automatically translating OCL constraints to graph constraints and application conditions of transformation rules (in Sections 5.3.1 and 5.3.2). The theoretical background for our technique is fully elaborated and proven to be correct. The most comprehensive references are [55] for the calculation of preconditions and [114] for the translation of OCL into graph constraints.
- Implementation: Providing a ready-to-use tool, called *OCL2AC*, based on EMF and Henshin (in Section 5.3.4). The domain developer can use this tool to automate the whole process.
- Evaluation of this implementation in which we study the complexity and the performance of constraint translation and integration (in Section 5.4).

The rest of this chapter is structured as follows: Section 5.2 contains our running example, a simplified statecharts meta-model. Section 5.3 describes the design and implementation of our technique in more detail and presents the developed tool and its functionalities while Section 5.4 report the results of several research questions investigated regarding the restrictiveness, complexity, and performance. We discuss related work in Section 5.5 and conclude in Section 5.6.

5.2 Running Example

To illustrate the behavior of our developed technique, we use a simple Statecharts meta-model displayed in Figure 5.1.

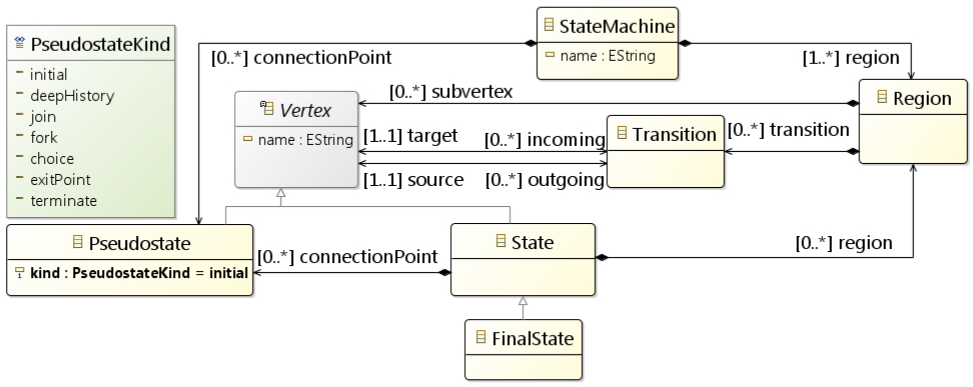


Figure 5.1: A simple Statecharts meta-model

A StateMachine contains at least one Region and potentially Pseudostates of a kind such as initial. A Region contains Transitions and Vertices. Vertex is an abstract class with concrete subclasses State and Pseudostate. A State can again contain Regions and Pseudostates to support the specification of state hierarchies. FinalState inherits from State. Transitions connect between Vertices.

A basic constraint on statecharts, which is not expressible by just the graphical structure of the meta-model or by multiplicities is the constraint no_outgoing_transitions: A FinalState has no outgoing transition. This constraint can be specified in OCL as follows:

```
context FinalState invariant no_outgoing_transitions: self.outgoing->
  isEmpty();
```

Our technique can translate this constraint into the graph constraint in Figure 5.2. The graph constraint states that a FinalState does not have an outgoing transition. Node names signify inclusions of graphs along the nesting structure.

$$\forall\left(\boxed{\text{self:FinalState}},\ \nexists\left(\boxed{\text{self:FinalState}}\xrightarrow{\text{outgoing}}\boxed{\text{var9:Transition}}\right)\right)$$

Figure 5.2: Graph constraint no_outgoing_transitions

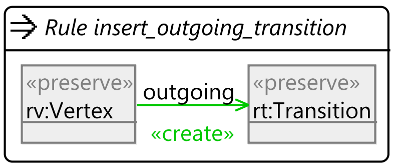


Figure 5.3: Transformation rule

Figure 5.3 shows a simple transformation rule in Henshin. The rule inserts an edge of type outgoing between a Vertex and a Transition which both have to exist before rule application and are preserved.

As a second capacity, our technique can adapt a given transformation rule such that it is not applicable if the result would violate a graph constraint. This behavior is achieved by integrating a new application condition into the rule. Integrating, e.g., the constraint no_outgoing_transitions into the rule insert_outgoing_transition results in the application condition displayed in Figure 5.4. The upper part forbids node rv:Vertex being matched to a FinalState. The lower part requires that the rule is matched to consistent models only, i.e., not containing already a FinalState with an outgoing Transition. It may be omitted if consistent input models can always be assumed.

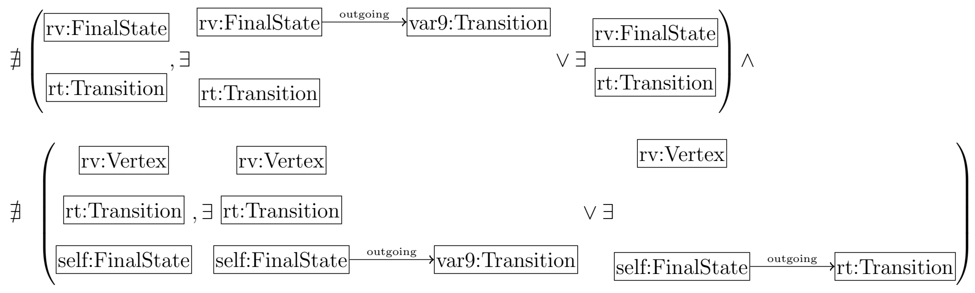


Figure 5.4: Application condition for the rule insert_outgoing_transition after integrating the constraint no_outgoing_transitions

5.3 Solution Design and Implementation

In this section, we present the solution design of the developed technique and present its main functionalities. Our correct-by-construction technique consists of two main components: (1) a translation from OCL constraints to graph constraints and (2) an integration of graph constraints as guaranteeing application conditions into transformation rules. Each of these two main components is designed to be usable on its own.

5.3.1 From OCL to Graph Constraints

The first component of our technique takes a domain meta-model and a set of OCL constraints as inputs and returns a set of semantically equivalent (nested) graph constraints as output. The translation process is composed of the steps shown in Figure 5.5, which can be automatically performed.

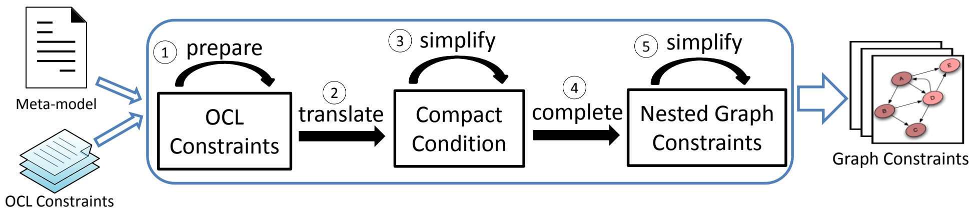


Figure 5.5: From OCL to graph constraints: Component design

In Step (1) an OCL constraint is prepared by refactorings. This is done to ease the translation process, especially to save translation rules for OCL constraints. The semantics of the constraint is preserved during this preparation. Step (2) translates an OCL constraint to a graph constraint along with the abstract syntax structure of OCL expressions. This translation largely follows the one in [114].

Let us consider the translation of OCL constraint no_outgoing_transitions to the graph constraint displayed in Figure 5.2: The expression self.outgoing→isEmpty() is refactored to not(self.outgoing→size() $\geqslant$ 1). Hence, a translation rule for isEmpty() is not needed. Then, this sub-expression is translated to a compact condition containing a graph with one edge of type outgoing from a node of type FinalState to a node of type Transition. The existence of such a pattern is negated.

Then a first simplification of the resulting compact condition takes place in Step (3), using equivalence rules [106, 114]. Applying those can greatly simplify the representation of a condition; they can even collapse nesting levels.

Step (4) completes the compact condition to a nested graph constraint being used to compute application conditions later on. The resulting nested graph constraint is simplified in Step (5) using equivalence rules again.

5.3.2 From Graph Constraints to Left Application Conditions

The second component of our technique takes a Henshin rule and a graph constraint as inputs and returns the Henshin rule with an updated application condition guaranteeing the given graph constraint. Figure 5.6 gives an overview of the steps to be performed:

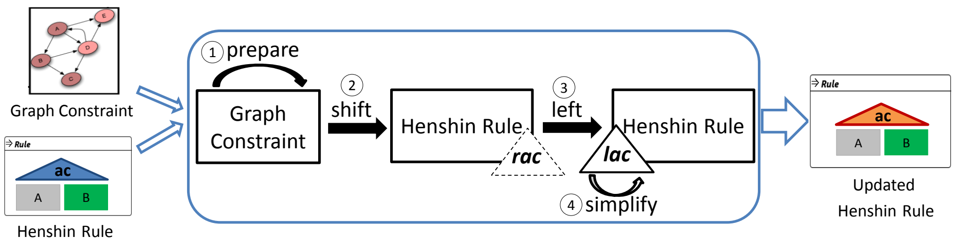


Figure 5.6: Integration as application conditions: Component design

In Step (1) the given graph constraint is prepared; if the input is a compact constraint, it is expanded to a nested constraint. Moreover, it is refactored to eliminate syntactic sugar. The operator ⇒ for implication, for example, is replaced with basic logic operators.

Step (2) shifts a given graph constraint *gc* to the RHS of the given rule so that we get a new right application condition *rac* for this rule. The main idea of `shift` is to consider all possible ways in which *gc* could be satisfied after rule application. This is done by overlapping the elements of the rule's RHS with each graph of *gc* in every possible way. This overlapping is done iteratively along with the nesting structure of *gc*. This algorithm is formally defined in [55] and shown to be correct. The result of this calculation is yet impractical as one would need first to apply the rule and then check the right application condition to be fulfilled. Therefore, we continue with the next step.

Step (3) translates a right application condition *rac* to the LHS of the given rule *r* to get a new left application condition *lac*. It is translated by applying the rule reversely to the right application condition *rac*, again along with its nesting structure. If the inverse rule of *r* is applicable, the resulting condition is the new left application condition. Otherwise *r* gets equipped with the application condition `false`, as it is not possible to apply *r* at any instance without violating *gc*. The rule *r* with its new application condition has the property that, if it is applicable, the resulting instance fulfills the integrated constraint.

Step (4) simplifies the resulting left application condition using equivalences as for graph conditions. The output is the original Henshin rule with an updated left application condition guaranteeing the given graph constraint.

5.3.3 Graph Condition Representation

Figure 5.7 shows the meta-model we developed to represent nested graph conditions. Basically, a NestedConstraint contains exactly one NestedCondition which is concretely a QuantifiedCondition, a True or a Formula. The domain of a nested constraint must compose an empty Graph, i.e, a graph without nodes and edges. a NestedConstraint may contain AttributeConditions also. A Formula has an operator property and contains at least one nested condition. A QuantifiedCondition has a quantifier property with value EXISTS or FORALL, and has a nested condition, a graph and a morphism. A Graph may contain Nodes and Edges being connected together. A Morphism may contain NodeMappings and EdgeMappings. A NodeMapping maps between a source node and a target node using the references origin and image, respectively. Similarly, the EdgeMapping does between two edges. Each NestedCondition has a domain which refers to a graph and it may have Variables.

Figure 5.8 shows the abstract syntax of the *completed* form of the nested graph constraint no_outgoing_condition shown in Figure 5.2. The abstract syntax of the *compact* form of the constraint no_outgoing_condition is similar to that one but without the morphisms and their contents, i.e., without the nodes m1, m2 and nm (the gray-colored elements).

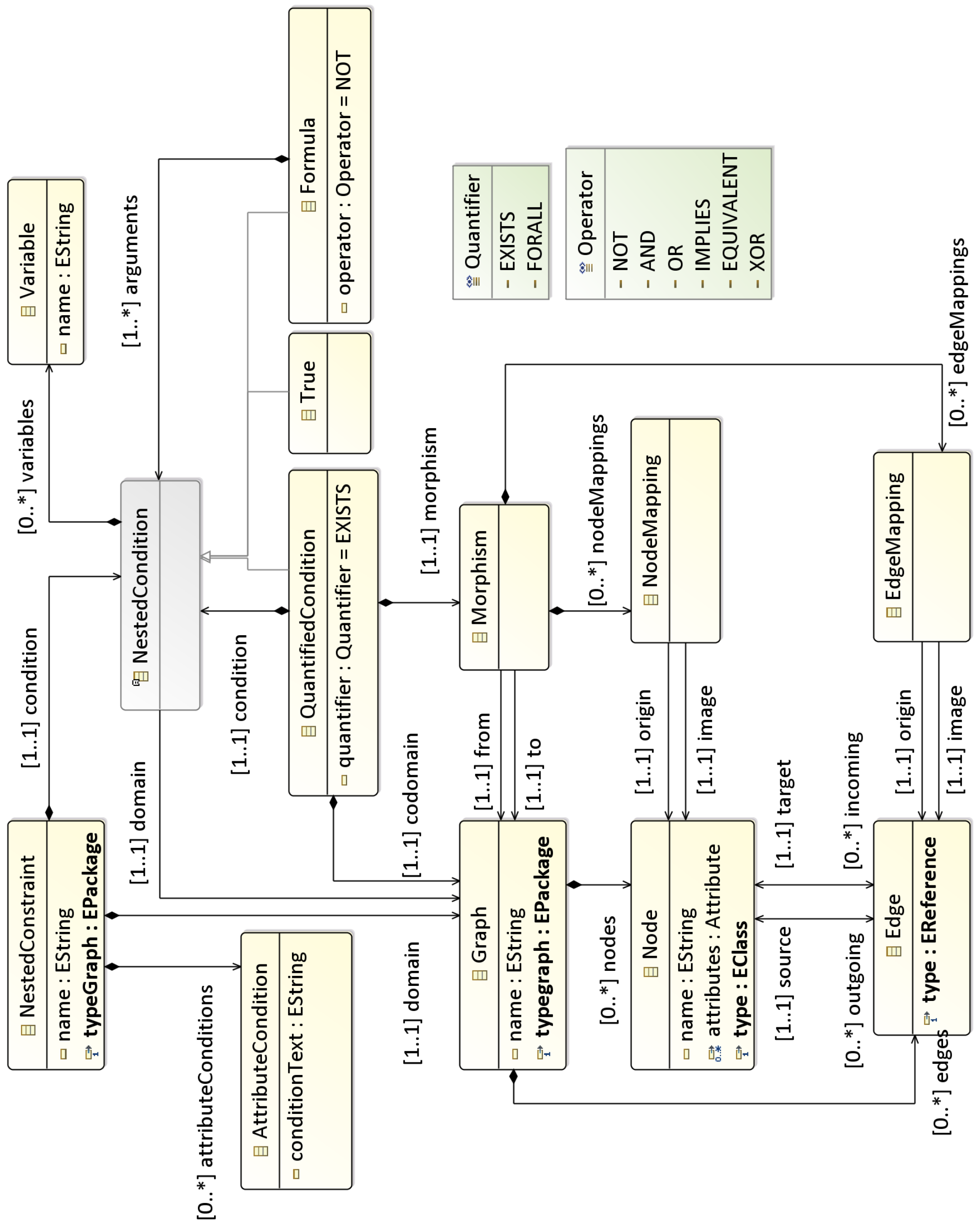


Figure 5.7: The meta-model of nested graph conditions

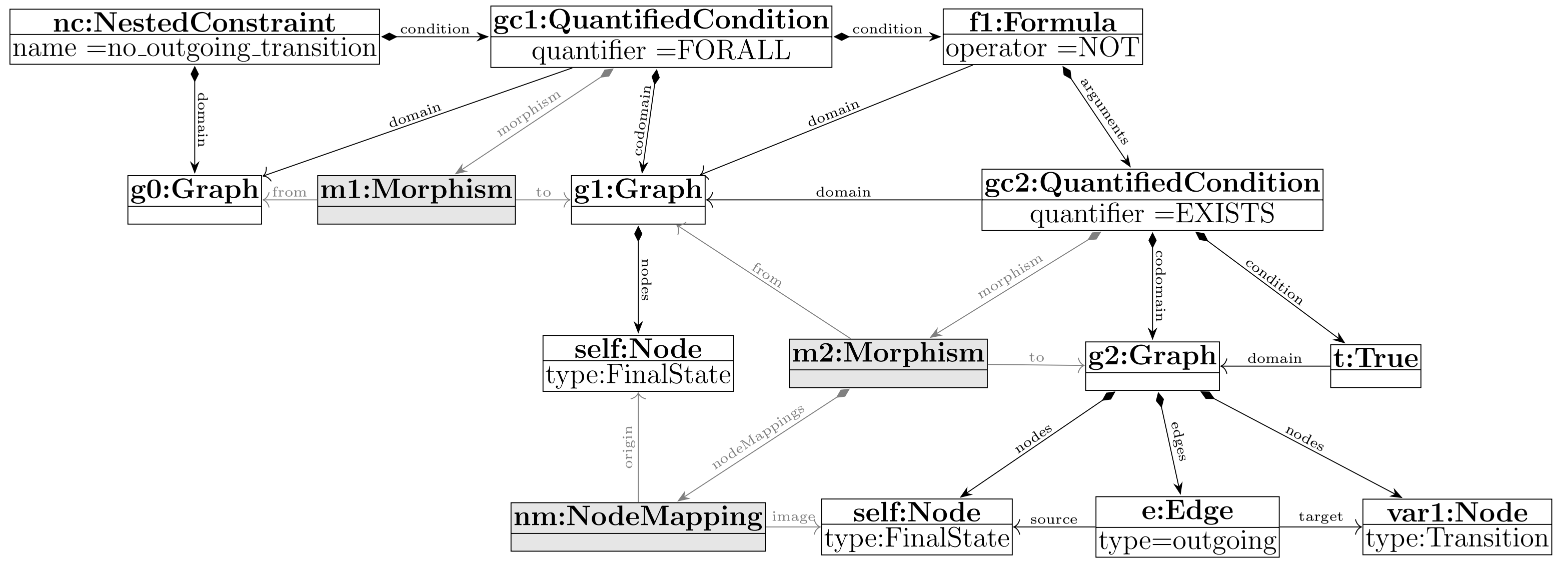


Figure 5.8: The abstract syntax of the completed form of the nested graph constraint no_outgoing_transitions

5.3.4 Tooling

In this section, we give an overview of the architecture of our developed tool with the help of the UML component diagram. Additionally, we discuss its features and limitation and present the test as well.

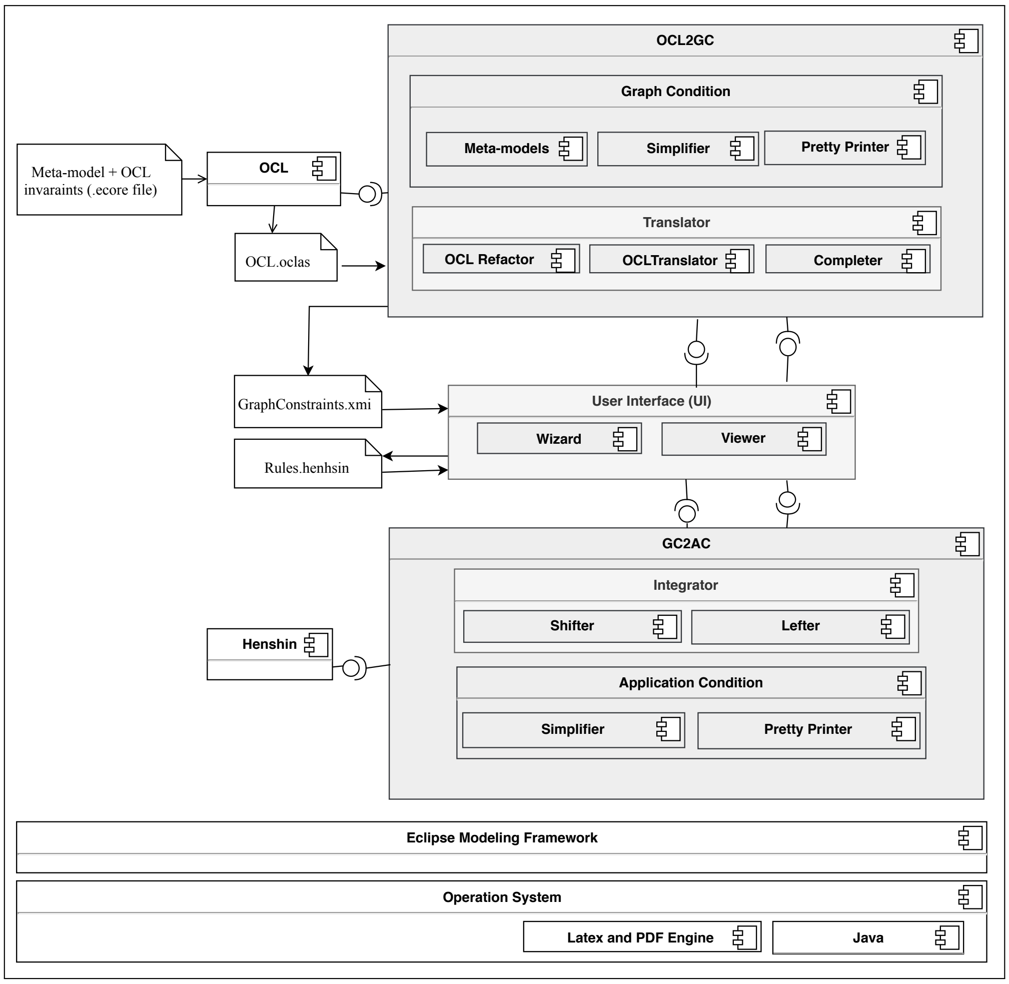


Figure 5.9: An overview of the architecture of our tool *OCL2AC*

The developed tool *OCL2AC* consists of several components depicted in Figure 5.9 and described as follows:

- *OCL2GC*: It has two main components: One is for specifying, simplifying, and displaying graph conditions (*Graph Condition*) and the other one is for refactoring and translating OCL constraints to graph conditions (*Translator*). *OCL2GC* takes a domain meta-model (typed by Ecore) and a set of OCL constraints as inputs and automatically returns a set of semantically equivalent graph constraints as an output.

- *User Interface*: It is for handling the user inputs and displaying the outputs.
- *GC2AC*: It comprises two main components: One is for integrating a graph constraint as a constraint-guaranteeing application condition into a rule (*Integrator*) and the other one is for simplifying and displaying application conditions (*Application Condition*). *GC2AC* takes a transformation rule defined in Henshin and a graph constraint as inputs and automatically returns the Henshin rule with an updated application condition respecting the given graph constraint. Note, *GC2AC* does not yet check in advance whether the given transformation rule with its application condition does already guarantee the constraint or not.

OCL2GC uses the functionalities of the OCL plug-in [35] to extract the OCL constraints as a model file. *OCL2GC* takes the OCL model file as an input and returns a set of semantically equivalent graph conditions as an output. The output is provided as an instance model of the meta-model for representing graph conditions depicted in Figure 5.7. The component *User Interface* takes a Henshin model file (including rules) and the model file of graph constraints as inputs, and the output is the Henshin model file including the rules being updated. This component mainly calls the functionalities of the component *GC2AC*. *GC2AC* uses the Henshin plug-in [61] to deal with rules specified in Henshin. All the developed components are developed as plug-ins on top of the Eclipse Modeling Framework and presented as grey-colored components in Figure 5.9.

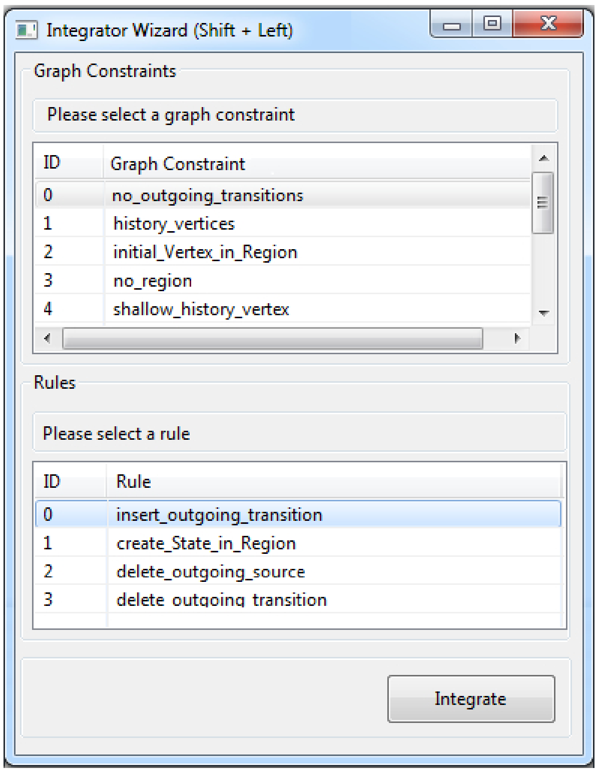
Both components *OCL2GC* and *GC2AC* can be used independently as an Eclipse-based tool. The domain developer can use both components to automate the whole process. The tool is available for download at [142].

Features

OCL2AC additionally provides the following features:

Wizard. *OCL2AC* provides a user-friendly wizard for selecting a rule and a graph constraint that shall be integrated, as shown in Figure 5.10. The inputs of the wizard are a Henshin model (file) and a graph constraint model being generated by *OCL2GC* or manually designed w.r.t the developed meta-model for graph conditions.

Pretty printing. *OCL2AC* additionally provides tool support for *pretty printing graph constraints and application conditions* of Henshin rules in a graphical form as shown in Figures 5.2 and 5.4 above. Pretty printing is supported for both compact and detailed representation of nested constraints and application conditions. The pretty printer is developed as an additional Eclipse plug-in which automatically derives the corresponding LaTeX-representation. The input can be a graph constraint or a Henshin rule (for printing the left application condition); the outputs are in LaTeX and in PDF (being rendered from the LaTeX-output) as shown in Figure 5.11. Additionally, an


Integrator Wizard (Shift + Left)
Graph Constraints
Please select a graph constraint
Rules
Please select a rule
Integrate

ID	Graph Constraint
0	no_outgoing_transitions
1	history_vertices
2	initial_Vertex_in_Region
3	no_region
4	shallow_history_vertex

ID	Rule
0	insert_outgoing_transition
1	create_State_in_Region
2	delete_outgoing_source
3	delete_outgoing_transition

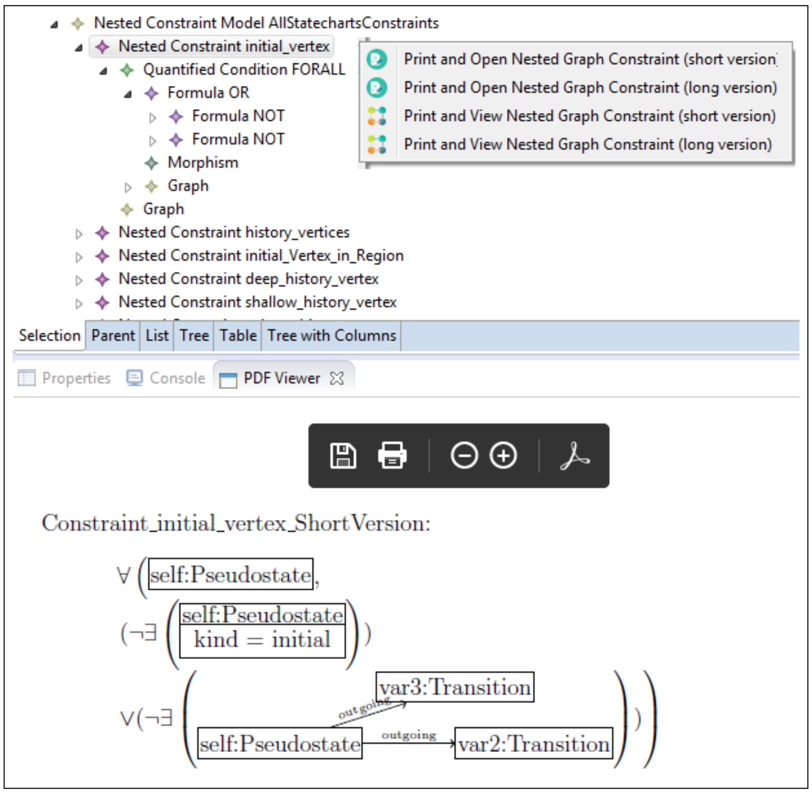
Figure 5.10: User-friendly wizard for the integrator

Eclipse PDF viewer has been developed to display results which can be printed or saved, thereafter.

Constraints as application conditions. Our tool *OCL2AC* can be used to integrate a given constraint as a left application condition into an *empty* rule (the LHS and RHS are empty graphs), and thus we get a rule which checks if the model is consistent w.r.t. the constraint. The updated rule is applicable if and only if the input model satisfies the constraint. For example, integrating the constraint *no_outgoing_transitions* into an *empty* rule produces an application condition being semantically equivalent to the constraint as shown in Figure 5.12. Thus, the rule with the resulting application condition is applicable if and only if the input model is consistent w.r.t. the constraint, i.e., if the model has no *FinalState*-node being connected to a *Transition*-node with an *outgoing*-edge. Such a rule can be used to assert a model w.r.t. the constraint, for example.

Limitation

Since the theory beyond our technique considers a first-order, two-valued logic and relying on sets as the only collection type, our first component *OCL2GC* supports



Nested Constraint Model AllStatechartsConstraints
Nested Constraint initial_vertex
Quantified Condition FORALL
Formula OR
Formula NOT
Formula NOT
Morphism
Graph
Graph
Nested Constraint history_vertices
Nested Constraint initial_Vertex_in_Region
Nested Constraint deep_history_vertex
Nested Constraint shallow_history_vertex

Print and Open Nested Graph Constraint (short version
Print and Open Nested Graph Constraint (long version)
Print and View Nested Graph Constraint (short version)
Print and View Nested Graph Constraint (long version)

Constraint_initial_vertex_ShortVersion:

$$\forall\left(\boxed{\text{self:Pseudostate}},\ (\neg\exists\left(\boxed{\begin{array}{c}\text{self:Pseudostate}\\ \hline \text{kind} = \text{initial}\end{array}}\right))\ \vee(\neg\exists\left(\begin{tikzcd} & \boxed{\text{var3:Transition}} \\ \boxed{\text{self:Pseudostate}} \arrow[ur, "\text{outgoing}"] \arrow[r, "\text{outgoing}"] & \boxed{\text{var2:Transition}}\end{tikzcd}\right))\right)$$

Figure 5.11: The visualization of a nested graph constrain

$$\neg\exists\left(\boxed{\text{self:FinalState}},\ \exists\left(\boxed{\text{self:FinalState}}\xrightarrow{\text{outgoing}}\boxed{\text{var27:Transition}}\right)\right)$$

Figure 5.12: Application condition being semantically equivalent to the constraint *no_outgoing_transitions*

a slightly restricted subset of Essential OCL: *OCL2GC* translates OCL constraints corresponding to first-order, two-valued logic and relying on sets as the only collection type. These restrictions are revealed in the restricted use of collection operation size as well as the omission of operation iterate and types void and invalid. Moreover, operations associated with collections not being sets (like isUnique) or with sets of primitive values (like sum) are not supported. Likewise, the translation of user-defined operations is not supported. Operations defined in OCL are supported as long as they are not recursive and follow the restrictions above. Pairing with the technique's functionality, we focus on OCL invariants and neglect other uses of OCL. Therefore, the operation oclIsNew is not supported. Moreover, there is only limited support to integrate constraints on attributes into Henshin rules that perform complex attribute computations. However, in Section 5.4.1 we show that the supported subset of OCL is sufficient even for a complex example.

Test

We applied our tool *OCL2AC* to the PetriNet meta-model being enriched with 11 OCL constraints (listed in Table B.1). The results of translating the OCL constraints to graph constraints are presented as concrete examples for the OCL translation rules of the theoretical foundation of the tool [114]. Furthermore, we applied the tool to another meta-model, namely the Statechart of the MagicDraw with 11 OCL constraints also (listed in Table B.2). Here, we calculated the results manually. In both cases, the received outputs using our tool conform to the theoretical results.

After that, the tool is applied to integrate the 11 OCL constraints of the Statechart of the MagicDraw as application conditions into a set of 15 editing rules of different kinds. First, we integrated one constraint into a rule and compared the output with the result of the manual calculation. The outputs are equivalent. Second, we integrated all the 11 constraints into each rule. Since the integration of all constraints results in a large application condition which is difficult to be checked manually, we performed the following tests: each enhanced rule is applied to a valid model at a specific match. Then, the output of the rule application is compared with the output of the EMF validator after applying the original rule at that match. Moreover, we checked the outputs of applying the rules at several matches. In both cases, the comparison results are identical. For these tests, we used valid models of sizes between 800 to 16000 elements covering all the meta-model types.

5.4 Evaluation

Research questions (RQs). Since the complexity may arise when translating a graph constraint to an application condition, and thus the translation time may be an obstacle to usability, we want to figure out the effectiveness of our technique in practice. Therefore, we ask the following research questions:

RQ 1:	How restrictive is our OCL translation?
RQ 2.1:	Is the complexity of a given OCL constraint comparable to that of the resulting graph constraint and what is the worst complexity of resulting graph constraints?
RQ 2.2:	What is the worst complexity of resulting application conditions and what is the number of graphs of the resulting application conditions?
RQ 3:	How fast is the translation process and how fast is the integration process?

5.4.1 Restrictiveness of Our Technique

To answer RQ 1, we investigated the OCL-constraints coming with the UML meta-model. More concretely, we considered the Eclipse plug-in org.eclipse.uml2.uml for

UML. OCL constraints are implemented as EMF operations, potentially using further operations as helpers. We extracted 406 invariants from the documented operations, which we further investigated. 32 invariants are not well-formed, well-written, or just empty. 200 of the remaining 374 well-formed invariants are trivial, i.e., equal to true or false. 104 of the remaining 174 non-trivial ones can be directly translated. The other 70 invariants use helper operations. By analyzing their source code, we found that 22 of these 70 invariants use recursive operations or non-supported OCL operations, the other 48 invariants use non-recursive operations. Hence, their operations can be preprocessed by inlining them into the invariants. In summary, 152 invariants out of 174 can be translated, i.e., more than 87 %.

5.4.2 Complexity Considerations

To answer RQ 2, we characterize the complexity of our OCL translation by considering the space complexity of the outcomes dependent on input constraints. We consider both components of our technique separately.

Experiment set-up. We consider the case study presented in [66] where editing rules are automatically and manually generated for the Statechart of MagiDraw [80]. Out of 17 original constraints, they identified 11 as necessary to be valid to edit a model in MagicDraw. Table B.2 presents the OCL expressions. In total, they used 84 editing rules for Statecharts. Those are suitable for our purpose, as rules of different sizes and for all kinds of actions and elements of the meta-model are provided. Note that, for all graph conditions, we evaluated the resulting compact conditions, since the completed graph conditions, i.e., the detailed version containing morphisms and their domains, are just internally needed for the tool.

Translation of OCL Constraints

We characterize the complexity of our first component which translates OCL constraints to graph constraints. Let n denote the number of abstract syntax elements of an OCL constraint (as they occur in the OCL meta-model). We estimate the number of syntax elements in the corresponding compact condition along the translation steps presented above. In Step (1), refactorings do constant pattern replacement, hence they do not change the complexity of a constraint significantly. Step (2) does the proper translation. Two of the translation patterns lead to an increase of model elements in the graph constraint: (a) Translation of v.role creates not only an edge of kind role but additionally the corresponding target node. (b) Translation of operation size() >= c creates c graph elements. Hence the size of the resulting graph constraint is roughly $n + \sum_i c_i$ with $\sum_i c_i$ being the sum of those constants the size-expressions of the constraint are compared to. Steps (3)–(5) just simplify constraints.

In practice, the values of occurring constants are usually pretty small (such as 1 or 2), so using operation size does not harm.

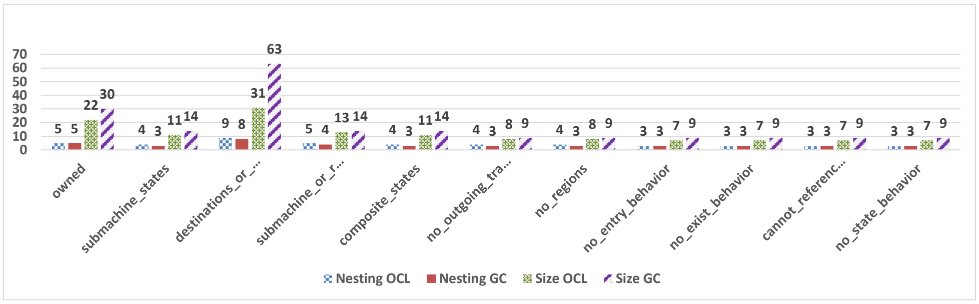


Figure 5.13: Comparision of complexity of OCL and GC

Complexity experiment. As an application case, we translated the 11 OCL constraints of the Statechart of the MagicDraw to graph constraints. To compare the complexity of results, we used several simple *complexity metrics*. When comparing OCL with graph constraints, we measured two factors: (1) size and (2) nesting depth. To measure the size of constraints, we just compared the number of elements in abstract syntax structures w.r.t. the corresponding meta-models. For nesting depth, we basically used the deepest nesting level of their abstract syntax trees. However, for OCL constraints, we did not count navigation expressions and, for nested graph constraints, we counted occurrences of $\exists G$ as one level. We tried to compare the logical nesting one has to follow when reading a constraint.

Applying the first component *OCL2GC* to the Statecharts meta-model with 11 constraints leads to metric values summarized in Figure 5.13.

The number of elements in the resulting graph constraints is increased by one-third on average compared to the original OCL constraints. In the worst case, the number of elements is doubled; in the best case, only one element is added. In contrast, the nesting of the constraints was never increased but decreased (by 1) in 6 of 10 cases.

The increase of elements mostly stems from the inclusion of target nodes: While OCL just navigates along roles, a graph constraint contains an edge and, additionally, its target node for each role. Moreover, a size-comparison can lead to the addition of $2n$ graph elements where OCL just refers to the integer n. These observations suggest an answer to *RQ 2.1*: There is a slight increase in metric values on average. We can argue, however, that it is accompanied by a gain of information: The context of attributes and associations, i.e., containing or target nodes, becomes visible, which might help in understanding the constraint. Considering that not all possible simplifications are implemented yet and that the nesting could be even further reduced in some of the resulting graph constraints, the translation of OCL constraints into graph constraints seems to be a promising way to ease the comprehension of constraints: Context information is added while structural complexity is reduced. However, the comprehensibility of a graph constraint is highly dependent on its visual organization. It is up to future work to develop a technique for condensing graph conditions considerably so that users can understand all the application conditions a rule has to fulfill.

Integration of Graph Constraints

To answer RQ 2.2, we roughly sketch how constraint complexity grows in the worst case when computing application conditions from a given graph constraint and a rule. The preparation in Step (1) and the computation of the left application condition in Step (3) do not change the complexity significantly. But in Step (2) all possible overlappings of the RHS of the rule with the prepared constraint need to be computed. Let n denote the number of nodes in the first graph of the constraint and m the number of nodes in the RHS of the rule. In the worst case (untyped graphs containing just nodes) this results in $\sum_{k=0}^{\min(n,m)} \binom{n}{k}\binom{m}{k}k! \leqslant (n+1)!m!$ different graphs with m to $n+m$ nodes each (compare also [105]). Further, each of the resulting graphs needs to be overlapped in each possible way with the graphs following on the deeper levels of nesting of the constraint. In practice, however, the size of an application condition is much more restricted. Due to node typing and graph structure, many node overlappings are not possible.

Complexity experiment. To find out how far this blow up of application conditions is a problem in practice, we conducted the following experiments considering the number of graphs as well as the number of nesting levels in application conditions.

To answer *RQ 2.2*, we compare the number of graphs of a nested graph constraint with the number of graphs of the resulting application condition. For a rule which already has an application condition, we subtract the number of its graphs from the result. Given the 11 OCL constraints of our application case, we translated them to graph constraints containing 2 to 10 graphs (36 in total) and integrated all of those in each of the 84 rules using *OCL2AC* (i.e., computing the guaranteeing application conditions). The newly added application conditions contain 77.3 graphs on average (with 36 being the best and 191 being the worst case) and 6 nesting levels. Thus, on average the number of graphs more or less doubles.

5.4.3 Performance Experiments

To answer RQ 3, we re-considered both set-ups, for UML and for Statecharts. Across 10 test runs, a set of 104 OCL constraints of the UML meta-model (see Section 5.4.1) was translated in 2.4 seconds on average. When integrating the 11 different graph constraints of the Statecharts meta-model into the 84 editing rules, the tool took only 2.3 seconds to update all the rules w.r.t. all the 11 constraints. All evaluations were performed with a desktop PC, Intel Core i7, 16 GB RAM, Windows 7, Eclipse Neon, Henshin 1.4.

Summarizing. Both components of *OCL2AC* work fast. The translation of OCL to graph constraints is promising to increase the comprehensibility of constraints, though a slight overhead in size is produced. Application conditions resulting from the integration process tend to be more complex than the original graph constraints.

In most cases, the growth is moderate. However, there are cases where the size of the results become too complicated. Further work is needed to investigate the circumstances for such a blow-up.

5.4.4 Threats to Validity

Although the chosen metrics are quite simple and we considered a limited number of OCL constraints and rules, the metrics are expressive enough to argue about complexity basically, and the OCL and rules examples are stemmed from standard use cases and meta-models (of statecharts and UML). Concerning the considered OCL constraints of the statecharts, it can be noticed that about half of them are not complex. However, all core features of OCL (logical operators, navigation expressions, and collection operators) are covered, and at least one rather complex constraint is included. Nevertheless, further larger case examples are interesting to be considered in the future. In Section 5.4.1 we used the Eclipse plug-in org.eclipse.uml2.uml for UML instead of the original UML specification since it allows us to automatically check the well-formedness of each constraint. Additionally, we noticed that some of the translated constraints cannot be persisted even they are well translated because UML uses non-persistence types, i.e., preventing their objects from being persisted.

5.5 Related Work

We first relate our work to other translations of OCL and subsequently to other calculations of application conditions. Note that the formal approach of our tool is compared to its related work in [55] and [114].

Translations of OCL. As pointed out in [53], model verification approaches are typically structured into two phases: formalization (e.g., translating a model with its OCL constraints to some formalism) and reasoning within that formalism, which usually means checking satisfiability. Approaches are either automatic but bounded (i.e., configuring the boundaries of the solution space) such as [26, 27, 52], or semi-automatic as [20]. For a given model with constraints, the output of an automatic supporting tool is a valid instance model. In contrast, our approach is unbounded (i.e., static) and automatic but works on a slightly restricted form of OCL only and yields a different output. Given a model with constraints and a set of rules, our technique provides all application conditions for the given rules to guarantee that models resulting from rule applications are valid w.r.t. the given constraints.

The translation processes in [13] and [116] are the ones most similar to ours since they also translate into graph-based languages: In [13], Bergmann proposes a translation of OCL constraints into so-called graph patterns. His work focuses on the possibility to validate models more efficiently compared to other OCL validators. That approach supports a similar, but a slightly larger subset of OCL. The correctness of that translation is not shown. The implementation covers only a subset of that translation. In

[116], Richa et al. consider ATL rules and translate them into Henshin rules. The guards of ATL rules (defined in OCL) are translated into application conditions of Henshin rules. To this end, they adapt and extend the translations proposed in [4] and [13].

Calculation of application conditions. In [106], Pennemann introduces ENFORCe which can check and ensure the correctness of high-level programs. It integrates graph constraints as left application conditions of rules as well. However, the tool is not published and it is not compliant with EMF. Furthermore, there is no translation from OCL to graph constraints available.

Clarisó et al. present in [28] how to calculate an application condition for a rule and an OCL constraint, directly in OCL. The supported subset of OCL is slightly larger than ours because staying with OCL, they can support operations that are not first-order. But they do not distinguish between conditions (on matches for rules) and constraints (on model graphs), which leads to a certain imprecision: Either they would need to consider their preconditions as queries to retrieve contexts to which the rule can be applied, or use rule matches as contexts to evaluate the constraints. This is what our tool does automatically by calculating an application condition for a given rule. They provide a correctness proof and a partial implementation.

To the best of our knowledge, we present the first ready-to-use tool for integrating OCL constraints as application conditions into transformation rules.

5.6 Conclusion

Our correct-by-construction technique *OCL2AC* takes OCL constraints and calculates automatically (graph-based) application conditions for transformation rules such that these rules become constraint-guaranteeing. Its first component *OCL2GC* translates OCL constraints into semantically equivalent graph constraints. Its second component *GC2AC* takes a graph constraint and a Henshin rule as inputs and updates the application condition of the rule such that the rule becomes *constraint-guaranteeing. OCL2AC* is a ready-to-use tool implemented as an Eclipse plug-in based on EMF and Henshin. Although the theory shows that the size of a computed application condition (in terms of a number of graphs) can grow over-exponentially in the worst case, we figured out the feasibility of our developed technique in practice. Tests with a larger case example showed that our technique (both components) works reasonably fast and that the translation of OCL to graph constraints can reduce the complexity. Regarding the complexity of the resulting application conditions, the number of graphs is more or less doubles compared to the size of the graph constraints. Although these results are far better than could be expected from theory, the number of graphs is still high and the number of levels could be reduced in most cases.

It is up to future work to develop a technique for condensing graph conditions considerably so that users can understand all the application conditions a rule has to fulfill. A promising way is to find sub-conditions, which subsume each other. Subgraph isomorphism is an obvious cause for such subsumption. Application conditions can be further condensed by taking multiplicity constraints into account. Furthermore, we plan to apply our tool in scenarios such as deriving consistency-preserving model editing rules from meta-models [66] and improving model refactoring rules [84]. Additionally, applications to model validation and model repair look promising.

6 Constructing Optimized Consistency-Preserving Application Conditions

*This chapter shares material with the ICGT'19 paper "Constructing Optimized Validity-Preserving Application Conditions" Nassar et al.[90]. (**Best Paper Award**)*

There is an increasing need for graph transformations ensuring consistent result graphs w.r.t. a given set of constraints. In a model refactoring process, for example, each performed refactoring should yield a consistent model graph. At least, it has to remain an element of the underlying modeling language. If a graph transformation starts at a consistent graph already, it is called *consistency-preserving*. Otherwise, it is considered to be *consistency-guaranteeing*. There is a formal construction for graph transformation systems making them consistency-guaranteeing. This is ensured by adding a consistency-guaranteeing application condition to each of its transformation rules. This theory has been implemented recently as an Eclipse plug-in called *OCL2AC*. Initial tests have shown that resulting application conditions can become pretty large. As there are interesting application cases where transformations just need to be consistency-preserving (such as model refactoring), we started to investigate this case further. We developed optimizing-by-construction techniques for application conditions for transformations that just need to be *consistency-preserving*. All presented optimizations are proven to be correct. Implementing and evaluating them, we found that the complexity of the resulting application conditions is considerably reduced (by factor 7 on average). Moreover, our optimization yields a speedup of rule application by approximately 2.5 times.

6.1 Introduction

Model transformations are the heart and soul of Model-Driven Engineering (MDE). They are used for various MDE-activities including translation, optimization, and synchronization of models [130]. Usually, a transformation (that may consist of several transformation steps) should yield a consistent result model, especially if it has been applied to an already consistent model. Intermediate models may not be required to be consistent as, e.g., argued in [36]. But there are scenarios where even intermediate models have to show consistency, at least a basic one, as the following example applications show: (1) Throughout a larger refactoring process, each performed refactoring should preserve the model's consistency [10]. (2) More generally, any in-place model change should preserve a basic consistency, enough to view an edited model in its domain-specific model editor [66]. Model editors typically ensure the creation of models with basic consistency right from the beginning. This is the application scenario we will use as a running example and for our evaluation. A similar scenario is considered in projectional editing for textual editors [134]. (3) Modeling the behavior of concurrent and distributed systems with model transformations, each model represents a system state that should fulfill system invariants such as safety properties [72]. (4) When generating code from abstractly specified model transformations, the transformations should be consistency-preserving, especially for safety-critical systems [48].

State of the art. From the formal point of view, the theory of algebraic graph transformation constitutes a suitable framework to reason about model transformations [39, 40], in particular about the rule-based transformation of EMF models [14]. Constraints are typically expressed as (nested) graph constraints [115, 55], into which a large and relevant part of OCL [99] can be translated [114]. Graph constraints can be integrated as application conditions into graph transformation rules as shown in [55]. Given a rule and a constraint, there are two variants of integration, namely computing a *constraint-preserving* or a *constraint-guaranteeing* rule. Both computations do not alter the actions of the rule but equip it with an application condition. Graph consistency is *preserved*, if applying an equipped rule to a consistent graph, the resulting graph is consistent as well. Graph consistency is *guaranteed*, if applying an equipped rule to a graph, the resulting graph is consistent. As for tool support, *OCL2AC* [89] automatically translates OCL constraints into graph constraints and integrates these as application conditions into transformation rules specified in Henshin [3]. It computes guaranteeing rules.

Tests of *OCL2AC* have shown that resulting application conditions can become very complex. Theoretically, application conditions of guaranteeing rules grow over-exponentially in the worst case [105]. As there are interesting application cases where transformations just need to be consistency-preserving (as pointed out above), it is worthwhile to investigate consistency-preserving transformations further. Habel and Pennemann [55] present a direct construction of the logically weakest application condition, enough to preserve consistency. As this kind of condition is logically weaker, our expectation was in the beginning that it can be expressed in a simpler form. In contrast, the resulting application conditions may contain even more elements than the consistency-guaranteeing ones. (This is due to the approach taken: The premise that the model was already consistent before rule application is added to the computed consistency-guaranteeing application condition. The resulting condition can be inherently difficult to simplify because of the used material implication operator and the nesting. An example can be found in Section B.1.2.)

Contribution and Structure. Focusing on consistency preserving transformations only, we developed optimizing-by-construction techniques to construct application conditions that preserve the consistency and are considerably less complex than the results of the original construction.

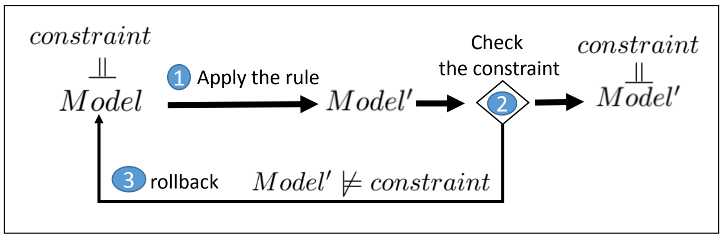
1. In Section 6.3, we take a constraint and a rule as a starting point and construct an application condition that preserves consistency. This construction is based on the construction of the guaranteeing application conditions but simplifies it by omitting parts that check for antecedent consistency, while keeping parts that prevent the introduction of violations. This automatic *approximation* of the preserving application condition is conceptually new and quite general in scope. While some of the simplifications are specific for EMF (Theorem 2), the others (Theorem 1) are proven for graph constraints in general and can be easily lifted to adhesive categories [73]. We will argue how some of these simplifications omit *global* checks that have to traverse the whole model while keeping *local* ones, i.e., checks being performed in the context of a rule match.

2. Practically, we have implemented the techniques as an Eclipse plug-in, called *Optimizer* on top of *OCL2AC* (Section 6.4). The domain developer can use this tool to automate the whole process. Additionally, we compared the application conditions of guaranteeing rules with those of preserving ones. The results show a considerable loss in complexity of application conditions (Section 6.5.1).

3. We provide an application case which shows that consistency-preserving transformations are used in practice. In domain-specific model editing (presented as scenario (2) above), every state of the transformation process has to ensure basic model consistency. The example comprises the MagicDraw Statechart meta-model with 11 OCL constraints and 84 editing rules. The optimizations do not only reduce the size of computed application conditions considerably but also improve the performance of consistency-preserving transformations.

   In addition, we have conducted several evaluations that do not specifically test our optimization but the overall approach.

   There are basically two strategies to achieve consistency preservation by assuming consistent input models: We either test for consistency after each transformation step and rollback the transformation step if its resulting model is not consistent anymore (*a-posteriori consistency check* as shown in Figure 6.1). Or the transformation performs consistency-preserving steps only, i.e., it does not allow any transformation step making a model inconsistent (*a-priori consistency check* as shown in Figure 6.2).



$Model \models constraint$
1 Apply the rule
$Model'$
Check the constraint
2
$Model' \models constraint$
3 rollback
$Model' \not\models constraint$

Figure 6.1: Steps of *a-posteriori* consistency check

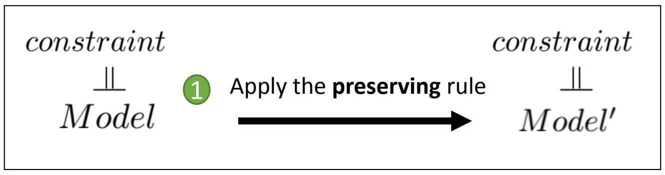


Figure 6.2: Step of *a-priori* consistency check

> We compared the run times of consistency checking after a transformation using existing OCL validators (*a-posterori approach*) with running a consistency-preserving transformation (being enriched with application conditions) with and without optimization (*a-priori approach*) (Section 6.5.2). Results show that both approaches are fast in practice.

We start our presentation with the running example in Section 6.2. Chapter B contains all proofs and more details about the evaluation.

6.2 Running Example

In this section, we illustrate the effect of our optimizations on application conditions computed by *OCL2AC*.

A simple Statecharts language serves as an example. Its meta-model is displayed in Figure 5.1 and explained in Section 5.2.

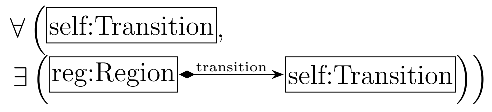


Figure 6.3: Graph constraint for TransitionInRegion

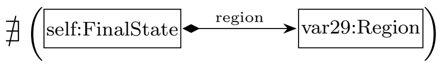


Figure 6.4: Graph constraint for no_region

The UML definition specifies several constraints on statechart models. For example, each Transition is required to be contained in a Region (TransitionInRegion) and a FinalState is forbidden to contain a Region (no_Region). Figures 6.3 and 6.4 show these constraints as graph constraints, respectively. In the UML, however, these constraints are specified in OCL; the OCL constraint for no_region, for example, is specified as

```
context FinalState invariant no_region: self.region->isEmpty()
```

Figure 6.5 shows a simple transformation rule in Henshin taken from [66] for specifying an edit operation in MagicDraw [80]. The rule moves an existing Region from an existing State (the old source) to another existing State (the new source). This is done by deleting the containment edge region from the old source and recreating it in

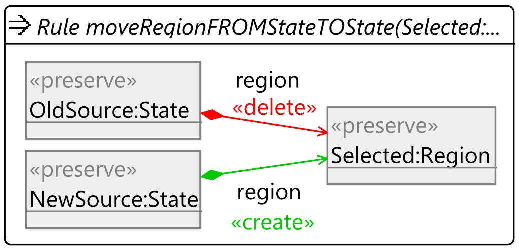


Figure 6.5: Transformation rule in Henshin

the new source. Rules specifying such edit operations may be used, e.g., to recognize semantic change sets while comparing two model versions [65, 66].

The validity of basic constraints should be preserved throughout editing because a typical model editor is not able to display an instance violating them. Since FinalState is a subtype of State, applying the rule moveRegionFromStateToState might introduce a violation of the constraint no_region. Using *OCL2AC* [89], a language engineer can automatically integrate a constraint as an application condition into the rule and calculate the according constraint-guaranteeing version of the rule. The guaranteeing application condition obtained by integrating constraint no_region into rule moveRegionFromStateToState forbids matching this rule to a FinalState. It checks additionally if the model already encompasses a FinalState containing a Region – either matched by the rule or not. Figure 6.6 presents the resulting guaranteeing application condition, which is composed of 7 graphs (explained later in Section 6.3.1). Knowing the input model to be valid (consistent), most of the checks are unnecessary, especially the checks which do not only involve elements being local to the rule application but amount to traversing every existing node, i.e., the global checks.

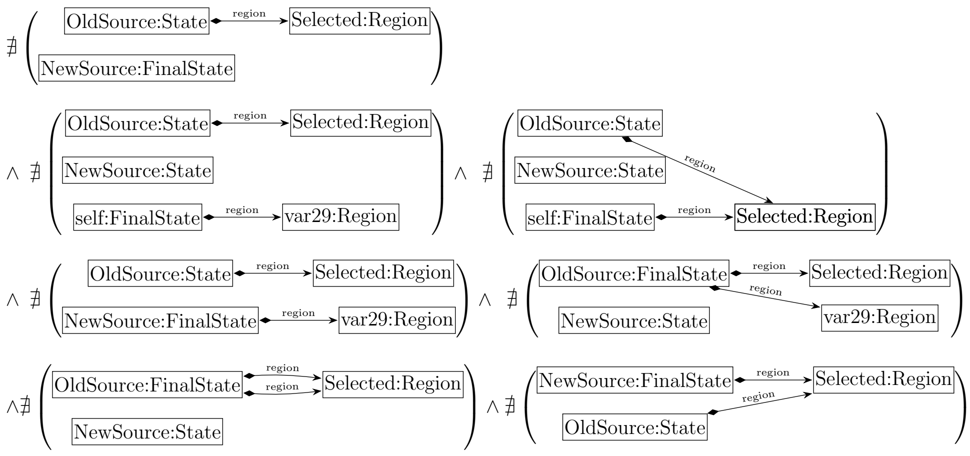


Figure 6.6: Non-optimized application condition for moveRegionFromStateToState after integrating the constraint no_region

In this chapter, we develop and implement optimizations that allow for omitting

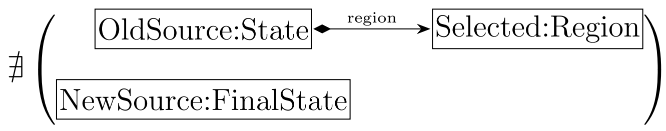


Figure 6.7: Optimized application condition for moveRegionFromStateToState still preserving the constraint no_region

certain parts from the construction of a guaranteeing application condition. In our example, we will arrive at the optimized application condition shown in Figure 6.7, which consists of only one graph that, moreover, only requires a local check. It forbids the rule node newSource:State to be matched to a FinalState.

As the rule moveRegionFromStateToState does not change any graph element occurring in constraint TransitionInRegion, this constraint cannot be violated by a result model if it was not violated before. Hence, the optimized application condition is just true. The guaranteeing condition (not shown), however, consists of three graphs. Thus, assuming valid input models, guaranteeing application conditions can be considerably simplified.

6.3 Optimizing Application Conditions

The application conditions being calculated by the approach of the tool *OCL2AC* guarantee consistency even if the input is not a consistent EMF-model. Since we focus on the consistency preservation of EMF-models in this work, the calculated conditions can be considerably simplified. In this section, we investigate several strategies to construct optimized consistency-preserving application conditions.

6.3.1 Approximating Preservation

In common application scenarios (like refactoring), a user can assume that rules are applied to instances showing a certain consistency. Hence, when applying a rule, an already valid constraint does not need to be guaranteed but just preserved. The construction Pres of a preserving rule (as mentioned in Fact 1) takes this into account. Though being logically weaker, the resulting application condition, however, can be even more complex with respect to the structure and the number of contained graphs and simplification is inherently difficult. Nevertheless, it is possible to simplify guaranteeing application conditions during the construction process if they just need to preserve consistency. In the following, we present three forms of simplification.

1. We collect all rule elements being deleted or created and check if this set overlaps with the set of all constraint elements. If this overlap is empty, the resulting preserving application condition is just true.

2. If a rule creates new graph structure only, positive constraints $\exists\, C$ do not need to be integrated into such a rule. Analogously, if a rule only deletes graph structure, negative constraints $\neg\exists\, C$ do not need to be integrated. In both cases, applications of such a rule cannot introduce a new violation of the constraint. Hence, the optimized application condition is just true.

3. When calculating an application condition, a constraint graph is overlapped with the RHS graph of a rule in all possible ways. For negative constraints $\neg\exists C$ it is not necessary to consider all possible overlappings. One may omit all the cases where C and the RHS R do not overlap in at least one element created. The parts of the application conditions arising from those cases would just check that the input graph already fulfills the constraint.

The third simplification omits cases where the arising graph in the application condition contains nodes not connected to nodes of the LHS of the rule, thus amounting to global checks upon application. We state the correctness of these simplifications in the following theorem. Figure 6.8 illustrates the theorems and their applications during the optimization process.

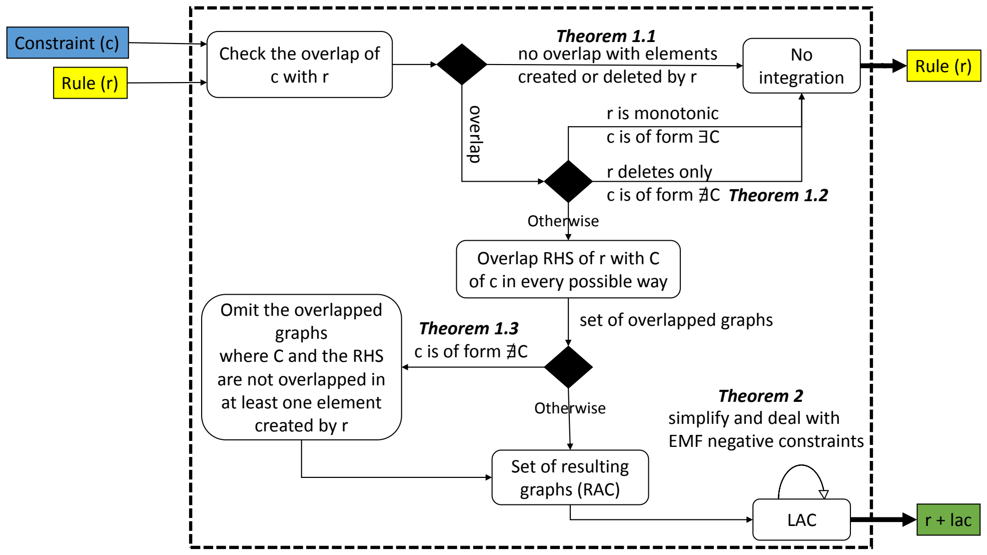

Constraint (c)
Rule (r)
Check the overlap of c with r
Theorem 1.1
no overlap with elements created or deleted by r
No integration
Rule (r)
overlap
r is monotonic
c is of form ∃C
r deletes only
c is of form ∄C
Theorem 1.2
Otherwise
Overlap RHS of r with C of c in every possible way
Omit the overlapped graphs where C and the RHS are not overlapped in at least one element created by r
Theorem 1.3
c is of form ∄C
set of overlapped graphs
Otherwise
Theorem 2
simplify and deal with EMF negative constraints
Set of resulting graphs (RAC)
LAC
r + lac

Figure 6.8: Optimization process based on the theorems

Theorem 1 (Correctness of simplifications). Let c be a graph constraint and $p = (L \hookleftarrow K \hookrightarrow R)$ be a plain rule. Let $\rho = (p, ac)$ be the same plain rule equipped with the application condition ac computed in one of the following ways:

1. If both the elements of $L\backslash K$ and the elements of $R\backslash K$ intersect emptily with every graph C occurring in the constraint c, then $ac = \texttt{true}$.

2. If p is monotonic and c is a positive constraint, then $ac = \texttt{true}$. Analogously, if p only deletes and c is a negative constraint, then $ac = \texttt{true}$.

3. If $c = \neg\exists\, C$, let $Gua(p,c)$ yield the right application condition $rac := \neg(\bigvee_{i\in I} \exists\, P_i)$ with morphisms $c_i : C \hookrightarrow P_i$ and $r_i : R \hookrightarrow P_i$. Let $rac_{pres} := \neg(\bigvee_{j\in J} \exists\, P_j)$ with $J \subseteq I$ including only those P_i where $c_i(C) \cap r_i(R\backslash K) \neq \emptyset$. Then ac is the application condition that arises by translating the right application condition rac_{pres} to the LHS of rule p.

Then for all transformations $G \Rightarrow_{\rho=(p,ac)} H$ where $G \models c$ also $H \models c$.

The proof follows a common pattern in all cases: Checking for the (non-)existence of graphs occurring in the constraint in all these cases is *sequentially independent* from the application of the rule. Hence, checking the constraint for validity always gives the same result, no matter if done before or after rule application.

Example 1 (compare Section 6.2). Constraint no_region is required to be integrated into rule moveRegionFromStateToState since a region-edge is created by this rule and contained in this constraint. Figure 6.6 shows the guaranteeing application condition. The first graph (the uppermost graph) results from a maximal overlapping of the constraint with the rule. Note that it is possible to identify nodes of types State and FinalState since FinalState is a subtype of State (compare Figure 5.1). The second graph results from copying the graph of the constraint and the RHS of the rule and putting them next to each other. The third graph results from merging the nodes of type Region. The forth and the fifth graph result from just merging nodes of type State and FinalState. The sixth and the seventh graph result from merging the nodes of type State and the nodes of type Region. In every case, the overlapping of the constraint with the RHS is then translated to the LHS of the rule.

Our proposed optimizations lead to the result displayed in Figure 6.7 by the application of Theorem 1, 3.: Except for the sub-condition containing the uppermost graph, all other sub-conditions in Figure 6.6 are omitted. The uppermost one has to be saved because the region-edge created by the rule is overlapped with the region-edge of the constraint. The omitted sub-conditions do not only involve elements being local to the rule application but amount to traversing every existing FinalState leading to global checks. To conclude, only one local check remains.

Example 2 (compare Section 6.2). The constraint TransitionInRegion is not required to be integrated into the rule moveRegionFromStateToState. Theorem 1, 1. justifies this: the rule moveRegionFromStateToState does not have any effect on the validity of the constraint since its application neither deletes nor creates elements that occur in the constraint.

6.3.2 Dealing with EMF's Built-in Negative Constraints

EMF has several built-in constraints [14]. Instance models that do not satisfy these EMF-constraints cannot even be opened in the EMF-editor. Most of these constraints are negative, i.e., they forbid certain patterns in instances to exist. Concretely, cycles over containment edges, nodes with more than one container, and parallel edges, i.e., two edges of the same type between the same two nodes, are forbidden. Therefore, given an application condition ac of a rule p, each occurrence of a sub-condition of the form $\exists\, A$ with A violating one of these EMF constraints, may be replaced by false without altering the meaning. We know that such patterns cannot appear in any EMF instance model. Thus, in the context of EMF, the result is semantically equivalent to the actual guaranteeing rule but may contain fewer sub-conditions.

Theorem 2 (Correctness of EMF-specific simplifications).

Let c be a graph condition over P and c' be the condition that results from replacing every occurrence of a sub-condition $\exists(a : C_1 \hookrightarrow C_2)$ of c by `false` if the graph C_2 contains parallel edges or multiple incoming containment edges to the same node. Then an injective morphism $p : P \hookrightarrow G$ into an EMF-model graph G satisfies c if and only if it satisfies c'. In particular, if c is a graph constraint, any EMF-model graph G satisfies c if and only if it satisfies c'.

The correctness of this theorem is proven by induction along with the nesting structure of the constraint in the cases of parallel edges and multiple containment nodes. The same argument also applies in the case of finite containment cycles. But since containment cycles of arbitrary length cannot be expressed as graph constraints, the correctness of replacing their occurrence by `false` is intuitive but not amenable to a formal proof by induction.

Example 3 (compare Section 6.2). Theorem 2 would drop the third, sixth and seventh sub-conditions from the application condition in Figure 6.6 by replacing it with false since it contains a node with more than one container or parallel edges.

6.4 Tooling

We developed our optimizer as an Eclipse plug-in on top of *OCL2AC* implementing all of the proposed simplifications except for the elimination of containment cycles. Given a Henshin rule and a graph constraint, our optimizer automatically renders the rule to preserve the validity of the constraint.

The optimizer consists of two main components depicted in Figure 6.9 and described as follow:

- **Analyzer:** It detects if a constraint needs to be integrated into a given rule, i.e., it implements (Theorem 1, 1 and 2).
- **Omitter:** It eliminates unnecessary subconditions from the constraint-guaranteeing application conditions during the construction process, i.e., it implements (Theorem 1, 3 and Theorem 2).

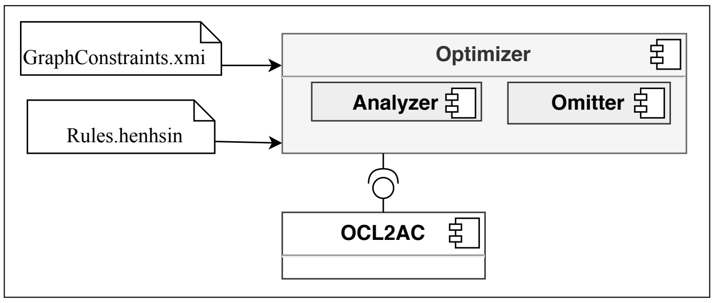


Figure 6.9: An overview of the architecture of our tool *Optimizer*

Additionally, we implemented simplifications of application conditions by applying well-known equivalence rules [105]:

$$\begin{aligned}
\exists\,(C_1, \exists\, C_2) &\equiv \exists\, C_2 \text{ if } C_1 \subseteq C_2\\
\exists\, C_1 \vee \exists\, C_2 &\equiv \exists\, C_1 \text{ if } C_1 \subseteq C_2\\
\exists\, C_1 \wedge \exists\, C_2 &\equiv \exists\, C_2 \text{ if } C_1 \subseteq C_2
\end{aligned}$$

A user can choose to automatically apply them. Applying these, entire graphs may be omitted and even levels of nesting may be collapsed.

The tool can be downloaded from our website.[1]

6.5 Evaluation

In this section, we show the highlights of our evaluation; the artifacts can be downloaded from the website of our tool.

Research questions (RQs). Our evaluation aims to answer the following RQs regarding the complexity and performance:

RQ 1: How complex are the resulting application conditions with and without optimizations? How does this compare to the complexity of the original graph constraints?

To perform consistency-preserving steps, there are two basic approaches: We either test for consistency after each transformation step and rollback the step if its resulting model is not valid (*a-posteriori* check) or the transformation is designed to perform consistency-preserving steps only (*a-priori* check). We, therefore, ask the following questions:

[1]https://ocl2ac.github.io/home/

RQ 2.1: How fast is the *a-priori* consistency check compared to the *a-posteriori* check? **RQ 2.2:** Does the optimization of application conditions improve the performance significantly?

General set-up. As an application case, we consider the scenario of in-place model transformations that should preserve a basic consistency such that the resulting instances can be opened in a domain-specific model editor throughout. In [66], Kehrer et al. derive consistency-preserving editing rules from a given meta-model. However, they support basic constraints like multiplicities only. More complex OCL constraints are left to future work. In their evaluation, this restriction has the most serious impact on the UML meta-model for Statecharts [100]. Out of 17 original constraints, they identified 11 to be enforced in MagicDraw [80]. In total, they used 84 editing rules for Statecharts.

We translated those 11 OCL constraints into graph constraints and then integrated them as application conditions into the 84 rules.

7 valid test models of sizes between 800 to 16 000 elements (nodes and references) are used to conduct our performance experiments. These test models are synthetic containing copies of an initial valid model composing 5 objects of each non-abstract class of the meta-model. All evaluations were performed with a desktop PC, Intel Core i7, 16 GB RAM, Windows 7, Eclipse Neon, Henshin 1.4.

6.5.1 Evaluating Complexity

In theory, the size of a computed application condition (the number of graphs) can grow over-exponentially in the worst case compard to the size of the original constraint [105]. In practice, however, the growth is moderate. Mainly due to node typing, many node overlappings are not possible. To find out how far this blow up of application conditions is a problem in practice, we conducted the following experiments considering the number of graphs as well as the number of nesting levels in application conditions. Additionally, we explore how far the complexity can be reduced using our optimization. Table 6.1 gives an overview of the results.

Integration without and with optimization. Given the 11 OCL constraints and the 84 editing rules, the newly added application conditions contain 77.3 graphs on average (with 36 being the best and 191 being the worst case) and 6 nesting levels as we have shown in Section 5.4.2. Nonetheless, the number of graphs is way too high and also the number of levels should be smaller in most cases. Hence, there is a clear need to optimize the resulting application conditions further.

To find out how efficient our optimizations of application conditions are, we conducted the same experiment as above using our developed optimizer. In a result, the average number of graphs in the application condition is 10.8 (with 0 being the best and 35 being the worst case), i.e., the complexity is reduced by a factor o 7 on average using our optimizer. Additionally, the deepest nesting level of 6 was often reduced to at most 2 levels. Theorem 1,1 turns out to be the main reason behind this considerable loss of complexity: Instead of integrating 11 constraints into each rule, on average, only 1.7 constraints are integrated into a rule.

Table 6.1: Numbers of graphs of application conditions and deepest nesting levels before and after optimization (with emphasis on extreme cases)

	w/o optimization		**w optimization**		
Rule	#graphs	level	#graphs	level	#integrated constraints
create_Transition	191	6	1	1	(1)
create_FinalState	44	6	31	6	(11)
delete_Trigger	37	6	0	0	(0)
Average (84 rules)	77.3	6	10.8	2.6	1.7

Table 6.1 shows extreme cases: Considering all 84 rules and the 11 constraints, the best optimization was reached with rule create_Transition where the resulting application condition with 191 graphs was reduced to a condition with just one graph. One of the lowest optimizations came along with rule create_FinalState. Since it is overlapped with all the 11 constraints, the number of the resulting graphs is reduced by a factor of 1.4 only (using Theorem 1, 3). Rule delete_Trigger started with one of the lowest number of graphs in their application conditions. This condition is eliminated altogether using our optimization.

The average time of integrating the 11 graph constraints for statecharts into all 84 rules was 2.3 sec. without optimization and 1.03 sec. with optimization. Hence calculating all needed application conditions for a given rule set is fast enough to be used in practice.

To answer RQ 1, given graph constraints with 2–10 graphs (3.2 on average) and 2–6 nesting levels (2.3 on average), non-optimized application conditions have 36–191 graphs (77.3 on average) and 6 nesting levels, while optimized ones have 0–35 graphs (10.8 on average) and 0–6 nesting levels (2.6 on average). Hence, condition sizes are considerably reduced (by a factor of 7 on average).

6.5.2 Evaluating Performance

To answer RQ 2.1 and RQ 2.2, we set up two test scenarios comparing the runtime of *a-posteriori* and *a-priori* consistency checks.

Experiment set-up. Each test scenario (TS) consists of 15 test cases, one case for 15 selected rules (out of 84). These 15 rules are representative w.r.t. supported editing actions and rule size, in particular, they cover all kinds of editing actions. Their sizes range between 3 and 7 model elements. The average size of an application condition of the 15 rules is 56.4 graphs with nesting level 6 (without optimization) and 16.8 graphs with nesting level 3.1 (with optimization). Table 6.2 presents their detailed information. A test case of TS 1 consists of first applying an original rule to

Table 6.2: Numbers of graphs of application conditions and deepest nesting levels before and after optimization for the selected 15 rules

	w/o optimization		**w optimization**		
Rule	#graphs	level	#graphs	level	#overlapped const.
addToConnectionPoint..	58	6	30	6	(1)
addToStateMachine..	77	6	25	6	(2)
createBehavior..	70	6	1	1	(3)
createFinalState..	44	6	31	6	(11)
createRegion..	41	6	7	2	(2)
deleteFinalState..	42	6	29	6	(11)
deleteState..	42	6	29	6	(11)
deleteTrigger..	37	6	0	0	(0)
moveConstraint..	39	6	0	0	(0)
moveFinalState..	86	6	0	0	(0)
moveTransition..	49	6	0	0	(0)
removeEntry..	55	6	27	6	(1)
removeExit..	55	6	27	6	(1)
setState_submachine..	77	6	25	6	(2)
unsetState_submachine..	74	6	22	6	(2)
Average (15 rules)	56.4	6	16.8	3.8	3.1

a test model at a random match and then checking the validity of the resulting model (using (a) the EMF validator [34] configured to employ the OCLinEcore validator [97] to validate OCL constraints and (b) the OCL interpreter [35]). A test case of TS 2 consists of applying an updated rule (with (a) the guaranteeing and (b) the optimized application condition) to a test model at a random match. To eliminate effects stemming from the choice of match, each test case of a test scenario is performed 100 times. A test scenario in TS 1 (a) is performed in one run time session such that caching of information can be used advantageously. The second variant of TS 1 (a) performs each *a-posteriori* check in a separate session making caching useless. All the test scenarios have been performed on all the 7 valid test models. A timeout (TO) takes place if the average run time exceeds 5 minutes. To evaluate an OCL constraint using the OCL interpreter, the context object has to be given. Focusing on approach differences, the following times were excluded from the evaluation time: The time needed to find the context objects of all OCL constraints for the OCL interpreter, the loading time of a test model to any validator, and the time needed to roll back to the state of a test model after applying a rule whose resulting model does not satisfy the constraints.

Experiment results. Table 6.3 shows the following results: *A-posteriori* checking is performed in 3 variants. TS 1 (a) uses the EMF validator with and without

Table 6.3: Average run time (in seconds) of a single rule application (and validation) over 15 test cases with 100 random matches each using models of varying size

		Model size						
Scenario	**(Caching)**	**800**	**1 500**	**3 000**	**6 000**	**10 000**	**13 000**	**16 000**
TS 1(a)	(yes)	0.01	0.01	0.01	0.02	0.04	0.05	0.06
TS 1(a)	(no)	1.66	1.71	1.76	1.79	1.8	1.83	1.85
TS 1(b)	(no)	128.97	185.08	254.17	TO	TO	TO	TO
TS 2(a)	(no)	0.01	0.01	0.04	0.13	0.3	0.5	0.79
TS 2(b)	(no)	0.01	0.01	0.02	0.05	0.12	0.22	0.33

caching mechanism since we noted the followings: In the first validation check, the EMF validator took 1.77 to 1.95 seconds to check a test model of size between 800 to 16 000, whereas in the next validation checks, it took only 5 to 63 milliseconds. Our understanding of this improvement is that the EMF validator saves the model state after the first validity check. Thus, in the next checks at the same run time session, the EMF validator is still able to reach the model in the cache such that only the elements affected by rule application are considered. Without caching, the average run times are less than 2 seconds; with caching, they are even about two magnitudes faster. Using the OCL interpreter (TS 1 (b)) instead leads to run times over 2 minutes or even timeouts (after 5 min.). *A-priori* checking is performed in two variants: In TS 2 (a) rules with non-optimized application conditions are used while the application conditions in TS 2 (b) are optimized. The run times of both variants are below 1 second and hence slightly better than in TS 1 (a) without caching. Using caching, however, TS 1 (a) is even faster. This consideration yields the answer to RQ 2.1. To answer RQ 2.2 we can see that using rules with optimized application conditions is two and a half times faster than without optimization. Almost all of the times our rules were applicable and thus the whole application condition of a rule was completely checked and evaluated.

To conclude, we can state that scenarios TS 1 (a) and TS 2 are both fast enough to be usable in practice. However, a rollback step in the *a-posteriori* approach (TS 1) may not always be feasible. For example, if the rollback step is defined by applying the inverse rule, this is might not always be applicable if the rule computes attribute values. Furthermore, in the *a-posteriori* approach, the rule action is performed first, which may cause dangerous situations in several fields such as a railway system, self-driving cars, and an e-health system.

6.5.3 Threats to Validity

External validity can be questioned since we consider a limited number of OCL constraints and rules. For our performance experiments, we selected 15 out of 84 editing rules which are representative concerning their kinds (rules for creating, deleting, setting, unsetting, and moving model elements) and sizes. Moreover, we reduced the effect of the rules' matches by executing each rule at 100 matches chosen randomly from each given model. For performance evaluation, we restricted our studies to synthetic models. As we did not spot any performance bottleneck, we are convinced that using realistic models would not yield basically different results.

Concerning the considered OCL constraints it can be noticed that about half of them are simple negative constraints. However, all core features of OCL (logical operators, navigation expressions, and collection operators) are covered and at least one rather complex constraint is included. And, more importantly, this kind of a set of constraints seems to be quite typical for the chosen application case. Constraints required by model editors are often negative to forbid input that is not allowed anyway. Therefore, we are quite confident that the results are representative. Nevertheless, further case examples are interesting to be considered in the future, especially if they include very large models.

6.6 Related Work

Related works can be distinguished into two groups: (1) other works ensuring transformation rules to be consistency-preserving and (2) simplifying (application) conditions and constraints.

Ensuring transformation rules to be consistency-preserving. In [7, 106], Azab, Pennemann et al. introduce ENFORCe, a prototype implementation that can check and ensure the correctness of graph programs. It integrates graph constraints as left application conditions of rules as well but supports (partially) labeled graphs, not EMF models, and there is no translation from OCL to graph constraints available.

Clarisó et al. present in [28] how to calculate an application condition for a transformation rule and an OCL constraint, directly in OCL. The supported subset of OCL is slightly larger than in *OCL2AC* because, staying with OCL, they can support operations that are not first-order. The authors provide a correctness proof for the presented translation into application conditions. In addition, there is a partial implementation. Resulting application conditions are not essentially further optimized, neither by ENFORCe nor in work by Clarisó et al.

Dyck et al. [32, 33] present works that extend the one presented by Becker et al. [9]. Their goal is to verify whether a graph transformation system preserves graph consistency concerning forbidden graph patterns. They can improve the performance of the check under very specific conditions, e.g., the NACs have no nesting, have a common domain and the forbidden graph constraint is of a particular form. One improvement is done by reducing the number of the patterns that have to be checked. They check the ones which relate to the local match of the rule. This is similar to Theorem 1,3 of our work where we remove irrelevant subconditions and keep the ones which relate to the rule match. However, the objective of their approach is not to automatically construct constraint-preserving rules but rather to verify the consistency of a set of transformation rules w.r.t. a set of forbidden graph patterns. If that fails to be the case, the rule system has to be adapted manually by the user. Additionally, EMF constraints are not considered.

To the best of our knowledge, our work is the only one which constructs and optimizes the resulting application conditions considerably.

Simplifying (application) conditions and constraints. Rules for semantic equivalences in graph constraints and conditions have been reported in several places [105, 106, 114] and their application can lead to considerable simplification in the structure of a constraint. There are also approaches and implementations simplifying OCL constraints, especially automatically generated ones [49, 29]. Depending on the usage scenario, such simplifications could provide a valuable pre-processing step to our approach.

6.7 Discussion

For a given constraint-preserving application condition created w.r.t. the construction provided by Habel and Pennemann in [55], for example, the application condition can be refactored to be of form CNF [106]. Then, rules for semantic equivalences can be applied to reduce the structure of a constraint. However, here there are several open questions: Which semantic equivalence rules have to be applied and in which order? Do we always get the simplified constraint-preserving application condition? How do such computations take time w.r.t the nesting levels?

Anyway, assuming semantic equivalence rules are defined and applied optimally (in an optimal order), we can argue that our approach leads to get the optimal ones: According to Theorem 1,1 and Theorem 1,2, we get *true* as an application condition. The application condition *true* cannot be semantically simplified further. According to Theorem 1,3, we keep only the relevant overlapped graphs w.r.t. the rule match and each overlapped graph is unique and needed. Concerning Theorem 2, which deals with the violations of the EMF constraints, it is independent of the semantic of the resulting application conditions. It can even be applied as a last step after using the semantic equivalence rules in general.

Moreover, using our approach, we may often get an application condition that is *structurally* simpler than the one resulted by optimally applying semantic equivalence rules. For example, using our approach, the application condition *true* is represented by not integrating the corresponding constraint at all. Whereas, the application condition *true* which is resulted by optimally applying the semantic equivalence rules may consist of several checks which are semantically equivalent to *true* but composing several graph patterns.

Based on the previous argumentation, we have formally shown in [91] that our techniques construct the weakest preserving application conditions.

6.8 Conclusion

Application scenarios where each graph transformation step has to preserve the consistency of models w.r.t. given constraints are needed in practice. As the construction of application conditions in [55] yields consistency-guaranteeing ones and assuming that the preservation of graph consistency is already sufficient, the resulting application conditions can be considerably optimized. We developed several techniques (in Theorem 1 and Theorem 2) to construct optimized consistency-preserving application conditions and implemented them as an Eclipse-based tool, called *Optimizer*, on top of *OCL2AC*. The evaluation results show that *OCL2AC* can lead to quite large application conditions which can be significantly optimized by a factor of 7 (on average) using our developed techniques. Accordingly, while the performance results of correct graph transformations are good in general, applying rules with optimized application conditions is shown to be ca. 2.5 times faster than applying non-optimized ones. Moreover, we showed that the runtime comparison of *a-priori* consistency check using our techniques with *a-posteriori* consistency check using the EMF validator is fast.

In the future, we intend to further optimize resulting application conditions by identifying redundant sub-conditions and by checking negative invariants of modeling languages. Our ultimate goal is to obtain understandable application conditions identifying exactly those portions of the given constraints that are relevant for a given rule. This work is already an essential step in that direction. Moreover, our optimization of conditions could have some interesting applications beyond MDE. We are interested, e.g., in assessing if our ideas can be beneficially integrated into proof systems [106, 120].

7 Summary and Outlook

7.1 Summary

In Model-Driven Engineering (MDE), model consistency is vital during software development since it affects the quality of the generated artifacts, the successful application of the transformations, and the correctness of their results. Thus, it affects the quality of the overall system. In this thesis, we developed several consistency-by-construction techniques for ensuring the consistency of models and model transformations. Our techniques are implemented as ready-to-use tools performing the tasks in a fully automated and interactive way. Moreover, we evaluated the techniques and showed their efficiency and soundness.

We presented *a model repair technique* by developing a generic rule-based algorithm for trimming and completing models and thereby resolving their cardinality violations. The developed technique allows models to be viewed and changed interactively during the repair process and thus aims at designing desired models. The modeler has the opportunity to choose the repair actions and the matches from the proper suggested ones, to stop the repair process, or to let the model be repaired automatically at any time. The technique considers EMF constraints, and thus, the resulting models have a standard format in the sense that they can be manipulated by other standard tools (conforming to the OMG specifications). Different kinds of repair rules are designed in a way that each successful application of a rule enhances (or at least preserves) the model consistency. We formally proved under which conditions the developed repair algorithm terminates. Its features are discussed in more general settings, such as supporting a set of OCL constraints. Since the approach is generic, i.e., it can work with any domain meta-model and its instance models, it has to be customized for each given DSML defined by a meta-model. Therefore, meta-techniques are specified to customize the approach to the given meta-model. A meta-tool, called *Meta2RR*, is developed as an Eclipse plug-in to derive the repair techniques for a given DSML automatically. A DSML tool, called *EMF Model Repair*, is developed as an Eclipse plug-in and uses the generated repair technique to repair instance models of the given domain. The tool is systematically tested. Furthermore, we showed that the tool repairs models of size 10 000 elements with about 1511 violations in just 47 seconds on average.

We provided *a model generation technique* to efficiently generate large, consistent, and diverse models. The generation process can start at a seed model being potentially inconsistent as well. Such models are needed in various application scenarios like model transformation testing, benchmarking model queries and transformations, model-driven search and optimization, or validating the suitability of MDE tools to deal with large input models. Our rule-based approach is configurable and utilizes the model repair algorithm that guarantees to yield consistent EMF models (conforming to their meta-model without OCL constraints). Several parameterization strategies are presented to support user specifications and thus aiming at generating diverse instance models. Our approach is generic that can work with any domain meta-model and its instances, and therefore, meta-techniques are specified to customize it with respect to a given DSML. A meta-tool, called *Meta2GR*, is developed to derive the model generation techniques for the given DSML automatically. A DSML tool, called *EMF Model Generator*, is developed as an Eclipse plug-in and uses the derived generation technique to generate consistent models by supporting user specifications. Using several meta-models of different domains, we showed that the tool generates consistent models of size 10 000 elements in 85 seconds on average. Moreover, we showed that the tool was able to generate consistent EMF models with half a million elements in few minutes on average. Besides, we presented that our developed tool shows a speed-up of several orders of magnitude compared to the state-of-the-art generation tool. Furthermore, using the developed parameterization strategies, we presented that one could generate several sets of consistent models where the distribution of elements among types is significantly different.

In several application scenarios like in safety-critical systems and model refactoring, models have to ensure particular consistency properties, and the resulting models should belong to the transformation's target language. Moreover, manually enhancing a set of rules to guarantee a set of constraints is a tedious, time-consuming, and error-prone task. Besides, it requires high skills related to the theoretical foundation and the environment. For these purposes, we developed an automated technique for adapting a given rule-based model transformation such that resulting models guarantee a given set of constraints. Based on existing theory, *two correct-by-construction techniques are developed: First, OCL constraints are translated into semantically equivalent graph constraints. Secondly, graph constraints can further be integrated as guaranteeing application conditions into transformation rules.* The resulting rule is applicable to a model only if its application does not violate the original constraints. To the best of our knowledge, we provided the first ready-to-use tool, called *OCL2AC*, implementing and automating both techniques. We assessed their performances, and they are efficient: A set of 104 OCL constraints of the UML meta-model are translated in 2.4 seconds on average. Furthermore, the tool took only 2.3 seconds to update all 84 editing rules of the Magicdraw Statechart w.r.t. 11 constraints. Furthermore, we considered the complexity of the outputs. Although the translation of OCL constraints to graph constraints slightly increases the complexity, the resulting graph constraint is accompanied by a gain of information that might help in understanding the constraint. Additionally, the nesting of the constraints was never increased but decreased. Application conditions resulting from the integration process tend to be more complicated than the original graph constraints. However, the complexity of the resulting output is far better than could be expected from theory. In most cases, the growth is moderate (the number of graphs more or less doubles on average). Nonetheless, there are cases where the size of the results becomes too large. Therefore, we tackled this problem in the next contribution.

We developed *optimizing-by-construction techniques for application conditions for transformations that need to be consistency-preserving.* For a given transformation rule and a set of constraints, our techniques render the model transformation rule consistency-preserving. I.e., the rule is applicable to a consistent model if and only if the resulting model is still consistent after the rule application. Our techniques are conceptually new and quite general in scope. Moreover, two theorems are formally presented showing the correctness of the techniques.

We have implemented the technique as an Eclipse plug-in, called *optimizer*, on top of the tool *OCL2AC*. After that, we carried out an evaluation comparing non-optimized application conditions with the optimized ones. The results show a considerable loss in the complexity of application conditions. Given graph constraints with 2–10 graphs and 2–6 nesting levels, non-optimized application conditions have 36–191 graphs and 6 nesting levels, while optimized ones have 0–35 graphs and 0–6 nesting levels. Hence, condition sizes are considerably reduced by a factor of 7 on average. Moreover, the optimizations do not only reduce the size of computed application conditions considerably but also improve the performance of consistency-preserving transformations. While the performance results of graph transformations with application conditions are good in general, applying rules with optimized application conditions is shown to be ca. 2.5 times faster than applying non-optimized ones. Furthermore, we evaluated the run-time of the overall approach in general by comparing *a-posteriori* consistency check with *a-priori* consistency check based on our techniques. Results show that our techniques are fast enough to be usable in practice.

7.2 Outlook

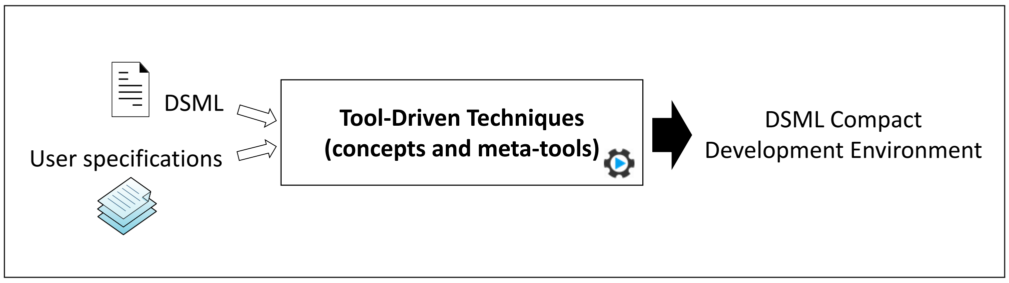


Figure 7.1: Vision: DSML Tool-Driven Engineering (Describe, Generate and Use)

Since tool development is one of the main obstacles for using DSMLs, our ultimate vision is to develop *tool-driven techniques* (new concepts, methods, and meta-tools) to ease and accelerate the DSML tool development. For a given DSML and user specifications (configurations), the tool-driven techniques should (semi-) automatically produce a compact development environment customized for the given DSML. The generated DSML development environment has to include at least the basic DSML tools needed during the development such as editing tools, repairing and quick fixing tools, quality assurance tools, testing tools, and versioning tools. Figure 7.1 illustrates the vision. This thesis represents a step forward towards this vision.

In the following, we give an outlook about the current and future work of our developed techniques of both parts.

7.2.1 Model Repair and Generation

Supporting a set of OCL constraints. Model repair and generation can be extended to support a set of OCL constraints by deriving proper rules from constraints. As we mentioned in Section 3.7.4, our approach can be straightforwardly extended to support a set of OCL constraints (of depth one) which are semantically equivalent to the lower and upper bound constraints such as the following expressions:

context T **inv**: self.cont/T.allInstances() → size()>=m/notEmpty()/isEmpty()/<n;
context T **inv**: self.cont→ select/any/collect(oclIsTypeOf(T1))→ notEmpty();
context T **inv**: self.name <>′ ′;
T and T1 denote to a node type and cont denotes to the outgoing reference.

Once such a set of OCL constraints is distinguished, the corresponding bounds of the meta-model can be refactored correspondingly. Furthermore, we have been working on deriving repair rules from more complex constraints to configure our repair algorithm. The idea is to translate OCL constraints to graph constraints and then to derive several repair rules from the graph structure of each constraint by considering the containment and non-containment paths (see Section 3.7.4 for more detail). We have already supervised a university project which derives repair rules configuring our repair algorithm from graph constraints. Since the structure of a graph constraint can be too complex and arbitrary, we intend to perform the following:

1. Translating commonly used OCL constraints in practice into graph constraints using our tool *OCL2AC* (namely, the second component *OCL2GC*).
2. Analyzing and classifying their structures instead of analyzing an arbitrary structure of a graph constraint.
3. Deriving the corresponding repair rules to configure the repair algorithm.

Besides, our project partner has been started investigating the idea by deriving graph programs for repairing graphs to satisfy a restricted form of graph constraints [56].

Additionally, we are interested in the following future work:

Editing operations for different purposes. Since we specified a catalog of different kinds of (repair and generation) rules with their meta-patterns and developed meta-tools that derive them from any given meta-model automatically, we are sure that they can be used to achieve various purposes such as model editing or for providing a benchmark of rules. Additionally, the rules are specified to respect EMF constraints and multiplicities and are flexible in the sense that they are a kind of atomic rules of different actions. They are designed in a way that each successful application of a rule enhances (or at least preserves) the consistency of the model. The various kinds of rules can be easily classified and used within control flows, such as transformation units. Lately, the catalog of our rules has been used for generating search operators in the field of search-based model engineering as presented in [22, 23].

Model synchronization. Since our rule-based approach and the developed tools record semantical model changes as an ordered list of rules' applications with their matches. This information may be beneficial, e.g., to resolve inconsistencies in distributed models or to synchronize distributed copies of a model. Instead of sending the whole repaired model, this list could be used as a patch to complement changes in other distributed copies of the model.

Realistic models. Based on heuristic information, we are interested in reordering the suggested proper repair actions during the repair process according to their occurrence in practice. Additionally, we want to generate realistic models by leveraging a stochastic controller.

7.2.2 Graph Constraints and Application Conditions

Foundation for handling complex attribute constraints. The theory beyond our tool *OCL2AC* does not consider in detail how to handle complex attribute constraints, including arithmetic operations. However, in the implementation, we advanced that as follows:

The meta-model of a graph constraint is extended to have a node type called *AttributeCondition* for specifying arithmetic operations. Each condition may have an arbitrary number of attribute conditions. The node type *Attribute* (part of the graph meta-model) is specified to have a filed *value* that can hold a primitive value or a variable. At this end, the arithmetic operations can be specified using the variables assigned to the fields. During the integration of a graph constraint as an application condition into a rule, the tool *OCL2AC* copies and introduces the attribute conditions of a graph constraint as attribute conditions of the rule. Moreover, the attribute variables of the graph constraint are introduced as rule variables. Note that the integrated attribute conditions may conflict with the already existing attribute conditions of the rule because, e.g., the arithmetic operations become unsolvable, or they use the same variable names with different semantics. However, the foundation and the correctness are not yet comprehensively studied.

Constraints as application conditions. As outlined in Section 5.3.4, integrating a constraint into an *empty* rule produces an application condition being semantically equivalent to the constraint. Thus, the rule with the resulting application condition is applicable if and only if the input model is consistent w.r.t. the constraint. The tool *OCL2AC* implements that also. This idea is promising to support and develop the following projects:

- *Deriving a configurable model checker from graph constraints*: Given a set of constraints, we apply *OCL2AC* to integrate each constraint into an empty rule. Thus, we get a check rule which is applicable if and only if the model satisfies the constraint. Then, we set the resulting enhanced rules in a transformation unit such as a sequential unit. If all the sub-units of the sequential unit are applicable, the model is consistent w.r.t. the configured constraints. Otherwise, the information about the violations is reported to the user. The sequential unit can be configured by the user who can choose which constraints have to be checked, i.e., which rules have to configure the unit. Moreover, the model checker could also be realized by deriving check rules from the structures of graph constraints directly. Here, the graph structure of the constraint constitutes the graphs of the check rule. For example, given a constraint of the form $\forall(P, \exists C)$, where P is a sub-graph of C, the check rule can be specified to preserve the graph P and forbid the graph $C \backslash P$. Thus, the check rule is designed to be applicable if and only if the model does not satisfy the constraint. If the check rule is not applicable, the model satisfies the constraint. The advantage of this specification is that using the rule match we can retrieve information about the violations such as the location. Our configurable model checker can also be used as a test model assertion.
- *Auto-detection for smells, refactoring, or quick fixing*: Given a set of smells specified as graph constraints or OCL constraints, one can use our tool *OCL2AC* to derive an application condition from a smell specification by integrating the constraint into an empty rule. This resulting application condition checks the existence of such a smell in a model. Appending the resulting application condition into the corresponding fixing or refactoring rule would foster the trust of automatically applying the refactoring/fixing rule to eliminate the smells while the model is manipulated. Additionally, the smell specification and its refactoring rule are composed together as one unit, i.e., as a rule with an application condition, and that might be more efficient if the rules are distributed.

Application scenarios for self-adaptive and safety-critical systems. Since our techniques perform the tasks automatically and they perform efficiently, we are looking for scenarios from, e.g., self-adaptive systems where the requirements (the constraints) are changed continuously, and the transformations have to be updated accordingly. Additionally, we are looking forward to applying our techniques to advance the safety-critical application scenarios in which the transformations have to fulfill particular properties. Systems like the Wireless Sensor Network (WSN) and the Internet of Things (IoT) are interesting application scenarios for applying our techniques since preserving the connectivity, for example, among devices is required [68].

Bibliography

[1] Acretoaie, V., Störrle, H.: Hypersonic: Model analysis and checking in the cloud. In: Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014. pp. 6–13 (2014), `http://ceur-ws.org/Vol-1206/paper_5.pdf`

[2] Apt, K.R., Wallace, M.: Constraint logic programming using Eclipse. Cambridge University Press (2007)

[3] Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I. pp. 121–135 (2010). https://doi.org/10.1007/978-3-642-16145-2_9, `https://doi.org/10.1007/978-3-642-16145-2_9`

[4] Arendt, T., Habel, A., Radke, H., Taentzer, G.: From core OCL invariants to nested graph constraints. In: Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings. pp. 97–112 (2014). https://doi.org/10.1007/978-3-319-09108-2_7, `https://doi.org/10.1007/978-3-319-09108-2_7`

[5] Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the eclipse modeling framework. Autom. Softw. Eng. **20**(2), 141–184 (2013). https://doi.org/10.1007/s10515-012-0114-7, `https://doi.org/10.1007/s10515-012-0114-7`

[6] Atlantic Zoo: Ecore, `http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore`, (visited 2019)

[7] Azab, K., Habel, A., Pennemann, K., Zuckschwerdt, C.: Enforce: A system for ensuring formal correctness of high-level programs. ECEASST

1 (2006). https://doi.org/10.14279/tuj.eceasst.1.82, https://doi.org/10.14279/tuj.eceasst.1.82

[8] Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA workshop on Integration of Model Driven Development and Model Driven Testing. (2006)

[9] Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006. pp. 72–81 (2006). https://doi.org/10.1145/1134285.1134297, https://doi.org/10.1145/1134285.1134297

[10] Becker, B., Lambers, L., Dyck, J., Birth, S., Giese, H.: Iterative development of consistency-preserving rule-based refactorings. In: Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings. pp. 123–137 (2011). https://doi.org/10.1007/978-3-642-21732-6_9, https://doi.org/10.1007/978-3-642-21732-6_9

[11] Benelallam, A., Tisi, M., Ráth, I., Izsó, B., Kolovos, D.S.: Towards an open set of real-world benchmarks for model queries and transformations. In: Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering colocated with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014. pp. 14–22 (2014), http://ceur-ws.org/Vol-1206/paper_7.pdf

[12] Bergmann, G.: Translating OCL to graph patterns. In: Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings. pp. 670–686 (2014). https://doi.org/10.1007/978-3-319-11653-2_41, https://doi.org/10.1007/978-3-319-11653-2_41

[13] Bergmann, G.: Translating OCL to graph patterns. In: Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings. pp. 670–686 (2014). https://doi.org/10.1007/978-3-319-11653-2_41, https://doi.org/10.1007/978-3-319-11653-2_41

[14] Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Software and System Modeling **11**(2), 227–250 (2012). https://doi.org/10.1007/s10270-011-0199-7, https://doi.org/10.1007/s10270-011-0199-7

[15] Bill, R., Fleck, M., Troya, J., Mayerhofer, T., Wimmer, M.: A local and global tour on momot. Software and System Modeling **18**(2), 1017–1046 (2019). https://doi.org/10.1007/s10270-017-0644-3, https://doi.org/10.1007/s10270-017-0644-3

[16] Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Morgan & Claypool

Publishers (2012). https://doi.org/10.2200/S00441ED1V01Y201208SWE001, https://doi.org/10.2200/S00441ED1V01Y201208SWE001

[17] Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML Progress Report: Structural Layer Proposal. In: Graph Drawing. pp. 501–512. Springer, Berlin and Heidelberg (2002). https://doi.org/10.1007/3-540-45848-4_59

[18] Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: 17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA. pp. 85–94 (2006). https://doi.org/10.1109/ISSRE.2006.27, https://doi.org/10.1109/ISSRE.2006.27

[19] Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., Ratiu, D.: Seamless model-based development: From isolated tools to integrated model engineering environments. Proceedings of the IEEE **98**(4), 526–545 (2010). https://doi.org/10.1109/JPROC.2009.2037771, https://doi.org/10.1109/JPROC.2009.2037771

[20] Brucker, A.D., Wolff, B.: HOL-OCL: A formal proof environment for UML/OCL. In: Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 97–100 (2008). https://doi.org/10.1007/978-3-540-78743-3_8, https://doi.org/10.1007/978-3-540-78743-3_8

[21] Bruckhaus, T.F.W., Madhavji, N.H., Janssen, I., Henshaw, J.: The impact of tools on software productivity. IEEE Software **13**(5), 29–38 (1996). https://doi.org/10.1109/52.536456, https://doi.org/10.1109/52.536456

[22] Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic consistency preserving search operators for search-based model engineering. CoRR **abs/1907.05647** (2019), http://arxiv.org/abs/1907.05647

[23] Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic consistency preserving search operators for search-based model engineering. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019. pp. 106–116. IEEE (2019). https://doi.org/10.1109/MODELS.2019.00-10, https://doi.org/10.1109/MODELS.2019.00-10

[24] Bézivin, J., Bruneliere, H., Jouault, F., Kurtev, I.: Model engineering support for tool interoperability. In: In Proceedings of 4th Workshop in Software Model Engineering at 8th Int. Conf. on Model Driven Engineering Languages and Systems (10 2005)

[25] Cabot, J., Clarisó, R.: Uml-ocl verification in practice. In: 1st Int. Workshop Challenges in Model Driven Software Engineering. vol. 5421, pp. 31–35 (2008), http://ssel.vub.ac.be/ChaMDE08/

[26] Cabot, J., Clarisó, R., Riera, D.: Umltocsp: a tool for the formal verification of UML/OCL models using constraint programming. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. pp. 547–548 (2007). https://doi.org/10.1145/1321631.1321737, https://doi.org/10.1145/1321631.1321737

[27] Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software **93**, 1–23 (2014). https://doi.org/10.1016/j.jss.2014.03.023, https://doi.org/10.1016/j.jss.2014.03.023

[28] Clarisó, R., Cabot, J., Guerra, E., de Lara, J.: Backwards reasoning for model transformations: Method and applications. Journal of Systems and Software **116**, 113–132 (2016). https://doi.org/10.1016/j.jss.2015.08.017, https://doi.org/10.1016/j.jss.2015.08.017

[29] Cuadrado, J.S.: Optimising OCL synthesized code. In: Modelling Foundations and Applications - 14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Toulouse, France, June 26-28, 2018, Proceedings. pp. 28–45 (2018). https://doi.org/10.1007/978-3-319-92997-2_3, https://doi.org/10.1007/978-3-319-92997-2_3

[30] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3), 621–646 (2006). https://doi.org/10.1147/sj.453.0621, https://doi.org/10.1147/sj.453.0621

[31] Daniel, G., Sunyé, G., Benelallam, A., Tisi, M.: Improving memory efficiency for processing large-scale models. In: Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014. pp. 31–39 (2014), http://ceur-ws.org/Vol-1206/paper_6.pdf

[32] Dyck, J., Giese, H.: Inductive invariant checking with partial negative application conditions. In: Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings. pp. 237–253 (2015). https://doi.org/10.1007/978-3-319-21145-9_15, https://doi.org/10.1007/978-3-319-21145-9_15

[33] Dyck, J., Giese, H.: k-inductive invariant checking for graph transformation systems. In: Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings. pp. 142–158 (2017). https://doi.org/10.1007/978-3-319-61470-0_9, https://doi.org/10.1007/978-3-319-61470-0_9

[34] Eclipse Foundation: Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/, (visited 2019)

[35] Eclipse Foundation: Eclipse OCL (Object Constraint Language), https://projects.eclipse.org/projects/modeling.mdt.ocl, (visited 2019)

[36] Egyed, A.: Instant consistency checking for the UML. In: 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006. pp. 381–390 (2006). https://doi.org/10.1145/1134285.1134339, https://doi.org/10.1145/1134285.1134339

[37] Egyed, A.: Fixing inconsistencies in UML design models. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007. pp. 292–301 (2007). https://doi.org/10.1109/ICSE.2007.38, https://doi.org/10.1109/ICSE.2007.38

[38] Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in UML design models. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy. pp. 99–108 (2008). https://doi.org/10.1109/ASE.2008.20, https://doi.org/10.1109/ASE.2008.20

[39] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2006). https://doi.org/10.1007/3-540-31188-2, https://doi.org/10.1007/3-540-31188-2

[40] Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation - General Framework and Applications. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2015). https://doi.org/10.1007/978-3-662-47980-3, https://doi.org/10.1007/978-3-662-47980-3

[41] Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software and System Modeling **8**(4), 479–500 (2009). https://doi.org/10.1007/s10270-008-0095-y, https://doi.org/10.1007/s10270-008-0095-y

[42] Engels, G., Sauer, S.: A meta-method for defining software engineering methods. In: Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday. pp. 411–440 (2010). https://doi.org/10.1007/978-3-642-17322-6_18, https://doi.org/10.1007/978-3-642-17322-6_18

[43] Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: Proceedings. 2004 First International Workshop on Model, Design and Validation, 2004. pp. 29–40 (Nov 2004). https://doi.org/10.1109/MODEVA.2004.1425846

[44] Fleurey, F., Baudry, B., Muller, P., Traon, Y.L.: Qualifying input test data for model transformations. Software and System Modeling **8**(2), 185–203 (2009). https://doi.org/10.1007/s10270-007-0074-8, https://doi.org/10.1007/s10270-007-0074-8

[45] Fowler, M.: Domain-Specific Languages. The Addison-Wesley signature series, Addison-Wesley (2011), http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html

[46] France, R.B., Rumpe, B.: Model-driven development of complex software: A research roadmap. CoRR **abs/1409.6620** (2014), http://arxiv.org/abs/1409.6620

[47] Garmendia, A., Jiménez-Pastor, A., de Lara, J.: Scalable model exploration through abstraction and fragmentation strategies. In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 23, 2015. pp. 21–31 (2015), http://ceur-ws.org/Vol-1406/paper3.pdf

[48] Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards verified model transformations. In: IN PROCEEDINGS OF MODEVA WORKSHOP ASSOCIATED TO MODELS'06. pp. 78–93 (2006)

[49] Giese, M., Larsson, D.: Simplifying transformations of OCL constraints. In: Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings. pp. 309–323 (2005). https://doi.org/10.1007/11557432_23, https://doi.org/10.1007/11557432_23

[50] Gómez Abel, A.T.: EMF random instantiator (2015), https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator,https://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/, (visited 2019)

[51] Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and System Modeling **4**(4), 386–398 (2005). https://doi.org/10.1007/s10270-005-0089-y, https://doi.org/10.1007/s10270-005-0089-y

[52] González, C.A., Büttner, F., Clarisó, R., Cabot, J.: Emftocsp: a tool for the lightweight verification of EMF models. In: Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012. pp. 44–50 (2012). https://doi.org/10.1109/FormSERA.2012.6229788, https://doi.org/10.1109/FormSERA.2012.6229788

[53] González, C.A., Cabot, J.: Formal verification of static software models in MDE: A systematic review. Information & Software Technology **56**(8), 821–838 (2014). https://doi.org/10.1016/j.infsof.2014.03.003, https://doi.org/10.1016/j.infsof.2014.03.003

[54] Graph Solver Tool. https://github.com/viatra/VIATRA-Generator/wiki, (visited 2019)

[55] Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19**(2), 245–296 (2009). https://doi.org/10.1017/S0960129508007202, https://doi.org/10.1017/S0960129508007202

[56] Habel, A., Sandmann, C.: Graph repair by graph programs. In: Software Technologies: Applications and Foundations. pp. 431–446. Springer International Publishing, Cham (2018)

[57] He, X., Zhang, T., Ma, Z., Shao, W.: Randomized model generation for performance testing of model transformations. In: IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014. pp. 11–20 (2014). https://doi.org/10.1109/COMPSAC.2014.103, `https://doi.org/10.1109/COMPSAC.2014.103`

[58] Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars - a constructive approach -. Electronic Notes in Theoretical Computer Science **2**, 118 – 126 (1995). https://doi.org/https://doi.org/10.1016/S1571-0661(05)80188-4, `http://www.sciencedirect.com/science/article/pii/S1571066105801884`, sEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation

[59] Hegedüs, Á., Horváth, Á., Ráth, I., Branco, M.C., Varró, D.: Quick fix generation for dsmls. In: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011. pp. 17–24 (2011). https://doi.org/10.1109/VLHCC.2011.6070373, `https://doi.org/10.1109/VLHCC.2011.6070373`

[60] Heitmeyer, C.: Developing safety-critical systems: The role of formal methods and tools. In: Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55. pp. 95–99. SCS '05, Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2006), `http://dl.acm.org/citation.cfm?id=1151816.1151826`

[61] Henshin: Henshin project, `https://www.eclipse.org/henshin/`, (visited 2019)

[62] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259, `http://doi.acm.org/10.1145/363235.363259`

[63] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002). https://doi.org/10.1145/505145.505149, `https://doi.org/10.1145/505145.505149`

[64] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1-2), 31–39 (2008). https://doi.org/10.1016/j.scico.2007.08.002, `https://doi.org/10.1016/j.scico.2007.08.002`

[65] Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. pp. 191–201 (2013). https://doi.org/10.1109/ASE.2013.6693079, `https://doi.org/10.1109/ASE.2013.6693079`

[66] Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings. pp. 173–188 (2016). https://doi.org/10.1007/978-3-319-42064-6_12, `https://doi.org/10.1007/978-3-319-42064-6_12`

[67] Kleppe, A., Warmer, J., Bast, W.: MDA explained - the Model Driven Architecture: practice and promise. Addison Wesley object technology series, Addison-Wesley (2003), `http://www.informit.com/store/mda-explained-the-model-driven-architecture-practice-9780321194428`

[68] Kluge, R., Stein, M., Varró, G., Schürr, A., Hollick, M., Mühlhäuser, M.: A systematic approach to constructing families of incremental topology control algorithms using graph transformation. Software and System Modeling **18**(1), 279–319 (2019). https://doi.org/10.1007/s10270-017-0587-8, `https://doi.org/10.1007/s10270-017-0587-8`

[69] Kolovos, D.S., Paige, R.F., Polack, F.: The grand challenge of scalability for model driven engineering. In: Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers. pp. 48–53 (2008). https://doi.org/10.1007/978-3-642-01648-6_5, `https://doi.org/10.1007/978-3-642-01648-6_5`

[70] Kolovos, D.S., Rose, L.M., Matragkas, N.D., Paige, R.F., Guerra, E., Cuadrado, J.S., de Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013. p. 2 (2013). https://doi.org/10.1145/2487766.2487768, `https://doi.org/10.1145/2487766.2487768`

[71] Kosiol, J., Fritsche, L., Nassar, N., Schürr, A., Taentzer, G.: Constructing constraint-preserving interaction schemes in adhesive categories. In: Recent Trends in Algebraic Development Techniques - 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 2-5, 2018, Revised Selected Papers. pp. 139–153 (2018). https://doi.org/10.1007/978-3-030-23220-7_8, `https://doi.org/10.1007/978-3-030-23220-7_8`

[72] Krause, C., Giese, H.: Probabilistic graph transformation systems. In: Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings. pp. 311–325 (2012). https://doi.org/10.1007/978-3-642-33654-6_21, `https://doi.org/10.1007/978-3-642-33654-6_21`

[73] Lack, S., Soboci'nski, P.: Adhesive and quasiadhesive categories. ITA **39**(3), 511–545 (2005). https://doi.org/10.1051/ita:2005028, `https://doi.org/10.1051/ita:2005028`

[74] Lamari, M.: Towards an automated test generation for the verification of model transformations. In: Proceedings of the 2007 ACM Symposium

on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007. pp. 998–1005 (2007). https://doi.org/10.1145/1244002.1244220, https://doi.org/10.1145/1244002.1244220

[75] Laurent, Y., Bendraou, R., Gervais, M.: Generation of process using multi-objective genetic algorithm. In: International Conference on Software and System Process, ICSSP '13, San Francisco, CA, USA, May 18-19, 2013. pp. 161–165 (2013). https://doi.org/10.1145/2486046.2486076, https://doi.org/10.1145/2486046.2486076

[76] Leblebici, E., Anjorin, A., Schürr, A.: Developing emoflon with emoflon. In: Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings. pp. 138–145 (2014). https://doi.org/10.1007/978-3-319-08789-4_10, https://doi.org/10.1007/978-3-319-08789-4_10

[77] Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J.: Assessing the state-of-practice of model-based engineering in the embedded systems domain. In: Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings. pp. 166–182 (2014). https://doi.org/10.1007/978-3-319-11653-2_11, https://doi.org/10.1007/978-3-319-11653-2_11

[78] Macedo, N., Guimarães, T., Cunha, A.: Model repair and transformation with echo. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. pp. 694–697 (2013). https://doi.org/10.1109/ASE.2013.6693135, https://doi.org/10.1109/ASE.2013.6693135

[79] Macedo, N., Tiago, J., Cunha, A.: A feature-based classification of model repair approaches. IEEE Trans. Software Eng. **43**(7), 615–640 (2017). https://doi.org/10.1109/TSE.2016.2620145, https://doi.org/10.1109/TSE.2016.2620145

[80] MagicDraw: Magic draw, https://www.nomagic.com/products/magicdraw, (visited 2019)

[81] McGill, M.J., Stirewalt, R.E.K., Dillon, L.K.: Automated test input generation for software that consumes ORM models. In: On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Confederated International Workshops and Posters, ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009, Vilamoura, Portugal, November 1-6, 2009. Proceedings. pp. 704–713 (2009). https://doi.org/10.1007/978-3-642-05290-3_86, https://doi.org/10.1007/978-3-642-05290-3_86

[82] McQuillan, J.A., Power, J.F.: A metamodel for the measurement of object-oriented systems: An analysis using alloy. In: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008. pp. 288–297 (2008). https://doi.org/10.1109/ICST.2008.58, https://doi.org/10.1109/ICST.2008.58

[83] Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. **152**, 125–142 (2006). https://doi.org/10.1016/j.entcs.2005.10.021, `https://doi.org/10.1016/j.entcs.2005.10.021`

[84] Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and System Modeling **6**(3), 269–285 (2007). https://doi.org/10.1007/s10270-006-0044-6, `https://doi.org/10.1007/s10270-006-0044-6`

[85] Mohagheghi, P., Fernández, M.A., Martell, J.A., Fritzsche, M., Gilani, W.: MDE adoption in industry: Challenges and success criteria. In: Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers. pp. 54–59 (2008). https://doi.org/10.1007/978-3-642-01648-6_6, `https://doi.org/10.1007/978-3-642-01648-6_6`

[86] Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings. pp. 130–145 (2009). https://doi.org/10.1007/978-3-642-02674-4_10, `https://doi.org/10.1007/978-3-642-02674-4_10`

[87] Nassar, N., Arendt, T., Taentzer, G.: Deducing model metrics from meta models. In: Modellierung 2016, 2.-4. März 2016, Karlsruhe. pp. 29–44 (2016), `https://dl.gi.de/20.500.12116/828`

[88] Nassar, N., Kosio, J., Kehrer, T., Taentzer, G.: Generating Large EMF Models Efficiently: A Rule-Based, Configurable Approach. In: Fundamental Approaches to Software Engineering, 23th International Conference, FASE 2020, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland (2020), (accepted)

[89] Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: OCL2AC: automatic translation of OCL constraints to graph constraints and application conditions for transformation rules. In: Graph Transformation - 11th International Conference, ICGT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings. pp. 171–177 (2018). https://doi.org/10.1007/978-3-319-92991-0_11, `https://doi.org/10.1007/978-3-319-92991-0_11`

[90] Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: Constructing optimized validity-preserving application conditions for graph transformation rules. In: Graph Transformation - 12th International Conference, ICGT 2019, Held as Part of STAF 2019, Eindhoven, The Netherlands, July 15-16, 2019, Proceedings. pp. 177–194 (2019). https://doi.org/10.1007/978-3-030-23611-3_11, `https://doi.org/10.1007/978-3-030-23611-3_11`

[91] Nassar, N., Kosiol, J., Arendt, T., Taentzer, G.: Constructing weakest constraint-preserving application conditions for model transformation rules. In: Journal of Logical and Algebraic Methods in Programming (JLAMP) (2020), (submitted)

[92] Nassar, N., Kosiol, J., Radke, H.: Rule-based repair of emf models: Formalization and correctness proof. In: Graph Computation Models (GCM 2017), Electronic Pre-Proceedings (2017), pages.di.unipi.it/corradini/Workshops/GCM2017/papers/Nassar-Kosiol-Radke-GCM2017.pdf

[93] Nassar, N., Radke, H., Arendt, T.: Rule-based repair of EMF models: An automated interactive approach. In: Theory and Practice of Model Transformation - 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings. pp. 171–181 (2017). https://doi.org/10.1007/978-3-319-61473-1_12, https://doi.org/10.1007/978-3-319-61473-1_12

[94] Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM Trans. Internet Techn. **2**(2), 151–185 (2002). https://doi.org/10.1145/514183.514186, https://doi.org/10.1145/514183.514186

[95] Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA. pp. 455–464 (2003). https://doi.org/10.1109/ICSE.2003.1201223, https://doi.org/10.1109/ICSE.2003.1201223

[96] Newman, W.: A preliminary analysis of the products of HCI research, using *Pro Forma* abstracts. In: Conference on Human Factors in Computing Systems, CHI 1994, Boston, Massachusetts, USA, April 24-28, 1994, Conference Companion. p. 219 (1994). https://doi.org/10.1145/259963.260383, https://doi.org/10.1145/259963.260383

[97] OCLinEcore: Eclipse OCL, https://wiki.eclipse.org/OCL/OCLinEcore, (visited 2019)

[98] OCUP2 Examination Team: Meta-Modeling and the OMG Meta Object Facility (MOF) (03 2017), https://www.omg.org/ocup-2/documents/Meta-ModelingAndtheMOF.pdf, (visited 2019)

[99] OMG: Object Constraint Language. Version 2.4 (02 2014), http://www.omg.org/spec/OCL/, (visited 2019)

[100] OMG: OMG Unified Modeling Language. Version 2.5 (03 2015), http://www.omg.org/spec/UML/2.5/, (visited 2019)

[101] OMG: XML Metadata Interchange. Version 2.5.1 (2015), https://www.omg.org/spec/XMI/About-XMI/, (visited 2019)

[102] OMG: MOF Query/View/Transformation. Version 1.3 (06 2016), https://www.omg.org/spec/QVT, (visited 2019)

[103] OMG: OMG Meta Object Facility (MOF). Version 2.5.1 (11 2016), http://www.omg.org/spec/MOF/, (visited 2019)

[104] Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating fuctional tests. Commun. ACM **31**(6), 676–686 (Jun 1988).

https://doi.org/10.1145/62959.62964, http://doi.acm.org/10.1145/62959.62964

[105] Pennemann, K.H.: Generalized constraints and application conditions for graph transformation systems. Diplomarbeit, Department für Informatik, Universität Oldenburg (2004), http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/Penn04-Dipl.pdf

[106] Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Carl von Ossietzky-Universität Oldenburg (2009)

[107] Pietsch, P., Yazdi, H.S., Kelter, U.: Generating realistic test models for model processing tools. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011. pp. 620–623 (2011). https://doi.org/10.1109/ASE.2011.6100140, https://doi.org/10.1109/ASE.2011.6100140

[108] Pietsch, P., Yazdi, H.S., Kelter, U.: Controlled generation of models with defined properties. In: Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin. pp. 95–106 (2012), https://dl.gi.de/20.500.12116/18346

[109] Popoola, S., Kolovos, D.S., Rodriguez, H.H.: EMG: A domain-specific transformation language for synthetic model generation. In: Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings. pp. 36–51 (2016). https://doi.org/10.1007/978-3-319-42064-6_3, https://doi.org/10.1007/978-3-319-42064-6_3

[110] Puissant, J.P., Straeten, R.V.D., Mens, T.: Resolving model inconsistencies using automated regression planning. Software and System Modeling **14**(1), 461–481 (2015). https://doi.org/10.1007/s10270-013-0317-9, https://doi.org/10.1007/s10270-013-0317-9

[111] Rabbi, F., Lamo, Y., Yu, I.C., Kristensen, L.M.: A diagrammatic approach to model completion. In: Proceedings of the 4th Workshop on the Analysis of Model Transformations co-located with the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 28, 2015. pp. 56–65 (2015), http://ceur-ws.org/Vol-1500/paper7.pdf

[112] Radke, H.: A Theory of HR* Graph Conditions and their Application to Meta-Modeling. Ph.D. thesis, University of Oldenburg, Germany (2016), http://oops.uni-oldenburg.de/2803

[113] Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints focusing on set operations. In: Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings. pp. 155–170 (2015). https://doi.org/10.1007/978-3-319-21145-9_10, https://doi.org/10.1007/978-3-319-21145-9_10

[114] Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. Sci. Comput. Program. **152**, 38–62 (2018). https://doi.org/10.1016/j.scico.2017.08.006, https://doi.org/10.1016/j.scico.2017.08.006

[115] Rensink, A.: Representing first-order logic using graphs. In: Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings. pp. 319–335 (2004). https://doi.org/10.1007/978-3-540-30203-2_23, https://doi.org/10.1007/978-3-540-30203-2_23

[116] Richa, E., Borde, E., Pautet, L.: Translation of ATL to AGT and application to a code generator for simulink. Software and System Modeling **18**(1), 321–344 (2019). https://doi.org/10.1007/s10270-017-0607-8, https://doi.org/10.1007/s10270-017-0607-8

[117] Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A methodology for verifying refinements of partial models. Journal of Object Technology **14**(3), 3:1–31 (2015). https://doi.org/10.5381/jot.2015.14.3.a3, https://doi.org/10.5381/jot.2015.14.3.a3

[118] Scheidgen, M.: Generation of large random models for benchmarking. In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 23, 2015. pp. 1–10 (2015), http://ceur-ws.org/Vol-1406/paper1.pdf

[119] Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006). https://doi.org/10.1109/MC.2006.58, https://doi.org/10.1109/MC.2006.58

[120] Schneider, S., Lambers, L., Orejas, F.: Automated reasoning for attributed graph properties. International Journal on Software Tools for Technology Transfer **20**(6), 705–737 (2018). https://doi.org/10.1007/s10009-018-0496-3

[121] Schneider, S., Lambers, L., Orejas, F.: A logic-based incremental approach to graph repair. In: Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. pp. 151–167 (2019). https://doi.org/10.1007/978-3-030-16722-6_9, https://doi.org/10.1007/978-3-030-16722-6_9

[122] Sedlmeier, M.: Evaluation of model comparison for delta-compression in model persistence. In: Proceedings of the 4rd Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations (STAF 2016) federation of conferences, Vienna, Austria, July 8, 2016. pp. 11–20 (2016), http://ceur-ws.org/Vol-1652/paper2.pdf

[123] Seidewitz, E.: What models mean. IEEE Software **20**(5), 26–32 (2003). https://doi.org/10.1109/MS.2003.1231147, https://doi.org/10.1109/MS.2003.1231147

[124] Selic, B.: What will it take? A view on adoption of model-based methods in practice. Software and System Modeling **11**(4), 513–526 (2012). https://doi.org/10.1007/s10270-012-0261-0, `https://doi.org/10.1007/s10270-012-0261-0`

[125] Semeráth, O., Nagy, A.S., Varró, D.: A graph solver for the automated generation of consistent domain-specific models. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 969–980 (2018). https://doi.org/10.1145/3180155.3180186, `https://doi.org/10.1145/3180155.3180186`

[126] Semeráth, O., Varró, D.: Graph constraint evaluation over partial models by constraint rewriting. In: Theory and Practice of Model Transformation. Lecture Notes in Computer Science, vol. 10374, pp. 138–154. Springer (2017). https://doi.org/10.1007/978-3-319-61473-1_10

[127] Semeráth, O., Varró, D.: Iterative generation of diverse models for testing specifications of DSL tools. In: Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. pp. 227–245 (2018). https://doi.org/10.1007/978-3-319-89363-1_13, `https://doi.org/10.1007/978-3-319-89363-1_13`

[128] Sen, S., Baudry, B., Mottu, J.: Automatic model generation strategies for model transformation testing. In: Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings. pp. 148–164 (2009). https://doi.org/10.1007/978-3-642-02408-5_11, `https://doi.org/10.1007/978-3-642-02408-5_11`

[129] Sen, S., Baudry, B., Precup, D.: Partial Model Completion in Model Driven Engineering using Constraint Logic Programming. In: In Proc. INAP'07 (2007)

[130] Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software **20**(5), 42–45 (2003). https://doi.org/10.1109/MS.2003.1231150, `https://doi.org/10.1109/MS.2003.1231150`

[131] Shannon, C.E.: A mathematical theory of communication. Mobile Computing and Communications Review **5**(1), 3–55 (2001). https://doi.org/10.1145/584091.584093, `https://doi.org/10.1145/584091.584093`

[132] Shaw, M.: What makes good research in software engineering? STTT **4**(1), 1–7 (2002). https://doi.org/10.1007/s10009-002-0083-4, `https://doi.org/10.1007/s10009-002-0083-4`

[133] Siegel, J., the OMG Staff Strategy Group: Developing in OMG's Model Driven Architecture (MDA). OMG (2001), `https://www.omg.org/cgi-bin/doc?omg/2001-12-01`, (visited 2019)

[134] Steimann, F., Frenkel, M., Voelter, M.: Robust projectional editing. In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017. pp. 79–90 (2017). https://doi.org/10.1145/3136014.3136034, https://doi.org/10.1145/3136014.3136034

[135] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison Wesley, Upper Saddle River, NJ, 2 edn. (2008)

[136] Straeten, R.V.D., Mens, T., Baelen, S.V.: Challenges in model-driven software engineering. In: Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers. pp. 35–47 (2008). https://doi.org/10.1007/978-3-642-01648-6_4, https://doi.org/10.1007/978-3-642-01648-6_4

[137] Strüber, D., Kehrer, T., Arendt, T., Pietsch, C., Reuling, D.: Scalability of model transformations: Position paper and benchmark set. In: Proceedings of the 4rd Workshop on Scalable Model Driven Engineering. CEUR Workshop Proceedings, vol. 1652, pp. 21–30 (2016), http://ceur-ws.org/Vol-1652/paper3.pdf

[138] Svendsen, A., Haugen, Ø., Møller-Pedersen, B.: Synthesizing software models: Generating train station models automatically. In: SDL 2011: Integrating System and Software Modeling - 15th International SDL Forum Toulouse, France, July 5-7, 2011. Revised Papers. pp. 38–53 (2011). https://doi.org/10.1007/978-3-642-25264-8_5, https://doi.org/10.1007/978-3-642-25264-8_5

[139] Taentzer, G.: Instance generation from type graphs with arbitrary multiplicities. ECEASST **47** (2012). https://doi.org/10.14279/tuj.eceasst.47.727, https://doi.org/10.14279/tuj.eceasst.47.727

[140] Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. pp. 283–299 (2017). https://doi.org/10.1007/978-3-662-54494-5_16, https://doi.org/10.1007/978-3-662-54494-5_16

[141] EMF Model Repair: Eclipse-based Tool, https://www.uni-marburg.de/fb12/arbeitsgruppen/swt/nassarn/modelrepair/index.html, (visited 2019)

[142] OCL2AC: Eclipse-based Tool, https://ocl2ac.github.io/home/, (visited 2019)

[143] Truyen, F.: The Fast Guide to Model Driven Architecture: The Basics of Model Driven Architecture. Cephas Consulting Corp (2006), https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf, (visited 2019)

[144] Viatra. https://www.eclipse.org/viatra/, (visited 2019)

[145] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013), http://www.dslbook.org

[146] Wang, J., Kim, S., Carrington, D.A.: Automatic generation of test models for model transformations. In: 19th Australian Software Engineering Conference (ASWEC 2008), March 25-28, 2008, Perth, Australia. pp. 432–440 (2008). https://doi.org/10.1109/ASWEC.2008.42, http://doi.ieeecomputersociety.org/10.1109/ASWEC.2008.42

[147] Xtext. https://www.eclipse.org/Xtext/, (visited 2019)

[148] Yazdi, H.S., Angelis, L., Kehrer, T., Kelter, U.: A framework for capturing, statistically modeling and analyzing the evolution of software models. Journal of Systems and Software **118**, 176–207 (2016). https://doi.org/10.1016/j.jss.2016.05.010, https://doi.org/10.1016/j.jss.2016.05.010

[149] Yue, T., Ali, S.: Empirically evaluating OCL and java for specifying constraints on UML models. Software and System Modeling **15**(3), 757–781 (2016). https://doi.org/10.1007/s10270-014-0438-9, https://doi.org/10.1007/s10270-014-0438-9

[150] Zschaler, S., Mandow, L.: Towards model-based optimisation: Using domain knowledge explicitly. In: Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna, Austria, July 4-8, 2016, Revised Selected Papers. pp. 317–329 (2016). https://doi.org/10.1007/978-3-319-50230-4_24, https://doi.org/10.1007/978-3-319-50230-4_24

A Appendix for Part I

A.1 Catalog of Rules' Schemes and their Meta-Patterns

In the following, we present all the rule schemes and their meta-patterns which are used by EMF repair (in Chapter 3). The rules are defined to deal with an arbitrary instance model violating possibly the negative and positive constraints (the lower and upper bounds). They are designed in a way that each successful application of a rule enhances (or at least preserves) the consistency of the model. Detailed information about the solution design of the rules is presented in Section 3.3.2. The rules are derived using different kinds of meta-patterns. Each rule scheme may comprise positive or negative application conditions that are derived from the given meta-model as well.

- **Set of required-node-creation rules**: For *each* containment type *with* lower bound > 0, a rule is derived which creates a node with its containment edge if the *lower-bound invariant* of the corresponding containment type *is not yet reached.* The container is determined by the input parameter p. The user gets the possibility to select the right container; otherwise, it is chosen randomly by the algorithm. Figure A.1 illustrates the rule scheme used to derive this kind of rules. A node of type B is created in an A-node if $NACm$ is satisfied, i.e., if there is no m B-nodes contained in the A-node. Note that for each non-abstract class type B' in clan(B) such a rule is derived.
- **Set of required-edge-insertion rules**: For *each* non-containment edge type *with* one of its lower bounds m or k being > 0, a rule is generated which inserts a corresponding edge if the composite application condition NAC at the bottom of Figure A.2 is satisfied. Condition $NACp$ forbids to insert an edge between two nodes if there is already one of that type. Conditions $NACm$ and $NACl$ check if lower bound m has not been reached and the upper bound l has not

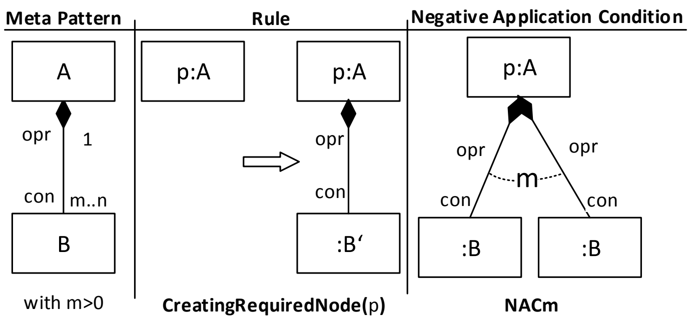


Figure A.1: Rule scheme for creating a required *B*-node

been exceeded. Then, an edge has to be inserted between the given nodes, and this is possible. Similarly, conditions *NACk* and *NACn* may be fulfilled; i.e., the lower bound *k* has not been reached, and the upper bound *n* has not been exceeded. Note, the NAC in Figure A.2 is generated in case that there is an opposite non-containment edge type and $0 < k, l, m, n < *$. For all other cases the NAC is adapted correspondingly. I.e., if the edge type is not opposite or $((k = 0)and(l = *))$, $NAC = NACp$ and $NACm$. If $((k = 0)and(n = *))$, $NAC = NACp$ and $NACm$ and $NACl$. Similarly, the NAC is specified regarding the bounds of the other edge direction.

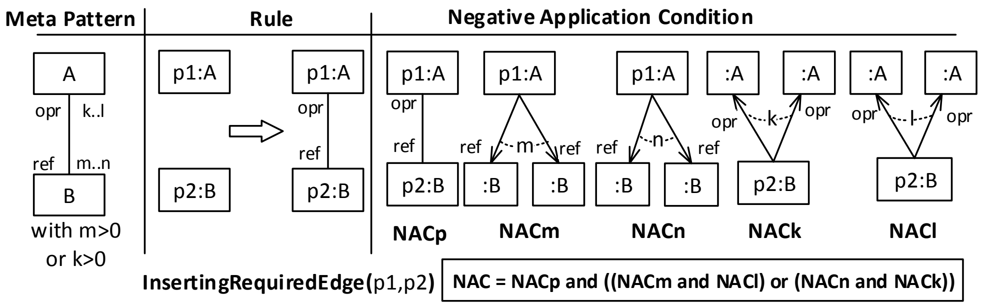


Figure A.2: Rule scheme for inserting a required non-containment edge

For each required non-containment edge type *ref* being a loop, two rule schemes are derived: The first rule inserts a *ref*-edge between two different nodes of the same type *A* if the composite application condition NAC is satisfied as shown in Figure A.3. This insertion can take place if there is not already a *ref*-edge between these nodes *(NACref)*, there are not already *m ref*-edges going out of the source node to other *A*-nodes*(NACm)*, and there is no loop *ref*-edge at the source node plus *m-1 ref*-edges running to other *A*-nodes *(NACrefm-1)*. The second rule scheme inserts a loop *ref*-edge at an *A*-node if the composite application condition NAC is satisfied as shown in Figure A.4. This loop can be inserted if there is not already a loop *ref*-edge at the *A*-node *(NACref)* and if there are not *m ref*-edges running to other *A*-nodes *(NACm)*.

- **Set of additional-node-creation rules:** Figure A.5 shows the meta-pattern as well as the resulting rule scheme for creating a node of type *B* being contained

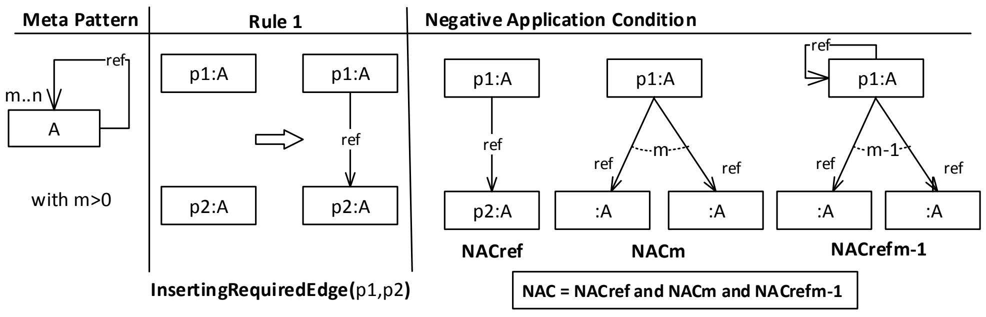


Figure A.3: First rule scheme for inserting a required non-containment loop

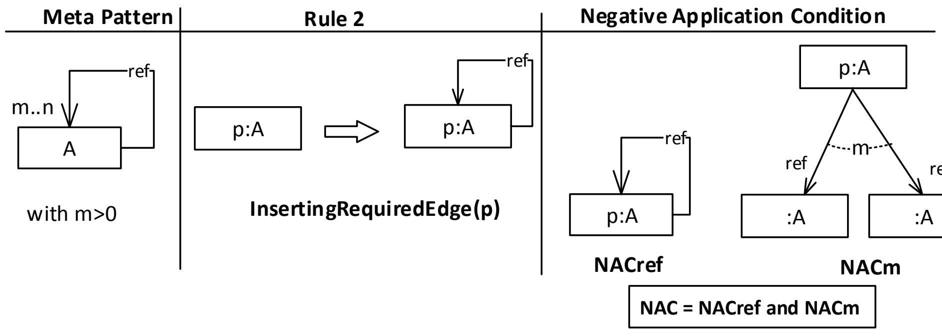


Figure A.4: Second rule scheme for inserting a required non-containment loop

in an A-node. This rule scheme can be applied if $NACn$ is satisfied, i.e., if there are not n B-nodes already contained in this A-node.

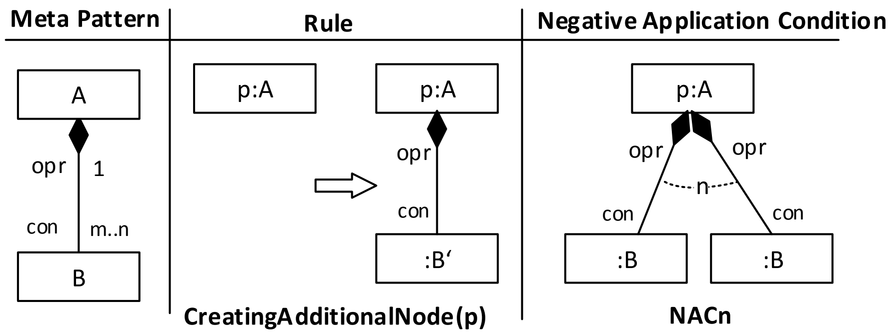


Figure A.5: Rule scheme for creating an additional B-node

- **Set of required-edge-checking rules:** This rule scheme checks if a required edge is missing. If a rule of this scheme can be successfully applied, there is at least one missing required edge. Such a rule scheme is generated for *each* non-containment edge type with $m > 0$. This meta-pattern and the resulting rule scheme are shown in Figure A.6.

  If the required non-containment edge type is a loop, another rule scheme is de-

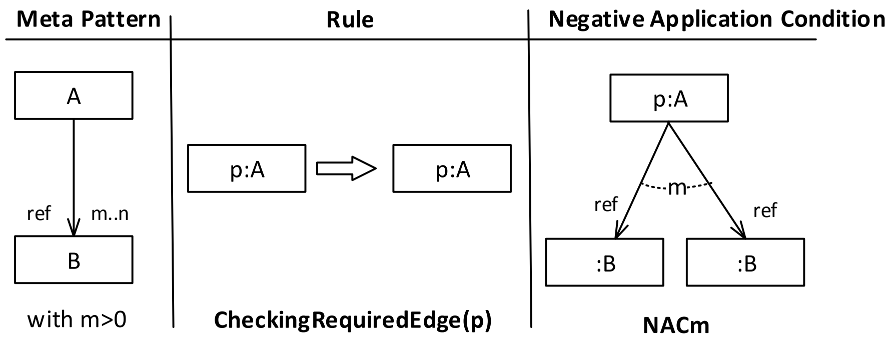


Figure A.6: Rule scheme for checking the existence of required non-containment edges

rived. The meta-pattern and the resulting rule scheme are shown in Figure A.7. A rule of this scheme is applicable if there are not already *m ref*-edges running from the selected *A*-node to other *A*-nodes *(NACm)* and there is not a loop *ref*-edge at the selected node plus *m-1 ref*-edges running to other *A*-nodes *(NACrefm-1)*.

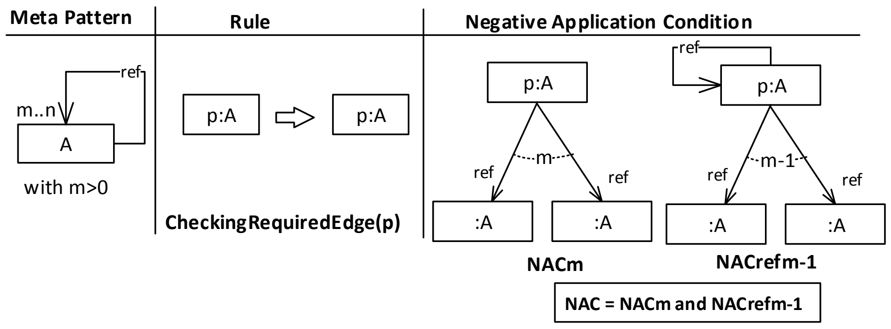


Figure A.7: Rule scheme for checking the existence of required non-containment loop edges

- **Set of exceeding-edge-removing rules:** The meta-patterns as well as the resulting rule schemes are shown in Figures A.8, A.9, and A.10. The derived rule schemes are used to remove exceeding (loop) edges in a correct way (i.e., without violating the *fulfillment* of their upper-bound invariants) The rule scheme in Figure A.8 removes an edge between an *A*-node and a *B*-node if the *PAC* is true, i.e., if there are *n+1 ref*-edges running from this *A*-node to other *B*-nodes in the model *(Cn+1)* or if there are *l+1 opr*-edges running from this *B*-node to other *A*-nodes in the model *(Cl+1)*.

  The rule scheme in Figure A.9 removes an edge of a loop type between two different nodes of type *A* (a source node and a target node) if the *PAC* is true, i.e., if there are *n+1 ref*-edges running from this source *A*-node to other *A*-nodes in the model *(Cn+1)* or if there are a loop *ref*-edge running from this source *A*-node to itself and *n ref*-edges running from this source *A*-node to other

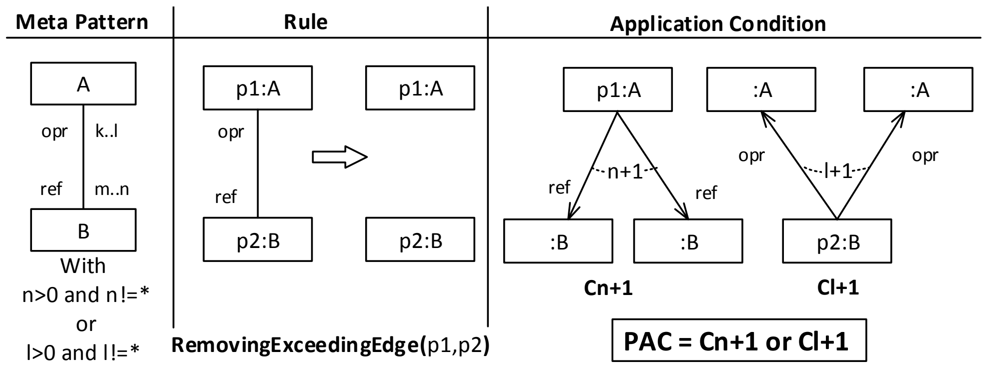


Figure A.8: Rule scheme for removing an exceeding edge

A-nodes in the model *(Cnref)*. (In a similar way, the existence of *opr*-edges has to be considered.)

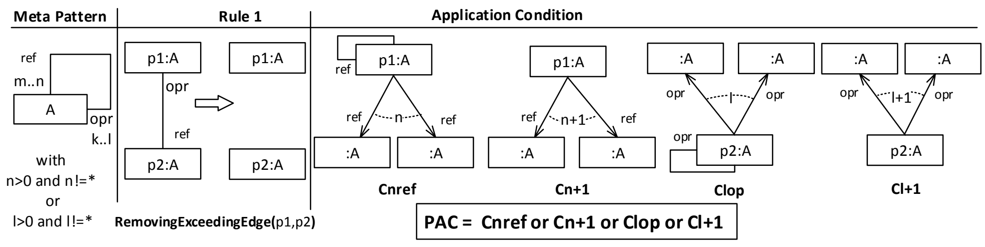


Figure A.9: First rule scheme for removing an exceeding loop edge

The rule scheme in Figure A.10 removes a loop edge at an *A*-node if the *PAC* is true, i.e., if the *A*-node has already a loop edge to itself *(Cref)* and if there, in addition, are *n ref*-edges running from this *A*-node to other *A*-nodes in the model *(Cn)* or *l opr*-edges running from the other *A*-node to further *A*-nodes in the model *(Cl)*.

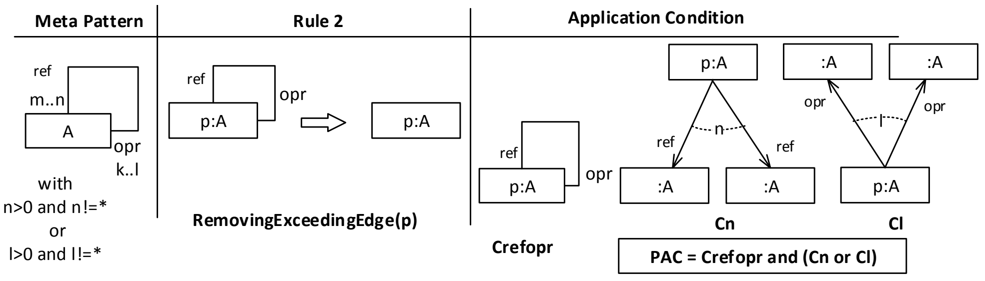


Figure A.10: Second rule scheme for removing an exceeding loop edge

- **Set of exceeding-node-finding rules:** For *each* containment type with a limited upper bound in the given meta-model, a rule is derived which finds an exceeding (supernumerous) node. The rule is applicable if there is at least one exceeding node in a given model. The meta-pattern and the resulting rule scheme are shown in Figure A.11. This rule scheme comprises an output parameter *output* to retrieve an exceeding model node. The rule is designed to comprise two nodes (the container node and the contained node). The contained node is specified with an output parameter. The condition *Cn* checks if there are *n* additional *B*-nodes yielding *n+1* *B*-nodes altogether, i.e., exceeding the upper bound.

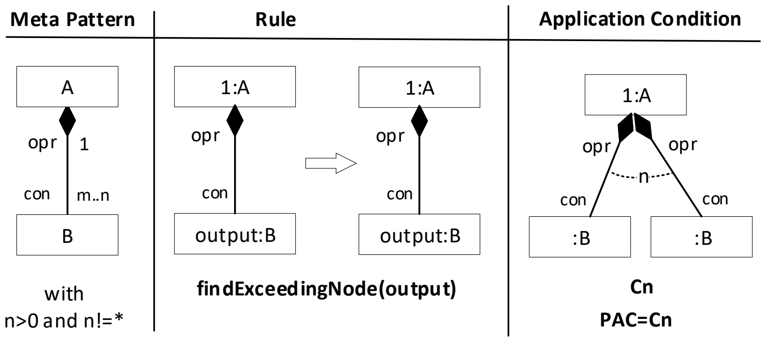


Figure A.11: Rule scheme for finding a supernumerous node

- **Set of node-content-deleting rules:** For *each* path of containment types of the given meta-model, a rule is derived. This rule scheme deletes a direct or indirect contained node of a given node. Given a path longer than 1, there is a rule for each sub-path. If the meta-model has such path meta-pattern as shown in Figure A.12, for example, two rules are derived. The first one deletes a *B*-node being contained in the given *A*-node and the other rule deletes a *C*-node being contained in a *B*-node being contained in the given *A*-node. Note that these rules aim to delete node contents. Each rule is configured with a parameter to delete nodes contained in a given node.

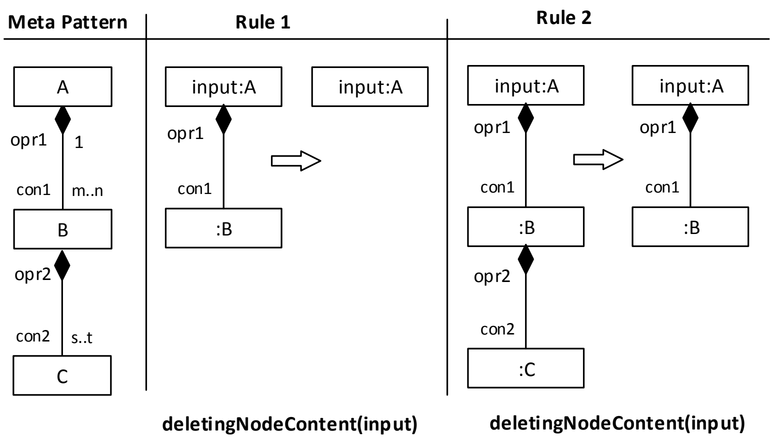


Figure A.12: Rule scheme for deleting the node-contents of a given node

- **Set of target-node-deleting rules:** For *each* containment type of a given meta-model, this rule scheme deletes a given node. The meta-pattern and the resulting rule scheme are shown in Figure A.13. Each rule is configured with a parameter to delete the given node if possible, i.e., once the given node has no contents and no non-containment edges.

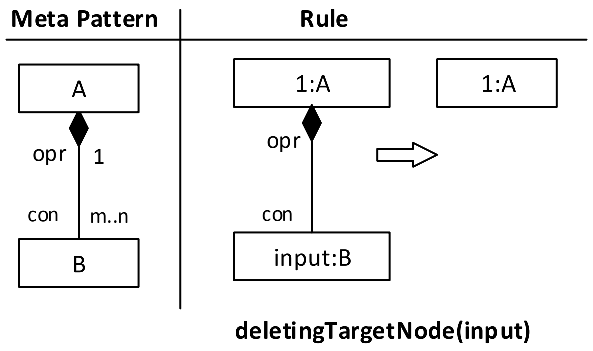


Figure A.13: Rule scheme for deleting a given node

- **Set of edge-removing rules:** For *each* non-containment type of the given meta-model, a rule scheme is generated which removes an edge. The meta-patterns and the resulting rule scheme are shown in Figure A.14 and Figure A.15. Note that these rules aim to delete node contents. Each rule is configured with a parameter to remove the non-containment edges of a given node.

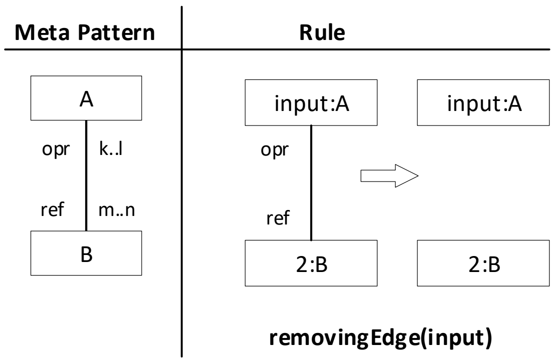


Figure A.14: Rule scheme for removing an edge

- **Set of node-moving rules:** For *each* containment type with a limited upper bound in the given meta-model, a rule is derived which moves a contained node from its container node to another container node without violating the corresponding upper bound of the target container node. The meta-pattern and the resulting rule scheme are shown in Figure A.16. The contained node, its container node and the target container node are specified with input parameters

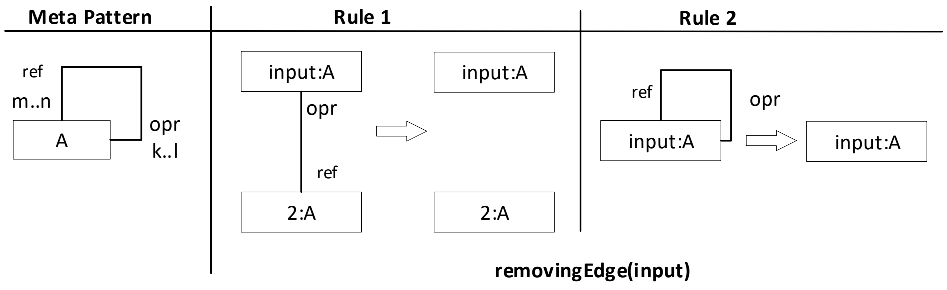


Figure A.15: Rule scheme for removing a loop edge

pa, *pb* and *pto*, respectively. A rule of this scheme is applicable if there are not already *n B-nodes* contained by the selected target container node.

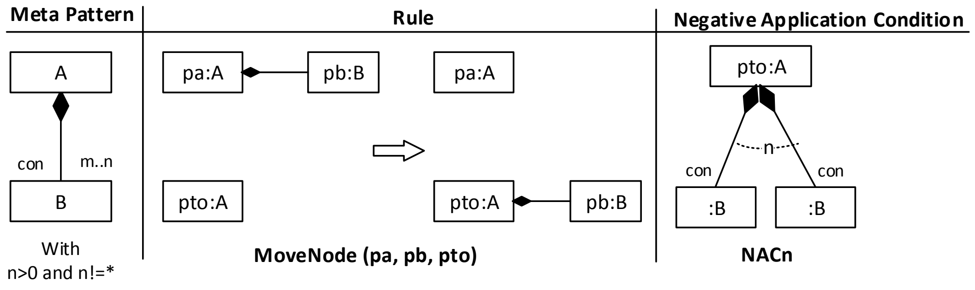


Figure A.16: Rule scheme for moving a contained node

In a similar way, Figure A.17 shows the rule scheme derived from a loop containment type with a limited upper bound in the given meta-model.

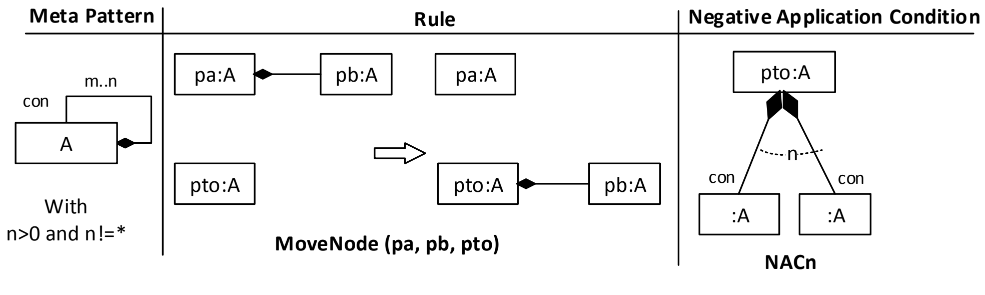


Figure A.17: Rule scheme for moving a contained node

- **Set of edge-moving rules:** For *each* non-containment type with a limited upper bound in the given meta-model, a rule is derived which moves an edge from its source node to another source node if the composite application condition *NAC* at the bottom of Figure A.18 is satisfied. Condition *NACp* forbid to insert an edge between two nodes if there is already one of that type. Condition *NACn* checks if the upper bound n has not been reached.

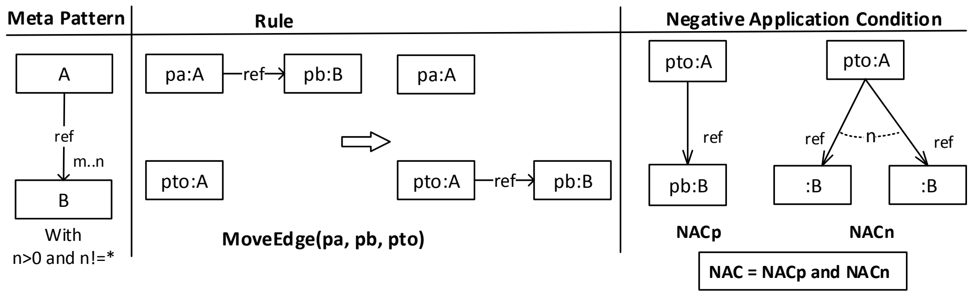


Figure A.18: Rule scheme for moving an edge

Similarly, Figure A.19 and Figure A.20 show the rule schemes derived from a loop non-containment type with a limited upper bound in the given meta-model.

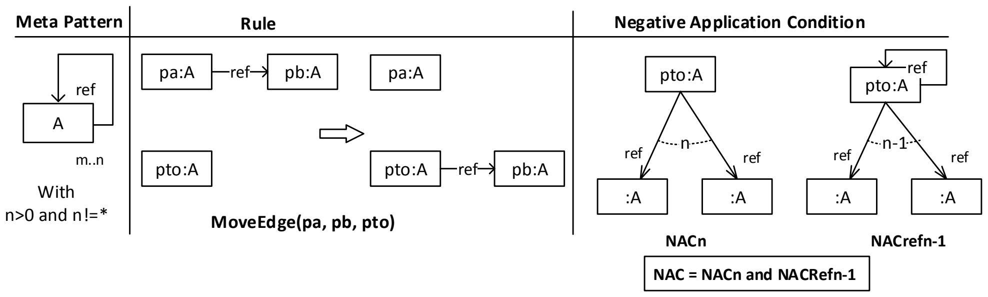


Figure A.19: Rule scheme for moving a loop edge

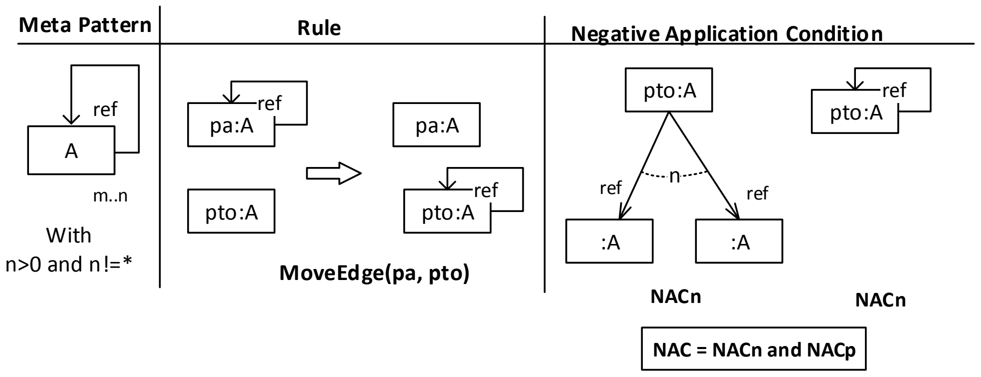


Figure A.20: Rule scheme for moving a loop edge

A.2 Rules' Examples

We demonstrate the derivation of transformation rules at the *Webpage* meta-model in Figure 3.2.

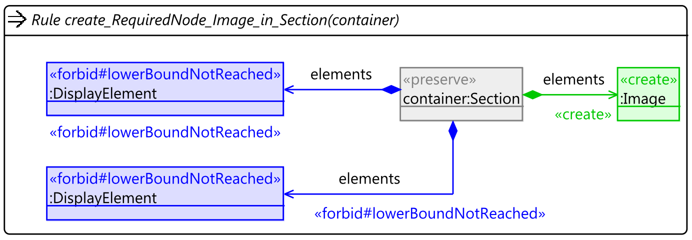

Rule create_RequiredNode_Image_in_Section(container)
«forbid#lowerBoundNotReached»
:DisplayElement
elements
«preserve»
container:Section
elements
«create»
:Image
«create»
«forbid#lowerBoundNotReached»
«forbid#lowerBoundNotReached»
:DisplayElement
elements
«forbid#lowerBoundNotReached»

Figure A.21: A rule for creating a required node of type *Image*

An example of the set of *required-node-creation rules*: Figure A.21 presents the rule for creating an *Image*-node being contained in an existing *Section*-node if there are not already two nodes of type *DisplayElement*contained. This rule is derived due to the lower bound of edge type *elements*. Similar rules have to be derived for edge types *header*, *body*, *navbars*, and *anchors*. Note that there have to be two node creation rules for *elements* since there are two different concrete target node types.

An example of the set of *required-edge-insertion rules*: Figure A.22 presents the derived rule for inserting an edge of type *target* between two existing nodes: from an *Anchor*-node to a *DisplayElement*-node assuming that the source node is not already connected to a node of type *DisplayElement* by a *target*-edge. The opposite *linked*-edge is automatically inserted in EMF as well.

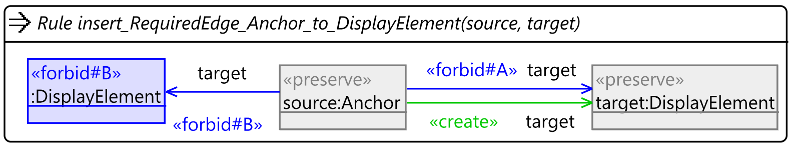


Figure A.22: A rule for inserting a required non-containment edge of type *target*

An example of the set of *required-edge checking rules*: Figure A.23 presents the rule which checks if a given *Anchor*-node is connected to a node of type *DisplayElement* by a *target*-edge. If this rule is applicable to a given instance model, there is a missing required edge of type *target*.

An example of the set of the *additional-node-creation rules*: Figure A.24 presents a rule for creating a *Footer*-node being contained in a *WebPage*-node if there are not already two *Footer*-nodes in the *WebPage*-node. This rule is derived due to edge type *footers*. Similar rules have to be derived for edge types *header*, *body*, *navbars*, *sections*, *subSections*, *elements*, *eanchor*, *footers*, *labels*, and *url*.

An example of the set of *exceeding-edge-removing rules*: Figure A.25 presents the derived rule for removing an *active*-edge between two existing nodes: from a *Hyper-*

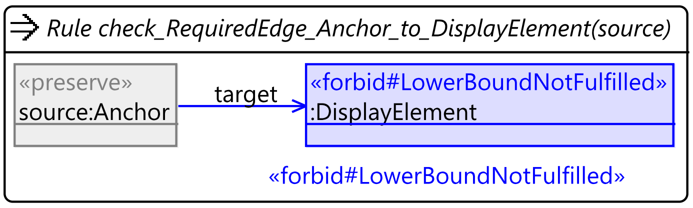

Rule check_RequiredEdge_Anchor_to_DisplayElement(source)
«preserve»
source:Anchor
target
«forbid#LowerBoundNotFulfilled»
:DisplayElement
«forbid#LowerBoundNotFulfilled»

Figure A.23: A rule for checking the existence of a required edge of type *target*

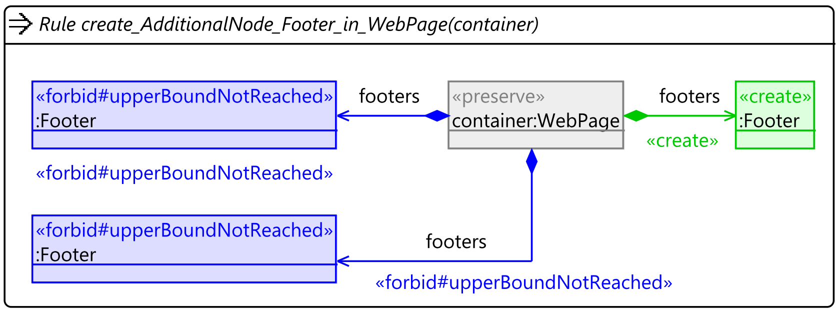

Rule create_AdditionalNode_Footer_in_WebPage(container)
«forbid#upperBoundNotReached»
:Footer
footers
«preserve»
container:WebPage
footers
«create»
:Footer
«create»
«forbid#upperBoundNotReached»
«forbid#upperBoundNotReached»
:Footer
footers
«forbid#upperBoundNotReached»

Figure A.24: A rule for creating an additional node of type *Footer*

Label-node to a *URL*-node assuming that the source node is already connected to an *URL*-node (including the *url*-edge) by an *active*-edge. A similar rule has to be derived for edge type *target*. Note that the upper bound of the edge type *linked* is unlimited. Therefore, no rule is derived for it.

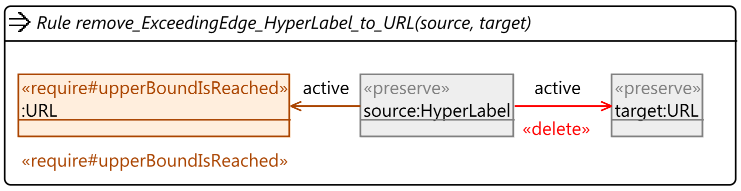

Rule remove_ExceedingEdge_HyperLabel_to_URL(source, target)
«require#upperBoundIsReached»
:URL
active
«preserve»
source:HyperLabel
active
«preserve»
target:URL
«delete»
«require#upperBoundIsReached»

Figure A.25: A rule for removing a supernumerous *target*-edge

Figure A.26 presents two rules for deleting the node contents of a given *Footer*-node. The first rule deletes an *URL*-node contained in a *HyperLabel*-node being contained in the *Footer*-node. The other rule deletes a *HyperLabel*-node being contained in the *Footer*-node. The *Footer*-node is given as an input parameter to these rules.

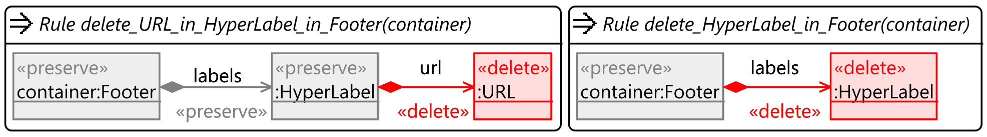


Figure A.26: Two rules for deleting a node (transitively) contained in a given *Footer*-node

An example of the set of *edge-removing rules*: Figure A.27 presents the rule for removing an *active*-edge from a *HyperLabel*-node to a *URL*-node.

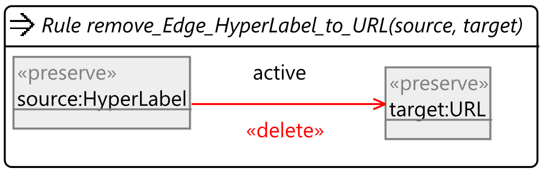


Figure A.27: A rule for removing an *active*-edge

An example of the set of *target-node-deleting rules*: Figure A.28 presents the rule for deleting the *Footer*-node (given as an input parameter) from a *WebPage*-node.

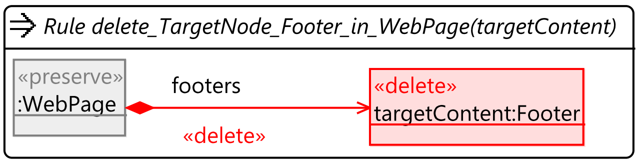


Figure A.28: A rule for deleting a given *Footer*-node

An example of the set of *exceeding-node-finding rules*: Figure A.29 presents the rule for finding a supernumerous *Footer*-node in a *WebPage*-node and retrieving it as an output parameter.

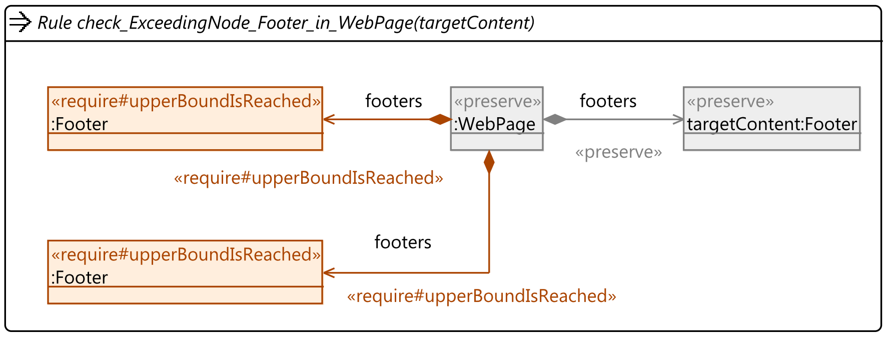

Rule check_ExceedingNode_Footer_in_WebPage(targetContent)
«require#upperBoundIsReached»
:Footer
footers
«require#upperBoundIsReached»
«preserve»
:WebPage
footers
«preserve»
«preserve»
targetContent:Footer
«require#upperBoundIsReached»
:Footer
footers
«require#upperBoundIsReached»

Figure A.29: A rule for finding a supernumerous *Footer*-node

An example of the set of *node-moving rules*: Figure A.30 presents a rule for moving a *URL*-node from its container *HyperLable*-node to another proper *HyperLable*-node. The input parameters *pa, pb* and *pto* are specified by the user.

An example of the set of *edge-moving rules*: Figure A.31 presents a rule for moving an *active*-edge from its source *HyperLable*-node to another proper *HyperLable*-node. The input parameters *pa, pb* and *pto* are specified by the user.

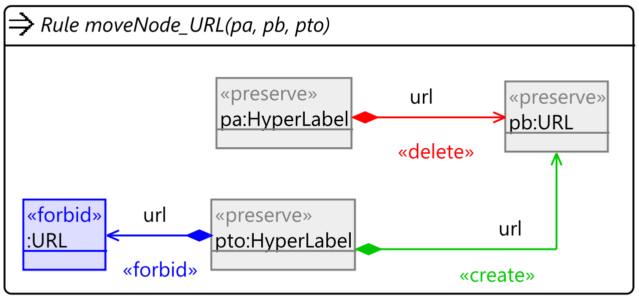


Figure A.30: A rule for moving a URL-node

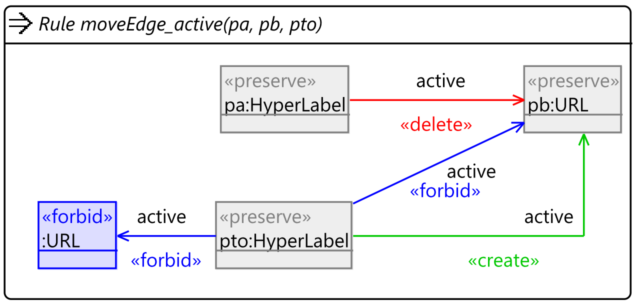


Figure A.31: A rule for moving an *active*-edge

A.3 Unit-based Template of the Repair Algorithm

In the following, we represent the repair algorithm (in Section 3.3.1) as transformation units. For the representation purpose, we use Henshin transformation units. Henshin is a language and tool environment for EMF model transformation [3]. A transformation unit may compose sub-units and rules.

The algorithm is represented a sequential unit *RepairModel* composing two sequential units: (1) *TrimInstance* and (2) *CompleteInstance*.

Model Trimming

Figure A.32 presents the transformation unit template of the control flow of model trimming. It consists of a sequential unit *TrimInstance* composing two layer units: (1) *Remove_All_ExceedingEdges* and (2) *Delete_All_ExceedingNodes*. Note, a layer unit is represented as a loop unit composing an independent unit. The layer unit *Remove_All_ExceedingEdges* is configured with all the derived *exceeding-edge-removing* rules whereas the layer unit *Delete_All_ExceedingNodes* composes conditional units. A conditional unit is of form *CheckThenDelete_ExceedingNode_N_in_R*. It is specified to check the existence of an exceeding node of type *N* and to delete it in a proper way. Its *if-statement* composes the corresponding *exceeding-node-finding* rule. If an *exceeding-node-finding* rule is applicable, there is an exceeding node and thus an exceeding node is retrieved. The *then-statement* can be represented as a sequential unit which deletes a given node of type *N* and its content in a proper way. The *else-statement* returns false. For each containment type whose upper bound is limited such conditional unit is derived. Please note that we consider that a conditional unit is applied *successfully*, i.e., returns true if the *then-statement* is executed. Otherwise, it returns the value of the execution of the *else-statement.* The sequential unit *Delete_SelectedNode_N_in_R_Hierarchy* is specified to delete a node and its content in a proper way as follows: (1) All the edges of the selected node and its contents are removed. (2) All the node contents are deleted in a bottom-up way, i.e, the deletion starts at the node leaves. Finally, the selected node, which is empty now, is deleted using an independent unit *Delete_SelectedNode_N_From_R* being configured with the *target-node-deleting* rules. Steps (1) and (2) could be specified using foreach units, which is not yet introduced in Henshin, with the help of a layer unit configured with *edge-remove* rules and a layer unit configured with the *target-node-deleting* rules, respectively. Step (2) could also be specified as a layer unit being configured with the *target-node-deleting* rules. Note, with the help of parameters, the rules are identified.

Model Completion

Figure A.32 presents the unit transformation template of the control flow of model completion. It consists of a sequential unit *CompleteInstance* composing three units as follows: (1) a layer unit *CreateAllRequiredNodes*, (2) a layer unit *ConnectCorrelatedNodes* and (3) a conditional unit *CheckAndValidate_AllRequiredEdges.* The layer unit *CreateAllRequiredNodes* is configured with all the derived *required-node-creation* rules and the layer unit *ConnectCorrelatedNodes* is configured with all the derived *required-edge-insertion* rules. The conditional unit *CheckAndValidate_AllRequiredEdges* is specified as follows: The *if-statement* is specified as an independent unit being configured with all the derived *required-edge-checking* rules. The *else-statement* is specified to return true. The *then-statement* is specified as conditional unit. The goal of this conditional unit is: (1) to find unfulfilled required edge and create its missing node, and then (2) call the model completion again. Its *if-statement* is specified as an independent unit *AddOneMissingNodeForUnfulfilledRequiredEdge* whereas the *then-statement* is specified to call the sequential unit *CompleteInstance.* The later independent unit is configured with conditional units of form *Is_RequiredEdge_E_to_Node_N_Missing.* For each required non-containment edge type, a such conditional unit is derived. The conditional unit *Is_RequiredEdge_E_to_Node_N_Missing* checks if there is a missing required edge of type *E.* The *if-statement* of it is configured with a corresponding rule of kind *required-edge-checking.* The check rule is applicable if and only if there is a missing edge of type *E* to a node of type *N.* In this case, the *then-statement* is called. Otherwise, the *else-statement* returns false. The *then-statement* is specified as a priority unit *Add_MissingNode_N* composing two ordered sub-units: (1) an independent unit *Add_Node_N_immediately.* (2) an independent unit *Add_Container_for_Node_N.* These two independent units are configured with the corresponding rules of kind *additional-node-creation.* The priority unit adds a node of type *N* immediately in an existing container node without violating the respective upper bound. If it failed, it adds a container node of a container type for the type N without violating the respective upper bound.

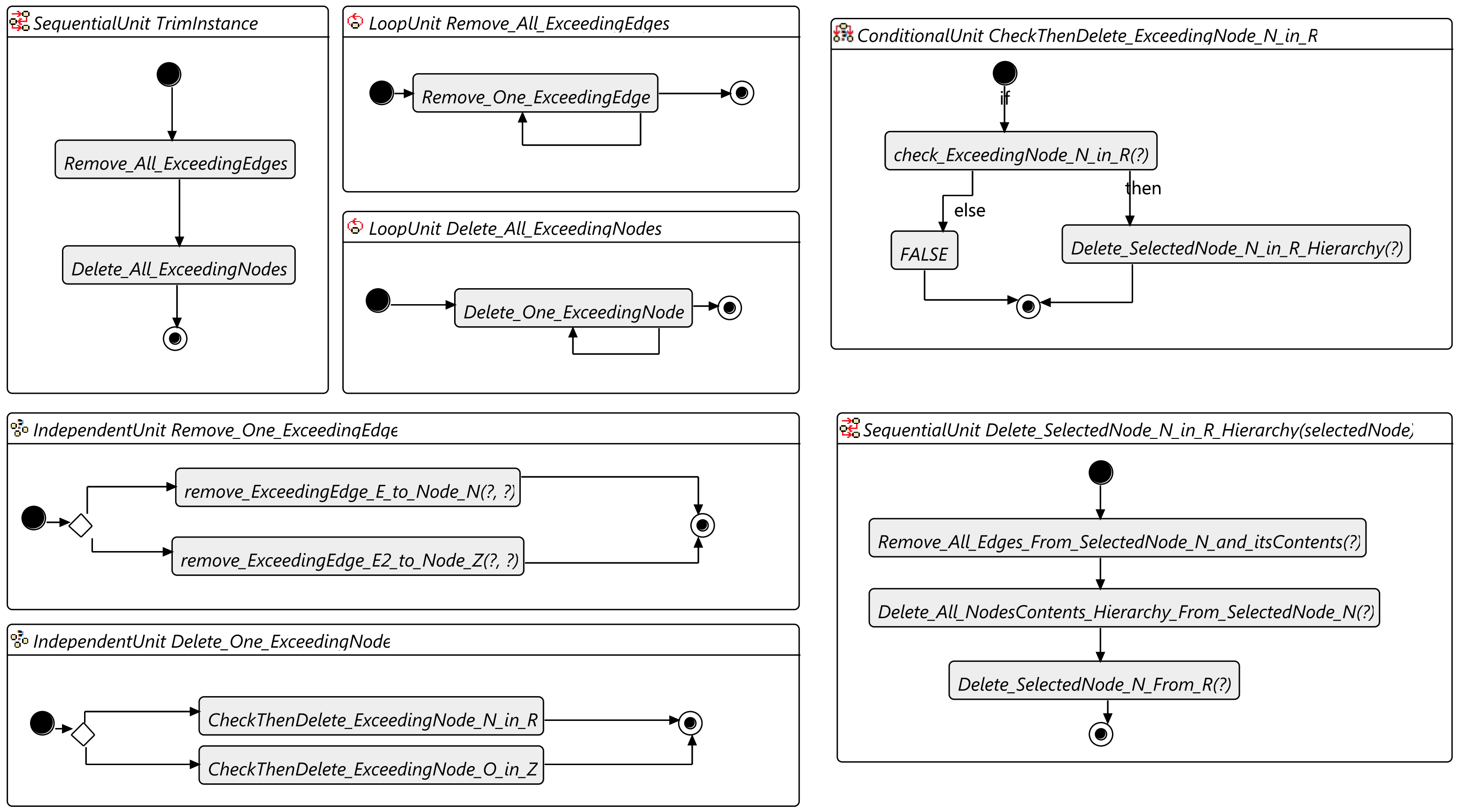

SequentialUnit TrimInstance
Remove_All_ExceedingEdges
Delete_All_ExceedingNodes
LoopUnit Remove_All_ExceedingEdges
Remove_One_ExceedingEdge
LoopUnit Delete_All_ExceedingNodes
Delete_One_ExceedingNode
ConditionalUnit CheckThenDelete_ExceedingNode_N_in_R
if
check_ExceedingNode_N_in_R(?)
else
then
FALSE
Delete_SelectedNode_N_in_R_Hierarchy(?)
IndependentUnit Remove_One_ExceedingEdge
remove_ExceedingEdge_E_to_Node_N(?, ?)
remove_ExceedingEdge_E2_to_Node_Z(?, ?)
SequentialUnit Delete_SelectedNode_N_in_R_Hierarchy(selectedNode)
Remove_All_Edges_From_SelectedNode_N_and_itsContents(?)
Delete_All_NodesContents_Hierarchy_From_SelectedNode_N(?)
Delete_SelectedNode_N_From_R(?)
IndependentUnit Delete_One_ExceedingNode
CheckThenDelete_ExceedingNode_N_in_R
CheckThenDelete_ExceedingNode_O_in_Z

Figure A.32: Transformation unit template for model trimming

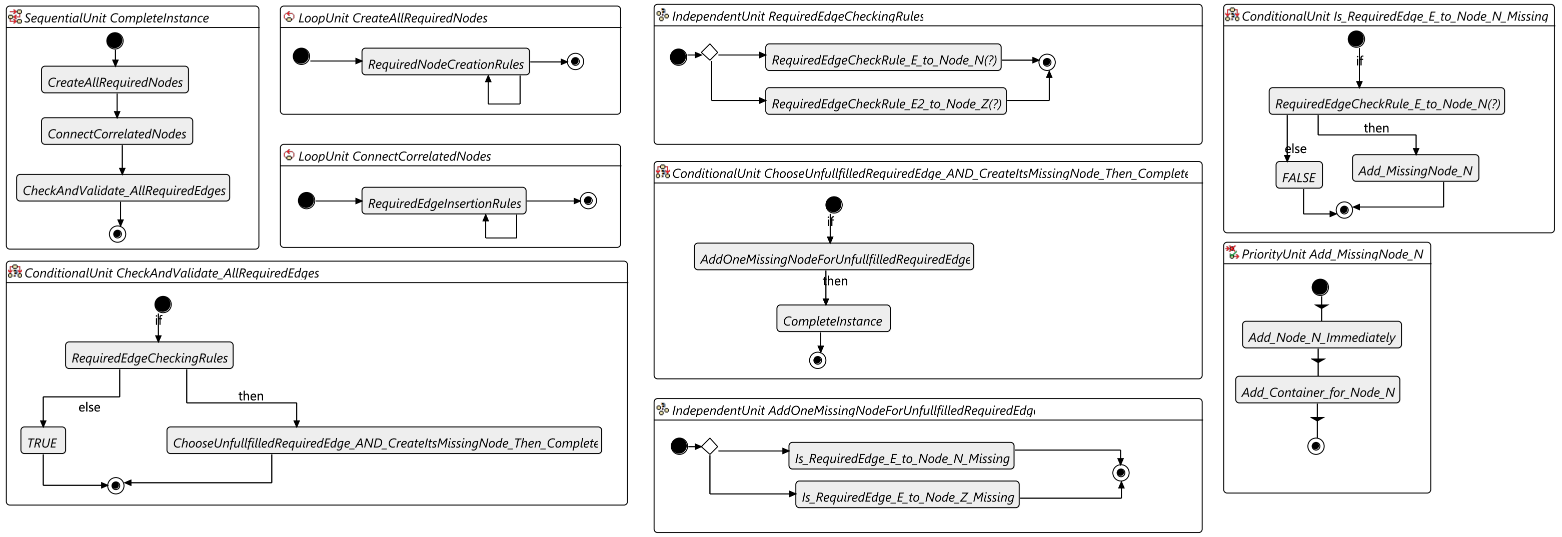

SequentialUnit CompleteInstance
CreateAllRequiredNodes
ConnectCorrelatedNodes
CheckAndValidate_AllRequiredEdges
LoopUnit CreateAllRequiredNodes
RequiredNodeCreationRules
LoopUnit ConnectCorrelatedNodes
RequiredEdgeInsertionRules
ConditionalUnit CheckAndValidate_AllRequiredEdges
if
RequiredEdgeCheckingRules
else
then
TRUE
ChooseUnfullfilledRequiredEdge_AND_CreateItsMissingNode_Then_Complete
IndependentUnit RequiredEdgeCheckingRules
RequiredEdgeCheckRule_E_to_Node_N(?)
RequiredEdgeCheckRule_E2_to_Node_Z(?)
ConditionalUnit ChooseUnfullfilledRequiredEdge_AND_CreateItsMissingNode_Then_Complete
if
AddOneMissingNodeForUnfullfilledRequiredEdge
then
CompleteInstance
IndependentUnit AddOneMissingNodeForUnfullfilledRequiredEdge
Is_RequiredEdge_E_to_Node_N_Missing
Is_RequiredEdge_E_to_Node_Z_Missing
ConditionalUnit Is_RequiredEdge_E_to_Node_N_Missing
if
RequiredEdgeCheckRule_E_to_Node_N(?)
then
else
FALSE
Add_MissingNode_N
PriorityUnit Add_MissingNode_N
Add_Node_N_Immediately
Add_Container_for_Node_N

Figure A.33: Transformation unit template for model completion

B Appendix for Part II

B.1 Proofs and Examples

B.1.1 Proofs

This section contains the proofs of the theorems presented in Chapter 6.

Proof (of Theorem 2 in Section 6.3.2). We first show that this replacement results in a graph condition again: Since all morphisms are injective, if a subcondition $\exists(a : C_1 \hookrightarrow C_2)$ is replaced by `false` because of such a violation, so are all subconditions dependent of C_2 in the tree structure of the condition: These subconditions contain the same violation.

We prove the general statement using structural induction.

The statement holds for $c =$`true` since `true'` $=$ `true` and $p \models$`true` for every injective morphism p.

Let $c = \exists(a : P \hookrightarrow C, d)$ be a condition and, by induction hypothesis, $q \models d \Leftrightarrow q \models d'$ for each injective morphism q from C to any EMF-model graph. First, if C neither contains parallel edges nor multiple incoming containment edges to the same node, then $c' = \exists(a : P \hookrightarrow C, d')$. Now, for every injective morphism $p : P \rightarrow G$, where G is any EMF-model graph

$$\begin{aligned} p \models c &\Leftrightarrow \exists\, q : C \hookrightarrow G,\ \text{s.t. } q \circ a = p \text{ and } q \models d \\ &\Leftrightarrow \exists\, q : C \hookrightarrow G,\ \text{s.t. } q \circ a = p \text{ and } q \models d' \\ &\Leftrightarrow p \models c' \ . \end{aligned}$$

Secondly, if C contains parallel edges or multiple incoming containment edges to the same node, then $c' =$ `false`. Therefore, no injective morphism $p : P \hookrightarrow G$ satisfies

c', where G is any graph. But since no EMF-model graph G contains parallel edges or multiple incoming containment edges to the same node, there does not exist any injective morphism $q : C \rightarrow G$ into any EMF-model graph G. Hence, there does never exist an injective morphism q s.t. $q \circ a = p$ and $q \models d'$. In summary, $p \not\models c \Leftrightarrow p \not\models c'$ for all injective morphisms $p : P \hookrightarrow G$ for any EMF-model graph G.

The induction steps for Boolean operators are routine. $\square$ $\square$

The next proof, for all three situations, relies on the notion of parallel independence which we first shortly recall.

Definition 13 (Parallel independence). Given two plain rules $p_i = (L_i \overset{l_i}{\hookleftarrow} K_i \overset{r_i}{\hookrightarrow} R_i)$ with $i = 1, 2$, two direct transformations $G \Rightarrow_{p_1,m_1} H_1$ and $G \Rightarrow_{p_2,m_2} H_2$ via those rules are *parallelly independent* if there exist two morphisms $d_1 : L_1 \rightarrow D_2$ and $d_2 : L_2 \rightarrow D_1$ as depicted below such that $m_1 = f_2 \circ d_1$ and $m_2 = f_1 \circ d_2$. The rules p_1 and p_2 are *parallel independent* if every pair of transformations $H_1 \underset{p_1}{\Leftarrow} G \underset{p_2}{\Rightarrow} H_2$ is.

$$\begin{tikzcd}
R_1 \arrow[d, hook, "n_1"'] & K_1 \arrow[l, hook', "r_1"'] \arrow[r, hook, "l_1"] \arrow[d, hook] & L_1 \arrow[dr, "m_1"'] \arrow[drrr, dashed, "d_1"] & & L_2 \arrow[dl, "m_2"] \arrow[dlll, dashed, "d_2"'] & K_2 \arrow[l, hook', "l_2"'] \arrow[r, hook, "r_2"] \arrow[d, hook] & R_2 \arrow[d, hook, "n_2"] \\
H_1 & D_1 \arrow[l, hook'] \arrow[rr, hook, "f_1"'] & & G & & D_2 \arrow[ll, hook', "f_2"] \arrow[r, hook] & H_2
\end{tikzcd}$$

Proof (of Theorem 1 in Section 6.3.1).

1. Let $G \Rightarrow_p H$ be a transformation via rule p where $G \models c$. By induction, we show the stronger statement that for every condition c over any graph P, there exists an injective morphism $g : P \hookrightarrow G$ with $g \models c$ if and only if there exists an injective morphism $g' : P \hookrightarrow H$ with $g' \models c$. In particular, for constraints $c = \exists (\varnothing \hookrightarrow C, d)$ this implies that $H \models c$ (via the empty morphism) if and only if $G \models c$. We prove the statement by induction over the structure of the condition. If $c = \texttt{true}$, $H \models c \Leftrightarrow G \models c$.

   Let $c = \exists (a : P \hookrightarrow C, d)$, $g : P \hookrightarrow G$ and $g \models c$. By definition, there exists an injective morphism $q : C \hookrightarrow G$ such that $q \circ a = g$ and $g \models d$. By induction hypothesis, there exists an injective morphism $q' : C \hookrightarrow H$ such that $q' \models d$. Consider the constant rule $id_P : P \hookleftarrow P \hookrightarrow P$ (which just checks for the existence of the graph P). The intersection between $L \backslash K$ and P is empty and therefore the rules p and id_P are parallelly independent. In particular, pairs of transformations $G\ {}_{id_P}{\Leftarrow} G \Rightarrow_p H$ are. Hence, for the match morphism g for rule id_P in G, there exists an according match morphism $g' : P \hookrightarrow H$ such that $g' = q' \circ a$ (compare the Local Church-Rosser Theorem and its proof [39]). Thus, $g' \models c$.

In the other direction, let $g' : P \hookrightarrow H$ and $g' \models c$. By definition, there exists an injective morphism $q' : C \hookrightarrow H$ such that $q' \circ a = g'$ and $q' \models d$. By induction hypothesis, there exists an injective morphism $q : C \hookrightarrow G$ such that $q \models d$. Consider, again, the constant rule $id_P : P \hookleftarrow P \hookrightarrow P$. The intersection between $R \backslash K$ and P is empty and therefore the rules p and id_P are sequentially independent. In particular, transformation sequences $G \Rightarrow_{id_P} G \Rightarrow_p H$ are. This is equivalent to p^{-1} and id_P being parallelly independent. Then, again by the Local Church-Rosser Theorem, there is an injective morphism $g : P \hookrightarrow G$ such that $g = q \circ a$. Thus, $g \models c$.

Induction over the Boolean operators is standard, again.

2. The proof for this second situation uses parallel independence in a similar fashion; since the considered constraints are not nested, we do not need an induction. Let H denote a graph which arises by application of a monotonic rule $r : K \hookrightarrow R$ to a graph G, i.e., $G \Rightarrow_r H$. A monotonic rule is parallel independent from the rule which just checks for the existence of the graph C, i.e., each pair of transformations $G\ {}_{id_C}\!\Leftarrow G \Rightarrow_r H$ is, where $id_C : C \hookleftarrow C \hookrightarrow C$ is constant. Hence, every match for the graph C in G extends to a match for graph C in H, i.e., $G \models c \Rightarrow H \models c$.

   Dually, checking the existence of a graph C is sequentially independent of applying a rule $l : L \hookleftarrow K$ which only deletes. Hence, every match for C in a graph H can be extended to a match for C in G, i.e., $H \models \exists\, C \Rightarrow G \models \exists\, C$ and hence $G \models c \Rightarrow H \models c$.

3. Let I be a set indexing the possible overlappings of C and R, i.e., the pairs of jointly surjective, injective morphisms $i_R : R \hookrightarrow P_i, i_C : C \hookrightarrow P_i$. Let $J \subset I$ be the subset of I that consists only of that indices denoting overlappings where at least one newly created element from the rule, i.e., an element from $R \backslash K$ is overlapped with one from C. Let $rac = \neg(\bigvee_{j \in J} \exists\, j_R : R \hookrightarrow P_j)$ be the right application condition of p arising by only considering overlappings from J (and not from the whole set I). Let ac denote the application condition for rule p when moving rac equivalently to the LHS of p.

   We prove the statement by contraction. Thus, let $G \models c = \neg\exists\, C, G \Rightarrow_{\rho=(p,ac),m} H$, so p is equipped with application condition ac, and assume $H \models \neg c = \exists\, C$. This means, there is an injective morphism $q' : C \hookrightarrow H$. By construction of I, there exists an $i \in I$ such that there is an injective morphism $\tilde{q'} : P_i \hookrightarrow H$ with $n = \tilde{q'} \circ i_R$ and $\tilde{q'} \circ i_C = q$ (compare Figure B.1). Since p is equipped with application condition ac, $i \in I \backslash J$; otherwise rule application would have been prevented by that application condition:

$$\begin{aligned} n \not\models \neg\exists\,(i_R : R \hookrightarrow P_i) &\Leftrightarrow n \models \exists\,(i_R : R \hookrightarrow P_i) \\ &\Leftrightarrow \exists\, \tilde{q'} : P_i \hookrightarrow H, \text{ s.t. } n = \tilde{q'} \circ i_R \ . \end{aligned}$$

   Now, like in the proofs above, checking for the existence of $P_i, i \in I \backslash J$ in graph H is independent of applying rule p. Hence, the morphism $\tilde{q'} : P_i \hookrightarrow H$ restricts to an injective morphism $\tilde{q} : P_i \hookrightarrow G$ and therefore $G \models \neg c = \exists\, C$ via the injective morphism $q = \tilde{q} \circ i_C$. This contradicts $G \models c$.

$$\begin{tikzcd}
& & & C \arrow[d, hook, "i_C"] \\
L \arrow[d, hook, "m"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] & R \arrow[r, hook, "i_R"] \arrow[d, hook, "n"] & P_i \arrow[ld, hook', "q'"] \\
G & D \arrow[l, hook', "d_1"] \arrow[r, hook, "d_2"'] & H &
\end{tikzcd}$$

Figure B.1: Existence of satisfying morphism

□

□

B.1.2 C-Preserving Application Conditions as Defined in the Theory [55]: An Example

Using our running example (see Section 6.2), we present the resulting c-preserving application condition ac_{pres} w.r.t. the construction provided by Habel and Pennemann in ([55], Def. 9). The construction of a c-preserving application condition is defined by them using an implication operator as follows:

$$ac_{pres} = (Shift(r^{-1}, c) \Rightarrow ac_{gua}) \tag{B.1}$$

where r^{-1} denotes the inverse rule of the given rule r, c denotes the given constraint and ac_{gua} denotes the c-guaranteeing application condition defined as $Left(r, (Shift(r, c))$ which is the output (of the second component) of *OCL2AC*.

Figure B.2 presents the c-preserving application condition of integrating the constraint `no_region` into the rule `moveRegionFromStateTOState` being constructed according to the formula (B.1). The left column displays the antecedent that expresses that the model was already valid before rule application and the right column displays the consequent of the conditional. The resulting application condition ac_{pres} contains 14 graphs although we have integrated only one constraint into the rule. To further simplify the result, the material implication has to be simplified by applying De Morgan's Law and the distributivity law yielding a formula in conjunctive normal form. It consists of 7 clauses where each clause contains 8 literals. To simplify the particular clauses, subgraph isomorphism checks have to be performed. As soon as the input constraint is really nested, the simplification becomes even more difficult since particular clauses can no longer be simplified by simply checking for subgraph isomorphisms. Our optimizing-by-construction technique simplifies the resulting application conditions *throughout the construction process* without the need for such costly computations. In the best case, as, e.g., in Figure 6.7, we even receive the logically weakest possible application condition in a simplified version as output.

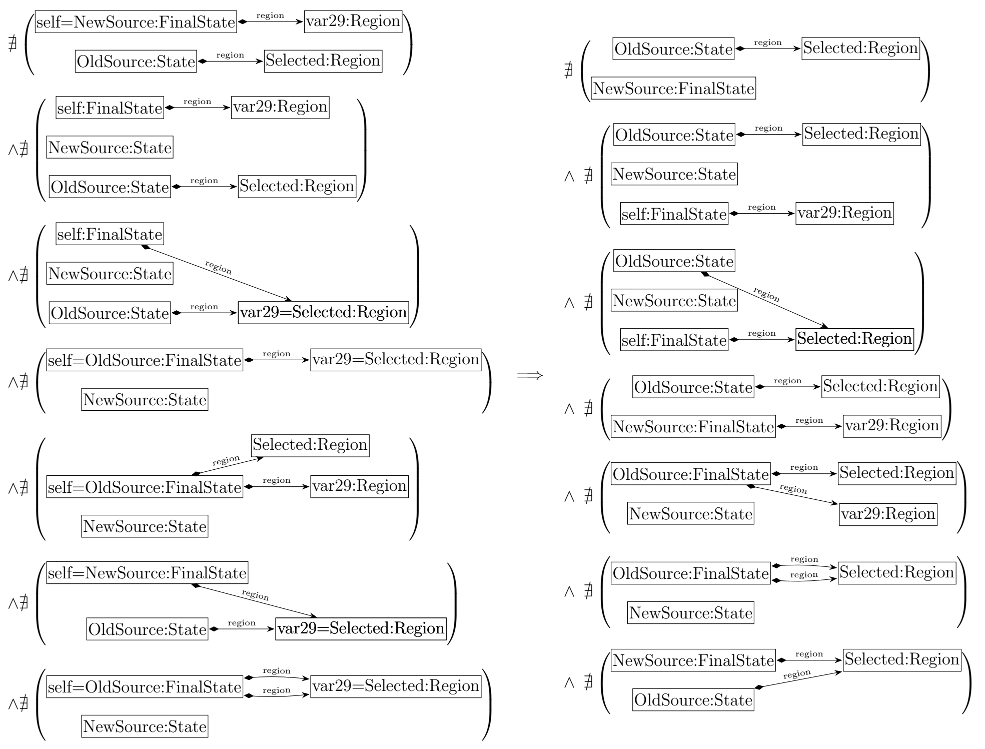

∄ self=NewSource:FinalState region var29:Region
OldSource:State region Selected:Region
∧∄ self:FinalState region var29:Region
NewSource:State
OldSource:State region Selected:Region
∧∄ self:FinalState region
NewSource:State
OldSource:State region var29=Selected:Region
∧∄ self=OldSource:FinalState region var29=Selected:Region
NewSource:State
∧∄ Selected:Region region
self=OldSource:FinalState region var29:Region
NewSource:State
∧∄ self=NewSource:FinalState region
OldSource:State region var29=Selected:Region
∧∄ self=OldSource:FinalState region region var29=Selected:Region
NewSource:State
⟹
∄ OldSource:State region Selected:Region
NewSource:FinalState
∧ ∄ OldSource:State region Selected:Region
NewSource:State
self:FinalState region var29:Region
∧ ∄ OldSource:State region
NewSource:State
self:FinalState region Selected:Region
∧ ∄ OldSource:State region Selected:Region
NewSource:FinalState region var29:Region
∧ ∄ OldSource:FinalState region Selected:Region
region
NewSource:State var29:Region
∧ ∄ OldSource:FinalState region region Selected:Region
NewSource:State
∧ ∄ NewSource:FinalState region Selected:Region
OldSource:State region

Figure B.2: The resulting c-preserving application condition ac_{pres} w.r.t. the construction provided by Habel and Pennemann in [55], compare with our optimized version in Figure 6.7

B.2 List of OCL Constraints

B.2.1 List of OCL Constraints of Petrinet

Table B.1: OCL constraints of Petrinet

ID	OCL Constraints of Petrinet
1	`invariant Example09: self.name <> '';`
2	`invariant Example10a: self.preArc -> notEmpty() or self.postArc -> notEmpty();`
3	`invariant Example10b: self.transition -> forAll(t|t.preArc -> notEmpty() or t.postArc -> notEmpty());`
4	`invariant Example11: self.preArc -> notEmpty() or self.postArc -> notEmpty();`
5	`invariant Example12: self.place -> forAll(p1|self.place -> forAll(p2|p1 <> p2 implies p1.name <> p2.name));`
6	`invariant Example13a: self.place -> exists(p|p.token -> notEmpty());`
7	`invariant Example13b: self.place -> select(p|p.token -> notEmpty()) -> notEmpty();`
8	`invariant Example13c: self.place -> collect(p|p.token) -> notEmpty();`
9	`invariant Example13d: Token.allInstances() -> notEmpty();`
10	`invariant Example14: self.weight >= 1;`
11	`invariant Example15: self.place -> size() >= 2;`

B.2.2 List of OCL Constraints of Statechart

Table B.2: OCL constraints of Statecharts

ID	OCL Constraints of Statecharts
1	`invariant owned: (stateMachine -> notEmpty() implies state -> isEmpty()) and(state -> notEmpty() implies stateMachine -> isEmpty());`
2	`invariant submachine_states: isSubmachineState=false implies connection -> notEmpty();`
3	`invariant destinations_or_sources_of_transitions: self.isSubmachineState=true implies (self.connection -> forAll(cp |cp.entry -> forAll(p|p.stateMachine = self.submachine) and cp.exit -> forAll (p | p.stateMachine = self.submachine)));`
4	`invariant submachine_or_regions: self.isComposite=true implies not (self.isSubmachineState=true);`
5	`invariant composite_states: self.connectionPoint -> notEmpty() implies self.isComposite=true;`
6	`invariant no_outgoing_transitions: self.outgoing -> size() = 0;`
7	`invariant no_regions: self.region -> size() = 0;`
8	`invariant no_entry_behavior: self.enty -> isEmpty();`
9	`invariant no_exist_behavior: self.exit -> isEmpty();`
10	`invariant cannot_reference_submachine: self.submachine -> isEmpty();`
11	`invariant no_state_behavior: self.doActivity -> isEmpty();`

B.2.3 List of OCL Constraints of UML

Table B.3: OCL constraints of UML

ID	OCL Constraints of UML
1	`invariant visibility_public_or_private: self.visibility = VisibilityKind::public orself.visibility = VisibilityKind::private;`
2	`invariant public_or_private: self.visibility = VisibilityKind::public orself.visibility = VisibilityKind::private;`
3	`invariant not_apply_to_self: not self.constrainedElement -> includes(self);`
4	`invariant binary_associations: self.memberEnd -> exists(aggregation <> AggregationKind::none) impliesself.memberEnd -> size() = 2;`
5	`invariant association_ends: (self.memberEnd -> size()>2) implies(self.ownedEnd -> includesAll(self.memberEnd));`
6	`invariant only_return_result_parameters: self.behavior.ownedParameter -> select(p | p.direction<>ParameterDirectionKind::return) -> isEmpty();`
7	`invariant derived_union_is_derived: isDerivedUnion=true impliesisDerived=true;`
8	`invariant derived_union_is_read_only: isDerivedUnion=true impliesisReadOnly=true;`
9	`invariant deployed_elements: self.deployment -> forAll (d | d.location.deployedElement -> forAll (de | de.oclIsKindOf(Component)));`
10	`invariant deployment_target: self.deployment -> forAll (d | d.location.oclIsKindOf(ExecutionEnvironment));`
11	`invariant at_most_one_return: self.ownedParameter -> select(par | par.direction = ParameterDirectionKind::return) -> size() <= 1;`
12	`invariant only_body_for_query: self.bodyCondition -> notEmpty() impliesisQuery=true;`
13	`invariant passive_class: not (self.isActive=true) impliesself.ownedReception -> isEmpty();`
14	`invariant visibility: self.feature -> forAll(f | f.visibility = VisibilityKind::public);`
15	`invariant protocol_transitions: self.region -> forAll(r | r.transition -> forAll(t | t.oclIsTypeOf(ProtocolTransition)));`
16	`invariant classifier_context: (not self.context -> isEmpty()) andself.specification -> isEmpty();`
17	`invariant classifier_context: self.context -> notEmpty() impliesnot self.context.oclIsKindOf(Interface);`
18	`invariant connection_points: self.connectionPoint -> forAll (c | c.kind = PseudostateKind::entryPoint orc.kind = PseudostateKind::exitPoint);`
19	`invariant method: self.specification -> notEmpty() impliesself.connectionPoint -> isEmpty();`

ID	OCL Constraints of UML
20	`invariant entry_or_exit: self.connectionPoint -> forAll(cp|cp.kind = PseudostateKind::entryPoint orcp.kind = PseudostateKind::exitPoint);`
21	`invariant initial_vertex: self.subvertex -> select (v | v.oclIsKindOf(Pseudostate)) -> select(p : Pseudostate | p.kind = PseudostateKind::initial) -> size() <= 1;`
22	`invariant deep_history_vertex: self.subvertex -> select (v | v.oclIsKindOf(Pseudostate)) -> select(p : Pseudostate | p.kind = PseudostateKind::deepHistory) -> size() <= 1;`
23	`invariant shallow_history_vertex: self.subvertex -> select(v | v.oclIsKindOf(Pseudostate)) -> select(p : Pseudostate | p.kind = PseudostateKind::shallowHistory) -> size() <= 1;`
24	`invariant submachine_states: isSubmachineState=true impliesconnection -> notEmpty();`
25	`invariant destinations_or_sources_of_transitions: self.isSubmachineState=true implies(self.connection -> forAll (cp | cp.entry -> forAll (p | p.stateMachine = self.submachine) andcp.exit -> forAll (p | p.stateMachine = self.submachine)));`
26	`invariant submachine_or_regions: self.isComposite=true impliesnot (self.isSubmachineState=true);`
27	`invariant composite_states: self.connectionPoint -> notEmpty() impliesself.isComposite=true;`
28	`invariant entry_pseudostates: self.entry -> notEmpty() impliesself.entry -> forAll(e | e.kind = PseudostateKind::entryPoint);`
29	`invariant exit_pseudostates: self.exit -> notEmpty() impliesself.exit -> forAll(e | e.kind = PseudostateKind::exitPoint);`
30	`invariant initial_vertex: (self.kind = PseudostateKind::initial) implies(self.outgoing -> size() <= 1);`
31	`invariant history_vertices: ((self.kind = PseudostateKind::deepHistory) or(self.kind = PseudostateKind::shallowHistory)) implies(self.outgoing -> size() <= 1);`
32	`invariant join_vertex: (self.kind = PseudostateKind::join) implies ((self.outgoing -> size() = 1) and(self.incoming -> size() >= 2));`
33	`invariant fork_vertex: (self.kind = PseudostateKind::fork) implies((self.incoming -> size() = 1) and(self.outgoing -> size() >= 2));`
34	`invariant junction_vertex: (self.kind = PseudostateKind::junction) implies((self.incoming -> size() >= 1) and(self.outgoing -> size() >= 1));`
35	`invariant choice_vertex: (self.kind = PseudostateKind::choice) implies ((self.incoming -> size() >= 1) and(self.outgoing -> size() >= 1));`
36	`invariant outgoing_from_initial: (self.kind = PseudostateKind::initial) implies(self.outgoing.guard -> isEmpty() andself.outgoing.trigger -> isEmpty());`
37	`invariant is_binary: self.memberEnd -> size() = 2;`

ID	OCL Constraints of UML
38	`invariant aggregation: self.aggregation = AggregationKind::composite;`
39	`invariant control_pins: self.isControl=true implies self.isControlType=true;`
40	`invariant extension_points: self.extensionLocation -> forAll (xp | self.extendedCase.extensionPoint -> includes(xp));`
41	`invariant operands: self.operand-> forAll (op|op.oclIsKindOf(LiteralString));`
42	`invariant subexpressions: if self.subExpression -> notEmpty() then self.operand-> isEmpty() else self.operand-> notEmpty() endif;`
43	`invariant one_output_parameter: self.ownedParameter -> select(p | p.direction=ParameterDirectionKind::out orp.direction=ParameterDirectionKind::inout orp.direction=ParameterDirectionKind::return) -> size() >= 1;`
44	`invariant generalize: self.generalization.general -> forAll(e |e.oclIsKindOf(Stereotype)) and self.generalization.specific -> forAll(e | e.oclIsKindOf(Stereotype));`
45	`invariant consider_and_ignore: ((self.interactionOperator= InteractionOperatorKind::consider) or(self.interactionOperator= InteractionOperatorKind::ignore)) implies self.oclIsTypeOf(ConsiderIgnoreFragment);`
46	`invariant consider_or_ignore: (self.interactionOperator= InteractionOperatorKind::consider) or(self.interactionOperator= InteractionOperatorKind::ignore);`
47	`invariant type: self.message -> forAll(m |m.oclIsKindOf(Operation) orm.oclIsKindOf(Reception) orm.oclIsKindOf(Signal));`
48	`invariant classifier_not_abstract: not (self.classifier.isAbstract = true);`
49	`invariant classifier_not_association_class: not self.classifier.oclIsKindOf(AssociationClass);`
50	`invariant no_type: self.target.type -> size() = 0;`
51	`invariant no_type: self.first.type -> size() = 0 andself.second.type -> size() = 0;`
52	`invariant contained: self.context -> size() = 1;`
53	`invariant not_static: self.structuralFeature.isStatic = false;`
54	`invariant one_featuring_classifier: self.structuralFeature.featuringClassifier -> size() = 1;`
55	`invariant input_pin: self.value.type = self.structuralFeature.featuringClassifier;`
56	`invariant type_of_result: self.result -> notEmpty() impliesself.result.type = self.object.type;`
57	`invariant type_of_result: self.result -> notEmpty() impliesself.result.type = self.object.type;`
58	`invariant property_is_association_end: self.end.association -> size() = 1;`
59	`invariant same_type2: self.value.type = self.end.type;`
60	`invariant end_object_input_pin: self.value -> excludesAll(self.qualifier.value);`

ID	OCL Constraints of UML
61	`invariant qualifier_attribute: self.LinkEndData.end -> collect(qualifier) -> includes(self.qualifier);`
62	`invariant type_of_qualifier: self.value.type = self.qualifier.type;`
63	`invariant one_open_end: self.endData -> select(ed | ed.value -> size() = 0) -> size() = 1;`
64	`invariant create_link_action: self.LinkAction.oclIsKindOf(CreateLinkAction);`
65	`invariant no_regions: self.region -> size() = 0;`
66	`invariant cannot_reference_submachine: self.submachine -> isEmpty();`
67	`invariant no_entry_behavior: self.entry -> isEmpty();`
68	`invariant no_exit_behavior: self.exit -> isEmpty();`
69	`invariant no_state_behavior: self.doActivity -> isEmpty();`
70	`invariant same_type4: self.value.type = self.variable.type;`
71	`invariant has_no: self.generalization -> isEmpty() andself.feature -> isEmpty();`
72	`invariant not_instantiable: isAbstract=true;`
73	`invariant sources_and_targets_kind: (self.source -> forAll(p | p -> oclIsKindOf(Actor) oroclIsKindOf(Node) oroclIsKindOf(UseCase) oroclIsKindOf(Artifact) oroclIsKindOf(Class) oroclIsKindOf(Component) oroclIsKindOf(Port) oroclIsKindOf(Property) or oclIsKindOf(Interface) oroclIsKindOf(Package) oroclIsKindOf(ActivityNode) oroclIsKindOf(ActivityPartition) or oclIsKindOf(InstanceSpecification))) and(self.target -> forAll(p | p -> oclIsKindOf(Actor) oroclIsKindOf(Node) oroclIsKindOf(UseCase) oroclIsKindOf(Artifact) oroclIsKindOf(Class) oroclIsKindOf(Component) oroclIsKindOf(Port) oroclIsKindOf(Property) oroclIsKindOf(Interface) oroclIsKindOf(Package) oroclIsKindOf(ActivityNode) oroclIsKindOf(ActivityPartition) oroclIsKindOf(InstanceSpecification)));`
74	`invariant convey_classifiers: self.conveyed.representation -> forAll(p | p -> oclIsKindOf(Class) oroclIsKindOf(Interface) oroclIsKindOf(InformationItem) oroclIsKindOf(Signal) oroclIsKindOf(Component));`
75	`invariant classifier_not_abstract: not self.newClassifier -> exists(isAbstract = true);`
76	`invariant no_type: self.object.type -> isEmpty();`
77	`invariant boolean_result: self.result.type = Boolean;`
78	`invariant property: self.end.association -> notEmpty();`
79	`invariant association_of_association: self.end.association.oclIsKindOf(AssociationClass);`
80	`invariant type_of_result: self.result.type = self.end.type;`
81	`invariant qualifier_attribute: self.qualifier.associationEnd -> size() = 1;`
82	`invariant association_of_association: self.qualifier.associationEnd.association.oclIsKindOf(AssociationClass)`

ID	OCL Constraints of UML
83	`invariant same_type5: self.result.type = self.qualifier.type;`
84	`invariant unmarshall: self.isUnmarshall = true;`

85	`invariant event_on_reply_to_call_trigger:`
	`self.replyToCall.oclIsKindOf(CallEvent);`

86	`invariant one_outgoing_edge: self.outgoing -> size() = 1;`
87	`invariant associated_actions: self.effect -> isEmpty();`

Typeset 2020-03-13