Philipps Universität Marburg

Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems

Dissertation
for the degree of
Doctor of Natural Sciences

Submitted by
Daniel Strüber, Dipl.-Inf.
born April 23, 1986 in Marburg.

Department of Mathematics
and Computer Science
Philipps-Universität Marburg

University Reference Number: 1180

Marburg, 2016.

Referees:
Prof. Dr. Gabriele Taentzer
Prof. Dr. Reiko Heckel
Prof. Dr. Juan de Lara

Submitted November 03, 2015.
Defended December 17, 2015.

Strüber, Daniel.
Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems.
Dissertation, Philipps-Universität Marburg (1180), 2016.

Curriculum vitae
2011, Diplom-Informatiker, Philipps-Universität Marburg.

Contents

List of Figures

List of Tables

Abstract

Model-Driven Engineering (MDE) is a software engineering paradigm that aims to increase the productivity of developers by raising the abstraction level of software development. It envisions the use of models as key artifacts during design, implementation and deployment. From the recent arrival of MDE in large-scale industrial software development – a trend we refer to as *MDE in the large* –, a set of challenges emerges: First, models are now developed at distributed locations, by teams of teams. In such highly collaborative settings, the presence of large monolithic models gives rise to certain issues, such as their proneness to editing conflicts. Second, in large-scale system development, models are created using various domain-specific modeling languages. Combining these models in a disciplined manner calls for adequate modularization mechanisms. Third, the development of models is handled systematically by expressing the involved operations using model transformation rules. Such rules are often created by *cloning*, a practice related to performance and maintainability issues.

In this thesis, we contribute three *refactoring techniques*, each aiming to tackle one of these challenges. First, we propose a technique to *split* a large monolithic model into a set of sub-models. The aim of this technique is to enable a separation of concerns within models, promoting a concern-based collaboration style: Collaborators operate on the sub-models relevant for their task at hand. Second, we suggest a technique to *encapsulate* model components by introducing modular interfaces in a set of related models. The goal of this technique is to establish modularity in these models. Third, we introduce a refactoring to *merge* a set of model transformation rules exhibiting a high degree of similarity. The aim of this technique is to improve maintainability and performance by eliminating the drawbacks associated with cloning. The refactoring creates *variability-based rules*, a novel type of rule allowing to capture variability by using annotations.

The refactoring techniques contributed in this work help to reduce the manual effort during the refactoring of models and transformation rules to a large extent. As indicated in a series of realistic case studies, the output produced by the techniques is comparable or, in the case of transformation rules, partly even *preferable* to the result of manual refactoring, yielding a promising outlook on the applicability in real-world settings.

Überblick

Model-Driven Engineering (MDE) ist ein Paradigma der Softwaretechnik, in dem es darum geht, das Abstrationsniveau und dadurch die Entwicklerproduktivität während der Softwareentwicklung zu erhöhen. Hierzu lässt man Software-Modellen eine wesentliche Rolle während Entwurf, Implementierung und Einführung eines Systems zukommen. In jüngerer Zeit wurde MDE verhäuft in industriellen Projekten hoher Komplexität und großen Umfangs eingesetzt – ein Trend, den wir als *MDE im Großen* bezeichnen. Wir betrachten drei Herausforderungen, die durch den Einsatz von MDE in solchen Szenarien entstehen: 1. Modelle werden an verteilten Standorten, durch *Teams von Teams* entwickelt. Dabei führt die Verwendung großer und unzureichend strukturierter Modelle zu erheblichen Problemen, etwa zu einer erhöhten Anfälligkeit für Editierkonflikte. 2. Große Systeme werden oft durch einen Verbund von Modellen spezifiziert, die auf domänenspezifischen Modellierungssprachen (DSMLs) basieren. Die Modelle heterogener DSMLs in systematischer Weise zu integrieren erfordert geeignete Modularisierungsstrategien. 3. Um die Entwicklung von Modellen systematisch zu beschreiben, spezifiziert man die dazu notwendigen Operationen durch Modelltransformationsregeln. Modelltransformationsregeln werden oft durch Klonierung erzeugt, also durch das Kopieren und Modifizieren vorhandener Regeln. Damit sind Nachteile für die Wartbarkeit und Performanz der erzeugten Regelsysteme verbunden.

Im Rahmen dieser Arbeit präsentieren wir drei neue Refactoring-Verfahren. Jedes dieser Verfahren zielt auf eine der benannten Herausforderungen ab: 1. Wir stellen ein Verfahren für das Aufsplitten eines monolithischen Modells in eine Menge von Teilmodellen vor. Dieses Verfahren ermöglicht die Umstrukturierung von Modellen hin zu einer *Trennung der Belange*. Bei der kollaborativen Entwicklung eines Systems können die beteiligten Entwickler somit auf Teilmodellen arbeiten, die für ihr aktuelles Arbeitspaket relevant sind. 2. Wir beschreiben

ein Verfahren zur Kapselung von Modellkomponenten durch die Einführung von Schnittstellen in einem Verbund von Modellen. Dieses Verfahren ermöglicht es, in Modellen von heterogenen DSMLs Modularität einzuführen. 3. Wir präsentieren ein Verfahren, um Modelltransformationsregeln, die gemeinsame Anteile aufweisen, zu verschmelzen. Dieses Verfahren zielt darauf ab, die Wartbarkeit und Performanz in Modelltransformationssystemen zu verbessern. Das Verfahren erstellt *variabiliätsbasierte Regeln*, ein neuartiger Typ von Regeln, in dem Variabilität anhand von Annotationen spezifiziert wird.

Die im Rahmen dieser Arbeit vorgestellten Verfahren ermöglichen es, den manuellen Aufwand während des Refactorings von Modellen und Modelltransformationsregeln erheblich zu reduzieren. In einer Reihe von realistischen Fallstudien zeigen wir, dass die erstellten Modelle und Regeln von vergleichbarer oder, im Fall von Regeln, teilweise sogar von *zu bevorzugender* Qualität gegenüber dem Ergebnis eines manuellen Refactorings sind. Wir versprechen uns daher eine hohe Relevanz der Beiträge für MDE im Kontext industrieller Softwareentwicklung.

Acknowledgements

I have been truly fortunate to work with and be inspired by a number of outstanding researchers who have made this thesis possible.

The encouragement from my advisor Gabi Taentzer made me decide to begin my PhD journey in the first place. Gabi helped me with planning the direction of my research, took great care in ensuring the soundness of the conceptual and formal aspects of my thesis, and fully supported me in following my own curiosity. Her constant support, advice, and trust were invaluable.

I wish to thank Reiko Heckel for his excellent suggestions that helped me to improve the overall presentation and many important details in my thesis and, additionally, for immediately agreeing to be my second referee. I like to thank Bernhard Seeger and Thorsten Thormählen for immediately agreeing to be members of my examination committee. I wish to thank Juan de Lara for immediately agreeing to be the third referee for the external examination of this thesis.

I wish to thank Marsha Chechik and Julia Rubin for our ongoing collaboration, a both challenging and gratifying endeavor. Even though our Skype conferences were not seldom disturbed by severe technical difficulties, Marsha's and Julia's tremendous energy and fine wits always allowed us to produce beautiful outcomes. Reading their survey paper on feature location was formative for me in many respects: First, it spawned my interest in heuristic approaches. Second, I learned that the same problem can be addressed from a virtually infinite variety of angles – an intellectually stimulating insight. Finally, no other paper boosted my sense of clarity in presentation as much as this one did.

I want to express my gratitude to the student programmers who contributed to the tools discussed in my thesis: Michael Lukaszczyk for

developing the visual component for the splitting tool, Matthias Selter for implementing a predecessor of said tool, Tim Schäfer for creating the initial tool support for composite models, Stefan Schulz for devising a graphical editor for composite models, and Jennifer Plöger for implementing several clone detection techniques for rule merging.

Harald Störrle's empirical work on models was invaluable to me as it helped me recognize the need for the refactoring techniques described in this thesis. I wish to thank Vlad Acrotaie and Harald for inviting me to join them as a collaborator in their work on usability-oriented model transformations, a very enjoyable and focused experience. From both of them, I learned a lot about clarity and style in writing.

During their stay in Marburg, I had the pleasure and honor to meet various people who were then members of Klaus Ostermann's work group. Christian Kästner gave me encouraging advice early during my studies. From him, I learned important ideas about variability, empirical methods, and research in general. The variability-oriented approach to model transformation discussed in this thesis is influenced by Christian's groundbreaking work on software product lines. Sebastian Erdweg offered me refreshing perspectives; discussing my work with him was both challenging and rewarding. Eric Walkingshaw's seminar on human factors in programming languages has deeply influenced my thoughts on our work as software engineering researchers. Finally, I enjoyed the company of Paolo Giarrusso, Tillmann Rendel, Jonathan Brachthäuser, Yufei Cai, Yi Dai and Klaus Ostermann himself, whose ability to set up a great research environment is inspiring.

I like to thank Thorsten Arendt, Stefan Jurack, Felix Rieger, Kristopher Born, Steffen Vaupel, Nebras Nassar, Dennis Priefer, Wolf Rost, Timo Kehrer, Christoph Bockisch, Manuel Wimmer, Florian Mantz, Yngwe Lamo, Claudia Ermel, Mischa Dieterle, Thomas Horstmeyer, Frank Hermann, Christian Krause, Matthias Tichy and Steffen Ziegert for discussing ideas, providing resources for this work and/or contributing to my journey in any other way. Felix, Steffen V., Thorsten, Kristopher, and my friend Sophia Bauch have proofread various chapters of my thesis and made useful suggestions for their improvement.

I would like to thank the anonymous reviewers of all published and earlier versions of my work. Many of their suggestions were helpful in strengthening the presentation of this thesis.

I wish to thank my friends and family for being kind and supportive.

Finally, my deep gratitude goes to the artists and scientists of the world who struggle to create great artifacts that contribute to our well-being.

Chapter 1

Introduction

The productivity of software developers can be substantially increased by raising the abstraction level of the languages at their disposal. As one of the most important and consistent threads running through the entire history of software engineering, the advancement in levels of abstraction has enabled a shift in focus from hardware-centric to human-oriented language paradigms. Milestones on the way to the state of the art include structured [28], object-oriented [15], and finally, declarative programming [66].

Model-Driven Engineering (MDE, [98]) has been established as a software paradigm envisioning *models* as first-class citizens in the design, implementation and deployment of software systems. MDE has already become an industrially accepted best practice in many application domains such as automotive and aerospace development, where the structure and behavior of complex systems are specified using models [65]. MDE includes, but is not limited to the practice of Model-Driven Development (MDD, [105]), the automated translation of a model towards a running software system. In Model-Driven Architecture (MDA, [104]), a well-known instantiation of MDD, models go through a series of *refinements* from abstract to concrete: A computation independent model (CIM) becomes platform-independent (PIM), platform-dependent (PDM), and finally, source code.

Catering to the different application domains and usage contexts of MDE, a large variety of **modeling languages** has evolved. Inspired by the success of general-purpose programming languages (GPLs), the Unified Modeling Language (UML, [78]) has been devised and widely

popularized as a general-purpose modeling language. In contrast to textual GPLs, UML focused on visual notations, providing 14 diagram types for purposes such as structural, behavioral, and deployment modeling. Yet, due to rapidly evolving technologies and diversifying developer groups, an increasing need for flexibility in the employed modeling languages has manifested. In recent years, we saw a rise of domain-specific modeling languages (DSMLs), providing abstractions and notations tailored to various domains and user groups. DSMLs are frequently developed using modeling platforms such as the Eclipse Modeling Framework (EMF, [106]) or Xtext [34]. In these platforms, the DSML is specified by defining a meta-model, a *model of models*. The models conforming to this meta-model are the words of the DSML.[1]

In MDE, models are not static artifacts, but routinely undergo changes. **Model transformation**, the automated modification or translation of a model, is a key enabling technology for MDE [101], pervasive in all its activities. Two major types of model transformation are distinguished: endogenous and exogenous transformation [27]. An endogenous transformation is either *in-place*, if it updates its input model, or out-place, if it produces a new model of the same modeling language. Examples include model optimizations, refinements, and refactorings in the context of quality assurance. An exogenous transformation produces a model or textual artifact of a different language. Examples include model translations, migration between DSMLs, and code generation. Model transformation is supported by a large variety of dedicated languages. An important paradigm embodied by many of these languages is graph transformation [33]: By representing them as graph patterns, model transformations can be specified in a high-level, visual manner.

Like all software artifacts, models exhibit certain quality characteristics that may change over the course of their lifecycles. **Model quality assurance** [5] is a cornerstone in MDE ensuring that the involved artifacts live up to the challenges imposed by maintenance and evolution. Based on their purpose, two main categories of techniques, tools, and processes are distinguished: *Analytical* quality assurance techniques allow to evaluate the quality of a given software model. *Constructive* quality assurance techniques enable to improve the quality of the model. A key example for a constructive technique is *model refactoring*, improving the structure of a model without changing its behavior [12].

A recent trend is the introduction of MDE in large-scale industrial settings to create and maintain systems of substantial size [108, 109, 60].

[1] In the case of Xtext, a user-specified grammar is transformed into a meta-model.

In the scope of this work, we term this trend **MDE in the large**. While MDE is a promising approach to tame the increased complexity involved in these settings, state-of-the-art MDE techniques and tools were frequently designed for and tested in scenarios of smaller scale, giving rise to scalability issues when the involved systems and models grow. In this work, we aim to tackle several of these challenges.

1.1 Challenges and Research Directions

Kolovos et al. [60] propose a research roadmap to investigate the challenges arising from the application of MDE in industrial large-scale scenarios. They identify three major categories: Challenges to *collaborative modeling*, challenges to *domain-specific modeling languages* (DSMLs), and challenges to *model queries and transformations*[2]. The three main research directions of this thesis span over these categories. In this section, we outline our research direction within each category.

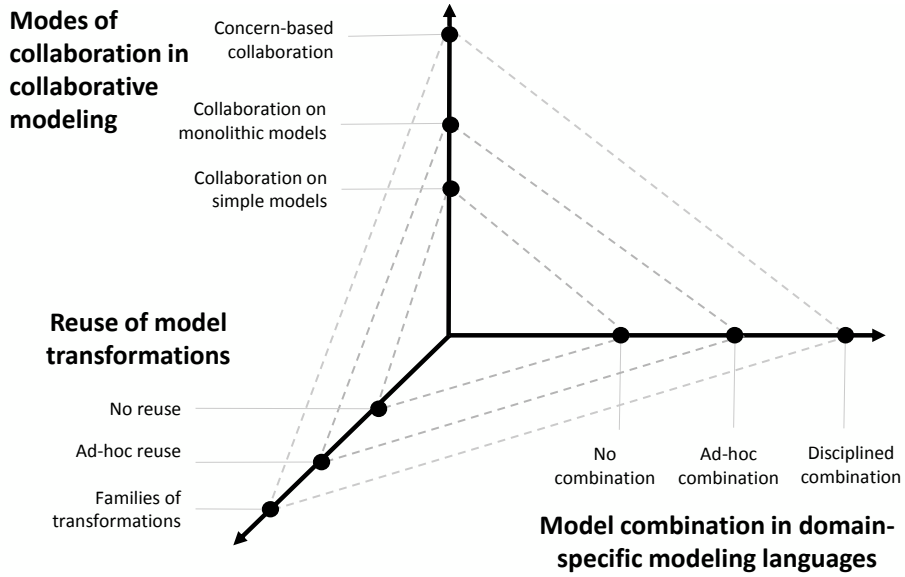

Figure 1.1: Research directions of this thesis.

In the overview shown in Fig. 1.1, each axis represents one of the research directions: *modes of collaboration in collaborative modeling*, *model combination in DSMLs*, and *reuse in model transformations*. Dashed trian-

[2] A fourth category mentioned in [60] is *scalable model persistence*.

gles indicate the *achieved* and *targeted* state of the art: The small triangle denotes traditional MDE as applied in small-scale scenarios. The medium-sized triangle delineates the current state of applying MDE in large-scale scenarios. The large triangle indicates our guiding vision for *MDE in the large*. The challenges we aim to address are represented by the vertices of the medium-sized triangle, i.e., those related to *monolithic models*, *ad-hoc model combination* and *ad-hoc transformation reuse*. Our guiding principles to tackle these challenges are represented by the vertices of the large triangle, i.e., *concern-based collaboration*, *disciplined combination*, and dedicated support for *families of model transformations*.

In the rest of this section, to introduce each direction, we revisit its general context as per [60] and discuss the addressed challenges in detail.

1.1.1 Collaborative Modeling

The first research direction deals with *"enabling large teams of modellers to construct and refine large models in a collaborative manner"* ([60], p.1). In general, the collaborative development of a software system is a well-understood task, supported by powerful collaboration and versioning tools [67]. Yet, when applying these established tools in a MDE context, developers face several impediments: Unlike in code repositories which are divided into extensive hierarchies of files that are viewed and modified independently, models tend to be large monolithic artifacts [109]. Moreover, the established text-based versioning tools are unsuited to account for the semantic differences between model revisions. Finally, collaboration tools must account for the large variety of domain-specific modeling languages (DSMLs) by providing customization capabilities for the DSML at hand with reasonable effort .

In this work, we focus on challenges imposed by *monolithic models*. We use the term monolithic (*"constituting a massive undifferentiated and often rigid whole"*, [72]) to refer to a model that is large (*massive*) and lacks an adequate internal organization to reflect its functional concerns *(undifferentiated)*. Several situations can give rise to monolithic models: A monolithic model may be the result of an insufficient initial design effort. Furthermore, a model may exhibit an organization that was found adequate at an earlier point in time, but has turned ineffective as the model has evolved. Finally, monolithic models may occur in scenarios where the model is created automatically, for instance by reverse engineering a design model from a code base.

Monolithic models are related to the following issues:

Proneness to editing conflicts: The state-of-the-art tool used for collaborative software development are version control systems (VCS), such as Git [67] or Subversion [22]. In a VCS, each developer owns a local copy of the project, performs changes on that copy and continuously commits the modifications to a central repository. The central repository maintains an official, stable state of the project. The state-of-the-art VCS operate in a file-based manner. When they are used to collaborate on a single-file monolithic model, frequent *editing conflicts* are a direct consequence. Some editing conflicts are easy to reconcile automatically. However, it is not uncommon that developers are forced to resolve conflicts manually, an error-prone and time-consuming activity.

Scattering and tangling: In visual modeling, users inspect and modify models by means of diagrams. Diagrams provide views on selected portions of the model at hand. If a model exhibits an inadequate organization, it is likely that the associated diagrams do so as well. In particular, this is the case if the diagram is generated automatically [69] rather than manually devised. Due to an inadequate organization, related elements may be *scattered* among several diagrams, while diagrams may be *tangled* with unrelated elements. Understanding diagrams affected by scattering and tangling may impose a high mental effort on developers, increasing maintenance time and impairing productivity.

Diagram size: As models grow in size and complexity, so do their diagrams, giving rise to issues related to diagram size. While visual representations are commonly related to advantages such as their appeal to certain cognitive facilities of the human mind [77], a diagram might become so large that its navigation becomes tedious, achieving the opposite effect as intended. In a study on the effect of diagram size to the performance of expert and novice modelers, Störrle determined a maximum size of "safe" diagrams, ranging in the magnitude of 50 elements [110]. A diagram size exceeding this boundary is linked to diminished developer performance.

1.1.2 Domain-Specific Modeling Languages

The second research direction is concerned with *"being able to construct large models and domain specific languages in a systematic manner"* ([60], p.1). First, to account for the various aspects of a complex system, its

specification requires multiple purpose-tailored DSMLs rather than a single monolithic one. Combining the models in heterogeneous DSMLs requires adequate composition and modularization concepts on the level of meta-models and models. Moreover, a DSML might be used to create large models. To query and transform these models efficiently, it might be recommended to extend the DSML with facilities for the management of large models. Finally, combing heterogeneous DSMLs poses challenges to their syntactical representations.

In this thesis, we focus on the *disciplined combination of DSMLs and their models*. The state-of-the-art modeling platforms used to develop DSMLs, such as EMF and Xtext, provide basic means to combine models by allowing *remote references*, links between models. While this mechanism is sufficient for spreading a model over a set of files, it promotes an undisciplined combination style, neglecting important engineering principles such as encapsulation and information hiding.

The following challenges are related to model combination in DSMLs:

Lack of systematic development methods: As there exists a shortage of systematic development methods, DSMLs and their conforming models are frequently developed in an ad-hoc way. Pointing out their crucial relevance for MDE, Kolovos et al. emphasize that these artifacts *"should be engineered using sound principles and methods"* [60]. The lack of adequate modularization mechanisms is a crucial obstacle to efforts in this direction: It gives rise to an undisciplined combination of models, neglecting important and generally acknowledged engineering principles such as *high cohesion* and *low coupling* [107].

Difficult reuse: Instead of building software components from scratch, there has been a recent shift towards the construction of systems from reusable components [16]. It is promising to view the models created using DSMLs as reusable components: Encapsulating the design models of selected system concerns, each represented by a dedicated DSML, into reusable artifacts may help to avoid redundant modeling effort in subsequently developed systems. To identify components suitable to the task at hand, the developers of these systems must be able to identify the abstract behavior of components easily. Yet modeling platforms do not provide dedicated means to separate a model's behavior specification from its implementation. In this respect, modeling platforms lag behind the capabilities of established programming languages, where *information hiding* [81], the separation of a module's functional interface from is implementation, has been established for decades.

Information exposure: Models created using DSMLs are frequently exchanged among multiple independent actors, such as enterprises or freelance developers. Yet, due to business, security, or legal reasons, it may be undesirable or even illegal to make the entire models available between all stakeholders. Consider a company maintaining a corporate data model that includes detailed information on all of its products. While displaying the general product information in a suited form to customers might be highly desirable, the company has no interest in exposing the product building plans. Thus, concepts to restrict the visibility of selected portions of a model are required. Such concepts are not included in state-of-the-art modeling platforms.

1.1.3 Model Transformations

The third research direction deals with *"advancing the state of the art in model querying and transformations tools so that they can cope with large models"* ([60], p.1). The task performed by model transformation tools is the application of one or multiple transformation rules on an input model. As a result, the input model is updated or a new model is created. There are two dimensions of scalability of model transformation tools: First, the model under transformation might be large. Second, the considered rule set might be large.[3]

In contrast to various approaches dealing with performance optimizations for large input models, as surveyed in [60], we consider the scalability issues involving large rule sets. Specifically, we focus on *families of model transformations*: variability-intensive transformation systems incorporating a substantial amount of redundancies between individual rules. Such rules are often created by copying and modifying a seed rule in order to create multiple similar variants – a mechanism referred to as *cloning* or *ad-hoc reuse*.

We consider the following challenges related to families of model transformations:

Maintainability: In the traditional view shared by many software engineering professionals, the practice of cloning is considered with suspi-

[3]It is worth pointing out that large transformation systems are large models, too: Model transformation languages are essentially domain-specific modeling languages for the domain of *model transformation*. Consequently, *model transformation rule* is a type of model.

cion [91]. Fowler [36] deems clones a severe kind of bad smell, related to drawbacks for the maintainability of the involved artifacts. For instance, if a bug is found, all instances of the clone must be updated to remove the bug. While programming languages equip developers with adequate concepts to avoid cloning and the related drawbacks, e.g., *inheritance* or *subroutines*, the development of reuse concepts is still in its preliminary stages in various model transformation languages [63]. In these languages, developers had no alternative to cloning until recently, leaving ad-hoc reuse the most frequently applied mechanism, an impediment to the maintainability of transformation systems.

Performance: Model transformations are applied in various execution modes, such as change-driven [86], incremental [44], or streaming [23] transformation. *Batch transformation* is one of the default modes, being frequently applied in model translations, simulations, and refactoring suites. In batch mode, each rule in the transformation system is considered non-deterministically, as long as one of the rules is applicable. When executed in batch mode, a transformation system comprising many similar rules may show significant performance bottlenecks: Each rule variant increases the computational effort of the transformation system. The larger the rule set becomes, he harder it is to handle efficiently, possibly rendering the entire transformation infeasible. Blouin et al. [14] report on a case study where the transformation engine was not able to execute a large transformation system with 250 rules.

1.2 Contributions

In this previous section, we have outlined three categories of scalability challenges related to collaborative modeling, domain-specific modeling languages (DSMLs), and model transformations. The first three contributions of this thesis are three refactoring techniques, each addressing one of these categories.

1. **Model splitting**. We propose a refactoring to *split* a monolithic model into a set of interconnected sub-models. The aim of this technique is to enable a separation of concerns within models, promoting a concern-based collaboration style: Each collaborator is assigned the sub-model for the task at hand. To derive its output, the technique uses information retrieval and model crawling techniques. It distinguishes itself from earlier split-

ting approaches by reflecting the stakeholders' intention more closely, taking into account textual descriptions of the desired sub-models. The textual descriptions may stem from requirement documents, existing documentation artifacts, or interview records. To give rise for a practical usage of the technique, it is not required to provide a complete list of textual descriptions upfront: The technique supports an incremental use, allowing to discover new sub-models successively. We applied the technique on a set of real-life class models, detecting a promising outcome in terms of precision and recall.

2. **Component encapsulation**. We suggest a refactoring allowing to *derive modular interfaces* for models created using DSMLs. By creating these interfaces, the technique produces *composite models* [51], an extension of standard modeling frameworks aiming to establish a disciplined combination of models: A composite model is a model with a set of export and import interfaces. Portions of the model are assigned to an export or import interface, declaring the portion to be exported to or imported from the environment. We reiterate and extend the existing formalization of composite models. As a main result, we obtain that the introduction of metamodel interfaces can be propagated to introduce interfaces in the conforming models.

3. **Rule merging**. We introduce a refactoring to *merge* a set of model transformation rules exhibiting a high degree of similarity. To eliminate the maintainability drawbacks and performance bottleneck, the proposed refactoring creates variability-based rules, explicating commonalities and differences between the rules by annotating portions with variability information. To construct the rules, we employ state-of-the-art clone detection and clustering techniques and introduce a novel merge construction algorithm. We applied the refactoring in three case studies, witnessing a decrease of clone portions together with significant performance savings.

Moreover, the third contribution *rule merging* gives rise to the introduction of a novel approach to model transformation that provides dedicated support for families of transformations. The key idea is to explicate the commonalities and differences of a set of rule variants and to use this information during the application of the variants. We put this approach forward as a fourth contribution of this thesis.

4. **Variability-based model transformation**. We propose a novel approach to model transformation based on *variability-based (VB) rules*. VB rules are a compact way to represent multiple rule variants in a model transformation system. Portions of a VB rule are annotated with presence conditions to indicate them as being part of one or several rule variants. During the application of rules, a significant performance speed-up can be achieved by considering the base rule – the portion without annotations – first. We give a formal definition of VB model transformation based on algebraic graph transformation, proving that VB rule application yields the same results as applying the corresponding rule variants individually. We show that VB model transformation can improve the compactness and performance of the involved rules considerably.

1.3 Methodology

The research methodology guiding this thesis is informed by a taxonomy of software engineering research proposed by Shaw [102]. Based on the intended research results – the *"tangible products"* created as output of the research – Shaw distinguishes five ways to approach a software engineering problem. In this taxonomy, possible results are *qualitative/descriptive models*, *techniques*, *systems*, *empirical predictive models*, and *analytic models*.

Each of the four contributions put forward by this thesis comprises a *technique* supported by an *analytic model*:

> "**Technique.** *Invent new ways to do some tasks, including procedures and implementation techniques. Develop a technique to choose among alternatives.*
> **Analytic model.** *Develop structural (quantitative or symbolic) models that permit formal analysis." ([102], p.660)*

Our Contributions 1–3 provide novel automated techniques to facilitate tasks previously imposing a considerable manual effort on developers. Contribution 4 includes a new technique to improve the efficiency of an already fully automated technique. In Contributions 1–4, formal models based on algebraic graph transformations are used to specify the input and output of each technique and argue for its correctness.

Moreover, Shaw distinguishes five kinds of validation techniques to show that a research result satisfies the requirements posed by the motivating problem: *persuasion*, *analysis*, *implementation*, *evaluation*, and *experience*.

In this work, we apply all of these techniques. Throughout the thesis, *persuasion* is used to motivate our design choices and rationales. In Contributions 1–4, we give formal *analytic proofs* to argue for correctness. For Contributions 1–4, we provide and discuss *implementations*. Contributions 1, 3 and 4 include empirical *evaluations* based on measurements of quantities relevant to address their motivating problems. Contributions 1 and 2 are validated using narrative *demonstrations* exemplifying potential experiences of applying the approaches.

1.4 Outline

The remainder of this thesis is structured as follows.

- In chapter 2, we introduce *model splitting*. We demonstrate the technique by example, give formal definitions of sub-models and model splitting, outline the central algorithm, discuss tool support, and evaluate the approach by giving qualitative and quantitative evidence of its usefulness.
- In chapter 3, we propose *component encapsulation*. We revisit the existing formal foundation of composite modeling, demonstrate the novel technique by example and give a formal proof of its soundness. To argue for its usefulness, we embed it into the larger context of a collaborative modeling process. We exemplify this process using a case demonstration.
- In chapter 4, we put forward *rule merging*. We introduce variability-based rules and their application formally and by example. We then demonstrate the novel refactoring technique, underpinning its concepts with formal definitions and a correctness proof. We evaluate its impact on the efficiency of model transformation systems in two realistic case studies.
- In chapter 5, we summarize and conclude this thesis. We give an outline on possible future research directions.

Chapter 2

Model Splitting

This chapter shares material with the FASE'14 paper "Splitting Models Using Information Retrieval and Model Crawling Techniques" [117] and the BigMDE'14 paper "Tool Support for Model Splitting Using Information Retrieval and Model Crawling Techniques" [112].

In this chapter, we consider the problem of splitting a model into sub-models to facilitate developer independence. We propose an automated technique that creates purpose-tailored decompositions by leveraging domain knowledge provided in the form of textual descriptions. The technique is based on information retrieval and model crawling techniques. We embed it in an approach that assists users in incrementally discovering the set of desired sub-models. We demonstrate the effectiveness of our approach on a set of real-life case studies, involving UML class models and EMF meta-models.

2.1 Introduction

Together with the increased popularity of modeling, models of practical use grow in size and complexity to the point where large monolithic models are difficult to comprehend and maintain. There is a need to *split* such large models into a set of dependent modules (a.k.a. *sub-models*), increasing the overall comprehensibility and allowing multiple distributed teams to focus on each sub-model separately. Earlier works, e.g., [58], suggest approaches for splitting models based on an analysis of strongly connected components, largely ignoring the semantics of

the split and the user intention for performing it. Other works require to fully annotate the model upfront [37], which allows reflecting the user intention precisely, but imposes a considerable effort on the user.

In this work, we propose an alternative, heuristic approach that allows splitting a model along functional concerns. The approach assumes that these functional concerns are explicitly specified by the user using *natural-language descriptions*. These descriptions may be retrieved from existing requirements documents, system documentation, or transcripts of developer interviews. The proposed approach is inspired by feature location techniques [30, 93], which discover implementation artifacts corresponding to a particular, user-defined, functionality.

In the core of our approach is an automated technique that employs *information retrieval (IR)* and *model crawling*. Given an input model and a set of its sub-model descriptions, the technique assigns each element to one of the specified sub-models, effectively producing a partitioning. The technique is applicable to any model for which a split results in sub-models that satisfy the well-formedness constraints of the original one, e.g., UML Class models, EMF models and MOF-based metamodels.

The user can decide whether the list of sub-models describes a *complete* or a *partial* split of the input model. In the former case, each input model element is assigned to exactly one sub-model, like in the example in Fig. 2.2, where the three sub-models "cover" the entire input model. In the latter case, when the complete set of the desired sub-models is unknown upfront, the technique produces assignments to known sub-models only. The remaining elements are placed in a sub-model called "rest". The user can inspect the "rest" sub-model in order to discover remaining sub-models in an incremental and iterative fashion, until the desired level of completeness is achieved.

Considering the challenges outlined in Sec. 1.1.1, this approach allows us to address the issues arising from the use of monolithic models. We briefly revisit these issues and discuss how the technique helps to alleviate them.

- *Proneness to editing conflicts* is an issue of monolithic models in state-of-the-art versioning control systems. As these systems operate in a file-based manner, frequent editing conflicts are expected if two or more parties collaborate on the same monolithic model at the same time, leading to a manual effort to resolve these

conflicts. Splitting a model in a set of sub-models, each being persisted in a separate file, helps to reduce this issue.

- *Scattering and tangling* refers to the situation that a diagram may inherit an inadequate organization of its associated model: Functionally related elements might be scattered across several diagrams, while a diagram might be tangled with functionally unrelated elements. By reorganizing the model into sub-models based on functional concerns, the proposed technique helps to establish models and diagrams that do not suffer from this issue.
- *Diagram size* becomes an issue when a diagram representation of a model becomes too large to be navigated and understood efficiently [110]. Splitting a model into multiple smaller sub-models helps to reduce the size of the involved models and the associated diagrams. Additionally, the technique can be used to split a model into disjoint sets of model elements that are displayed in several diagrams, without changing the actual organization of the model.

We make the following contributions:

- We describe an automated model splitting technique that combines information retrieval and model crawling.
- We propose a computer-supported iterative process for model splitting.
- We introduce a tool implementing the automated technique, discussing its design goals and implementation.
- We evaluate our approach on a set of benchmark case studies, including real-life UML and EMF models. Our results demonstrate that the proposed approach achieves high accuracy compared to the manually produced results and is able to assist the user in the iterative discovery of the desired sub-models.

The rest of the chapter is structured as follows. Sec. 2.2 gives a high-level overview of our approach. We describe the necessary preliminaries in Sec. 2.3, present a formal framework underpinning the approach in Sec. 2.4 and introduce its instantiation in the form of an automated algorithm in Sec. 2.5. We discuss tool support in Sec. 2.6. We report on the results of evaluating our approach in Sec. 2.7. We put our contribution in the context of related work in Sec. 2.8 and conclude in Sec. 2.9.

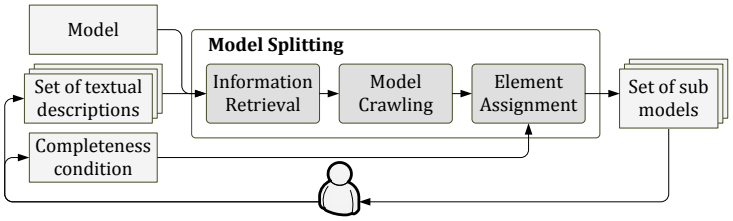

Figure 2.1: An overview of model splitting.

2.2 Overview

A high-level overview of the proposed approach is given in Fig. 2.1. The user provides as input a model that requires splitting, a set of textual descriptions of the desired sub-models, and the completeness configuration parameter that declares whether this set of sub-models is complete or partial. For the example in Fig. 2.2, the complete set would contain descriptions of all three sub-models – medical team, physical structure, and patient care, while a partial set would contain only some of these descriptions.

2.2.1 Example

Consider the class model of a hospital system (HSM) [84, p. 125] shown in Fig. 2.2. It describes the organization of the hospital in terms of its medical team (elements #1-7), physical structure (elements #8-17), and patient care (elements #18-29). Each of these concepts corresponds to a desired sub-model, visually encircled by a dashed line for presentation purposes. The goal of our work is to assist the user in determining elements that comprise each sub-model. The user describes the desired sub-models using natural-language text, e.g., using parts of the system documentation.

For example, the medical team sub-model in Fig. 2.2 is described in [84]. A fragment of the description is: "*Nurses are affiliated with a single ward, while physicians and technicians can be affiliated with several different wards. All personnel have access to a calendar detailing the hours that they need to be present at the various wards. Nurses record physicians' decisions. These are written on paper and handed to an administrative assistant to enter. The administrative assistant needs to figure out who needs to be at a particular procedure before they enter it in the system.*" The technique uses such de-

scriptions in order to map model elements to desired sub-models. The labels for the sub-models, e.g., "Medical Team", are assigned manually.

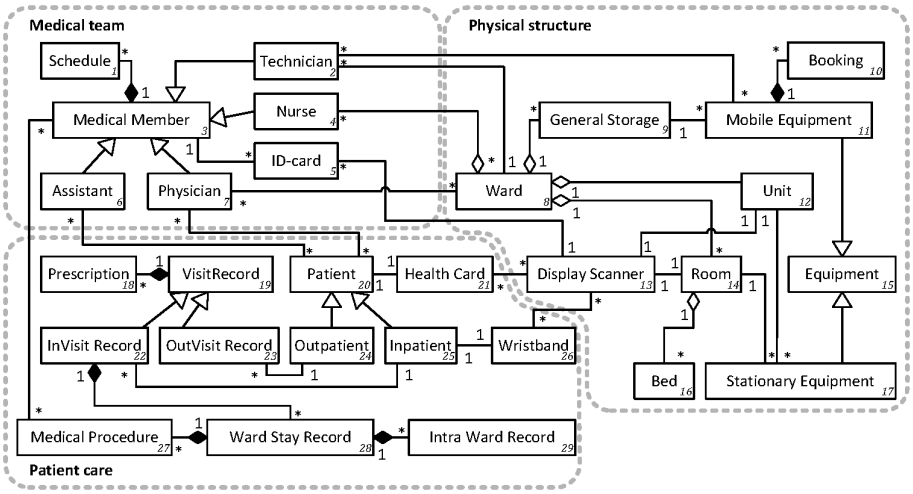

Figure 2.2: A UML class model of a hospital system.

2.2.2 Automated Technique

In the core of the proposed approach is an automated technique that *scores* the model elements wrt. their relevance to each of the desired sub-models. The scoring is done in two phases. The first one is based on *Information Retrieval (IR)* and uses sub-model descriptions: it builds a textual query for each model element, e.g., based on its name, measures its relevance to each of the descriptions and identifies those elements that are deemed to be most relevant for each of the descriptions.

The identified elements are used as *seeds* for the second phase, *Model Crawling*. In this phase, structural relationships between model elements are explored in order to identify additional relevant elements that were missed by the IR phase. The additional elements are scored based on their structural proximity to the already scored elements. In HSM, when identifying elements relevant to the medical team sub-model using the description fragment shown in Sec. 2.2, the IR phase correctly identifies elements #2,4,6,7 as seeds. It misses element #3 though, which is assigned a high score in the first iteration of crawling as it is closely related to the seeds. Once element #3 is scored, it im-

pacts the scoring of elements identified during later iterations of crawling. Eventually, each model element's relevance to each sub-model is scored.

The third phase, *Element Assignment*, assigns elements to sub-models based on their score. If a complete set of sub-models is given, each element is assigned to a sub-model for which it has the highest score[1]. In this case, the assignment results in a model partition. If a partial set of sub-models in given as an input, some model elements might not belong to any of these sub-models. Hence, we apply a threshold-based approach and assign elements to sub-models only if their scores are above a certain threshold.

2.2.3 Process

A partial set of sub-model descriptions can be further refined in an *iterative* manner, by focusing user attention on the set of remaining elements – those that were not assigned to any of the input sub-models. Additional sub-models identified by the user, as well as the completeness parameter assessing the user's satisfaction with the set of known sub-models are used as input to the next iteration of the algorithm, until the desired level of completeness is achieved.

Clearly, as additional sub-models are identified, element assignments might change. For example, when only the description of the medical team sub-model is used during a split, element #8 is assigned to that sub-model due to the high similarity between its name and the description: the term `ward` is used in the description multiple times. Yet, when the same input model is split w.r.t. the sub-model descriptions of both the medical team and the physical structure, this element is placed in the latter sub-model: Both its IR score and its structural relevance to that sub-model are higher. In fact, the more detailed information about sub-models and their description is given, the more accurate the results produced by our technique become, as we demonstrate in Sec. 4.8.

[1] An element that has the highest score for two or more sub-models is assigned to one of them randomly.

2.3 Preliminaries

In this section, we introduce the preliminaries to the proposed approach. We discuss a selection of relevant *information retrieval* and *feature location techniques*.

2.3.1 Information Retrieval Techniques

Term Frequency - Inverse Document Frequency Metric (*TF-IDF*) [85]. *Tf-idf* is a statistical measure often used by IR techniques to evaluate how important a term is to a specific document in the context of a set of documents (*corpus*). It is calculated by combining two metrics: *term frequency* and *inverse document frequency*. The first one measures the relevance of a specific document d to a term t ($tf(t, d)$) by calculating the number of occurrences of t in d. Intuitively, the more frequently a term occurs in the document, the more relevant the document is. For the HSM example where documents are descriptions of the desired sub-models, the term `nurse` appears in the description d of the medical team sub-model in Sec. 2.2 twice, so $tf(\texttt{nurse}, d) = 2$.

The drawback of term frequency is that uninformative terms appearing throughout the set D of all documents can distract from less frequent, but relevant, terms. Intuitively, the more documents include a term, the less this term discriminates between documents. The *inverse document frequency*, *idf(t)*, is calculated as follows: $idf(t) = \log(\frac{|D|}{|\{d \in D \mid t \in d\}|})$. This metric is higher for terms that are included in a smaller number of documents.

The total *tf-idf* score for a term t and a document d is calculated by multiplying its *tf* and *idf* scores: $tf\text{-}idf(t, d) = tf(t, d) \times idf(t)$. In our example, since the term `nurse` appears neither in the description of the physical structure nor in patient care, $idf(\texttt{nurse}) = \log(\frac{3}{1}) = 0.47$ and $tf\text{-}idf(\texttt{nurse}, d) = 2 \times 0.47 = 0.94$.

Given a query which contains multiple terms, the *tf-idf* score of a document w.r.t. the query is commonly calculated by adding the *tf-idf* scores of all query terms. For example, the *tf-idf* score of the query "*medical member*" w.r.t. the description of the medical team sub-model is $0 + 0 = 0$ as none of the terms appear in the description and thus their *tf* score is 0. The latent semantic analysis (LSA) technique described below is used to "normalize" scores produced by *tf-idf*.

Latent Semantic Analysis (LSA) [64]. LSA is an automatic mathematical/statistical technique that analyzes the relationships between queries and passages in large bodies of text. It constructs vector representations of both a user query and a corpus of text documents by encoding them as a *term-by-document co-occurrence matrix*. It is a sparse matrix whose rows correspond to terms and whose columns correspond to documents and the query. The weighing of the elements of the matrix is typically done using the *tf-idf* metric.

Vector representations of the documents and the query are obtained by normalizing and decomposing the term-by-document co-occurrence matrix using a matrix factorization technique called *singular value decomposition* [64]. The similarity between a document and a query is then measured by calculating the cosine between their corresponding vectors, yielding a value between 0 and 1. The similarity increases as the vectors point "in the same general direction", i.e., as more terms are shared between the documents. For example, the queries "*assistant*", "*nurse*" and "*physician*" result in the highest score w.r.t. the description of the medical team sub-model. Intuitively, this happens because all these queries only have a single term, and each of the terms has the highest *tf-idf* score w.r.t. the description. The query "*medical member*" results in the lowest score: none of the terms comprising that query appear in the description.

2.3.2 Feature Location Techniques

Feature location techniques aim at locating pieces of code that implement a specific program functionality, a.k.a. a *feature*. A number of feature location techniques for code have been proposed and extensively studied in the literature [30, 93]. The techniques are based on static or dynamic program analysis, IR, change set analysis, or some combination of the above.

While the IR phase of our technique is fairly standard and is used by several existing feature location techniques, e.g., SNIAFL [136], our model crawling phase is heavily inspired by a code crawling approach proposed by Suade [90]. Suade leverages static program analysis to find elements that are related to an initial *set of interest* provided by the user – a set of functions and data fields that the user considers relevant to the feature of interest. Given that set, the system explores the *program dependence graph* whose nodes are functions or data fields and edges are function calls or data access links, to find all neighbors of

the elements in the set of interest. The discovered neighbors are scored based on their *specificity* – an element is specific if it relates to few other elements, and *reinforcement* – an element is reinforced if it is related to other elements of interest. The set of all elements related to those in the initial set of interest is scored and returned to the user as a sorted *suggestion set*. The user browses the result, adds additional elements to the set of interest and reiterates.

Our modifications to this algorithm, including those that allow it to operate on models rather than code and automatically perform multiple iterations until a "fixed point" is achieved, are described in Sec. 2.5.

2.4 Framework

In this section, we describe a formal framework for model splitting: We specify the input and output of splitting – models and their decomposition to sub-models – on the basis of algebraic graph transformation concepts. As long as this specification is satisfied, the concrete instantiation of splitting may vary. We later describe our instantiation based on information retrieval and model crawling techniques.

Graphs, Typed Graphs, and Graph Splitting

The formal foundation of this thesis is Algebraic Graph Transformation [33], a key paradigm in formal reasoning about models. Graphs are a suited concept to capture the structure of visual models: Model elements can be considered as nodes, while their relationships – including distinguished ones, such as *typing* and *containment* – can be considered as edges.

According to these considerations, the input model in model splitting is essentially a *graph*. The meta-model that was used to create the input model is a distinguished graph called *type graph*. The typing of model elements is expressed using a structure-preserving mapping between these graphs, called *graph morphism*. The existence of a total morphism between the graph and the type graph renders the graph a *typed graph*.

Definition 1 (Graph). *A* graph $G = (G_N, G_E, src_G, trg_G)$ *consists of a set* G_N *of nodes, a set* G_E *of edges, and source and target functions,* $src_G, trg_G : G_E \to G_N$.

Definition 2 (Total (Partial) graph morphism)**.** *Given two graphs G and H, a pair of total (partial) functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a total (partial) graph morphism $f : G \to H$, a.k.a.* morphism*, if it fulfills the following properties: (1) $f_N \circ src_G = src_H \circ f_E$ and (2) $f_N \circ trg_G = trg_H \circ f_E$. If both functions f_N and f_E are injective, f is called* injective*. If both functions f_N and f_E are inclusions, f is called* inclusion*.*

Definition 3 (Typed Graph)**.** *A* typed graph *over a distinguished graph TG, called type graph, is a tuple $(G, type)$ with $type : G \to TG$. Nodes $x, y \in G_N$ are* related*, written related(x, y), iff $\exists e \in G_E$ s.t. either $src_G(e) = x \wedge trg_G(e) = y$, or $src_G(e) = y \wedge trg_G(e) = x$. If type(e) = t, we further say that x and y are* related through t*, written related$_t(x, y)$.*

For example, the HSM in Fig. 2.2 can be considered a graph with elements #1-29 as nodes and their relationships as edges. Element #7 is related to elements #3, #8 and #20. Assuming that this model conforms to a simple meta-model for class models, the corresponding type graph may have a node named *class* being simultaneously source and target to three edges named *association*, *composition*, and *inheritance*.

Definition 4 (Sub-graph)**.** *Let a typed graph $(G, type)$ be given. $(S, type_S)$ is* a sub-graph of *G, written $S \subseteq G$, iff $S_N \subseteq G_N$, $S_E \subseteq G_E$, $src_S = src_{|S_E}$ with $src_S(S_E) \subseteq S_N$, $tgt_S = tgt_{|S_E}$ and $type_S = type_{|S}$*[2]*.*

That is, while sources of all of a sub-graph's relationship are elements within the model, it does not have to be true about the targets. For example, each dashed frame in the example in Fig. 2.2 denotes a valid sub-graph of HSM. All elements inside each frame form the element set of the corresponding sub-graph. There are two types of relationships between these elements: those with the source and the target within the sub-graph, e.g., all inheritance relations within the medical team sub-graph, and those spanning two different sub-graphs (often, these are association relationships).

Definition 5 (Constraint)**.** *A constraint φ is an atomic statement about a graph. A satisfaction relation, written $\vdash$, determines whether a given graph G satisfies φ, written $G \vdash \varphi$.*

For instance, HSM satisfies the constraint "there are less than 42 nodes".

Definition 6 (Splittability)**.** *A graph satisfying a constraint φ is* φ-splittable *iff every sub-graph of G satisfies φ.*

[2]For a function f: $G \to H$ with $S \subseteq G$, $f_{|S}$: $S \to H$ denotes the restriction of f to S.

Considering the basic constraints of class models (without packages), all class models are splittable: We can take any set of classes with their relationships and obtain a class model. Class models with packages have a constraint "every class belongs to exactly one package". To make them splittable, we may remove the packages before splitting and then reintroduce them after the splitting is performed, in a new form.

Definition 7 (Graph split)**.** *Given a typed graph* $(G, type)$*, a* graph split $Split(G) = \{S | S \subseteq G\}$ *is a set of sub-graphs s.t.* $\forall S_1, S_2 \in Split(M) : (S_1 \neq S_2) \Rightarrow (S_{1_N} \cap S_{2_N} = \emptyset)$*.*

By Def. 4, if $\bigcup_{S \in Split(G)} S_N = G_N$, then $\bigcup_{S \in Split(M)} S_E = G_E$. A split of HSM, consisting of three sub-graphs, is shown in Fig. 2.2.

2.5 Instantiation

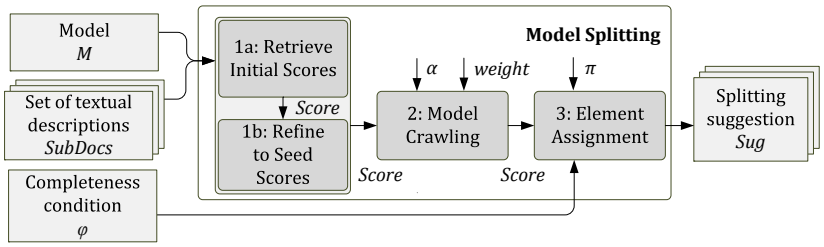

Figure 2.3: An outline of the proposed splitting algorithm.

In this section, we describe our instantiation of model splitting. Our instantiation is based on an algorithm that incorporates Latent Semantic Analysis (LSA) and model crawling, being outlined in Fig. 2.3. Its inputs are a model M, a set of textual descriptions of desired sub-models *SubDocs*, and a completeness condition ϕ which is *true* if *SubDocs* represents a desired partitioning of M and *false* if this set is partial. The splitting suggestion Sug represents a graph split (Def. 7). To ensure that the sub-models in Sug are valid, M needs to be splittable as per Def. 6 w.r.t. to the constraints of its modeling language.

The algorithm is based on scoring the relevance of model elements for each target sub-model (steps 1-2), and then assigning each element to the most relevant sub-model (step 3). The relevance scoring is done by first applying LSA and then using the scored sets of elements as

seeds for model crawling. The latter scores the relevance of all model elements w.r.t. specificity, reinforcement, and cohesiveness of their relations. The algorithm also uses parameters w, α and π which can be user adjusted for the models being analyzed. Our experience adjusting them for class model splitting is discussed in Sec. 2.7.

Step 1a: Retrieve Initial Scores Using LSA. The user provides the input model M and natural-language sub-model descriptions *SubDocs* as unrelated artifacts. They need to be preprocessed before LSA can establish connections between them. *SubDocs* are textual and can be used as input documents directly. Textual queries are retrieved from elements of M by extracting a description – in class models, the element's name. LSA then scores the relevance of each sub-model description to each model element description as described in Sec. 2.3.1. The resulting scores are stored in *Score*, a data structure that maintains a map from *(sub-model number, element)* pairs to scores between 0 and 1.

Step 1b: Refine initial scores to seed scores. Some scored elements may not be suited as starting points for model crawling. If a model element description occurred in many different sub-model descriptions, its score might be too low. In this step, we use the technique proposed in [136] which involves inspecting the scores in descending order. The first gap greater than the previous is determined to be a *separation point*; all scores below it are discarded. The remaining scores are normalized for each sub-model to take the entire (0, 1] range.

Step 2: Model crawling. The aim of model crawling is to score the relevance of each model element for each target sub-model. Model crawling is a breadth-first search: beginning with a set of seeds, it scores the neighbors of the seeds, then the neighbors' neighbors, et cetera.

This step is outlined in Fig. 1: An exhaustive crawl is performed for each target sub-model. While there exists a scored element with unscored neighbors, we determine for each of these elements x and each relationship type t the set of directly related elements, calling it *OneHop* (lines 5-7). To score each unscored element in *OneHop*, the *TwoHop* set comprising *their* related elements is obtained (lines 8-9). The score is computed at line 10 as a product of x's score, a fraction quantifying specificity and reinforcement, and a type-specific weighting factor $w(t)$:

$$\text{calculateScore}(j,y) = \text{Score}(j,x) * \frac{|\mathit{TwoHop} \cap \mathit{Scored}|}{|\mathit{OneHop}| * |\mathit{TwoHop}|} * w(t))^{\alpha}$$

Algorithm 1 Crawl model.

Input: M: Model conforming to meta-model MM
Input: *SubDocs*: A set of i target sub-model descriptions
Input: *Score :* $((1..i) \times M_N) \rightarrow [0,1]$: Map of (*sub-model number, element*) pairs to scores
Constant: $w : MM_E \rightarrow (0,1]$: Weighting function for relationship types
Constant: $\alpha \in (0,1]$: Calibration parameter
Output: *Score :* $((1..i) \times E) \rightarrow [0,1]$

```
1  function CRAWLMODEL(M, SubDocs, Score)
2      for each 1 ≤ j ≤ i
3          while ∃x, y∈M_N: related(x, y) ∧ Score(j, x)>0 ∧ Score(j, y)=0
4              for each t ∈ MM_E
5                  Var Scored ← {x ∈ M_N | Score(j, x) > 0}
6                  for each x ∈ Scored
7                      Var OneHop ← {y ∈ M_N | related_t(x, y)}
8                      for y ∈ OneHop \ Scored
9                          Var TwoHop ← {z ∈ M_N | related_t(z, y)}
10                         Score.put((j, y),calculateScore(j,y))
11     return Score
```

A constant exponent α is applied to fine-tune the scoring distribution. Finally, we use a special operator, proposed by [90], to account for elements related to already scored elements through multiple relations. The operator, denoted by the underlined *put* command, merges the scores obtained for each relationship. It assigns a value higher than the maximum of these scores, but lower than 1.

This procedure adjusts the feature location algorithm proposed in [90] in three respects: (A1) We perceive neighborhood as being undirected; relations are navigated in both directions. Not considering directionality is powerful: It allows us to eventually access and score *all* model elements, provided the model is connected. (A2) The weighting factor embodies the intuition that some relations imply a stronger coherence than others. An example is *composition* in UML, which binds the life cycles of elements together. (A3) We modified the scoring formula to reflect our intuition of reinforcement and specificity. The enumerator rewards a large overlap of the set of scored elements and those related to the element being scored, promoting high specificity and high reinforcement. The denominator punishes high connectivity of elements being analyzed, i.e., low specificity, and elements being scored, i.e., low reinforcement.

Step 3: Element Assignment. The scores can now be used to construct a splitting suggestion. A splitting suggestion *Sug* is constructed by assigning suggested model elements to sub-models. When the complete split is desired, i.e., ϕ = *true*, each element is put into the sub-model for which it has the highest score. Ties are broken by selecting one at random. This guarantees that each element is assigned to exactly one sub-model. For a partial split, i.e., ϕ = *false*, an element is assigned to a sub-model only if its score exceeds the user-provided threshold value π. As a result, each element is assigned to zero or one sub-models.

Proposition 1. *Given a set of constraints $\mathcal{C}$ and a graph G that is φ-splittable for every $\varphi \in \mathcal{C}$, the algorithm described in this section computes a graph split $Split(G)$ as defined in Def. 7 s.t. every sub-graph satisfies each $\varphi \in \mathcal{C}$.*

Proof sketch: In step 3, each element is assigned to at most one sub-graph. Thus, all pairs of sub-graphs eventually have disjoint sets of model elements, as required by Def. 7. The resulting sub-graphs satisfy each constraint $\varphi \in \mathcal{C}$ because G was already φ-splittable (Def. 6).

2.6 Tool Support

In this section, we present tool support for the proposed model splitting technique, comprising a set of editors and an engine for creating the splitting suggestion. First, we describe the tool support from the user perspective. Second, we explicate the design goals and actions taken to implement these goals. Third, we discuss the implementation of the tools.

User Process

The user process, shown in Fig. 2.4, comprises two manual tasks (2 and 4) and three automated tasks (1, 3 and 5). The manual tasks rely on human intelligence and domain knowledge. They are facilitated by textual and visual tool support. The automated tasks are triggered by context menu entries.

(1) Start the splitting process. Using a context menu entry on the input model, the user triggers the creation of a splitting description file. The splitting description is automatically opened in a textual editor, shown in Fig. 2.5. By default, the file contains a small usage example.

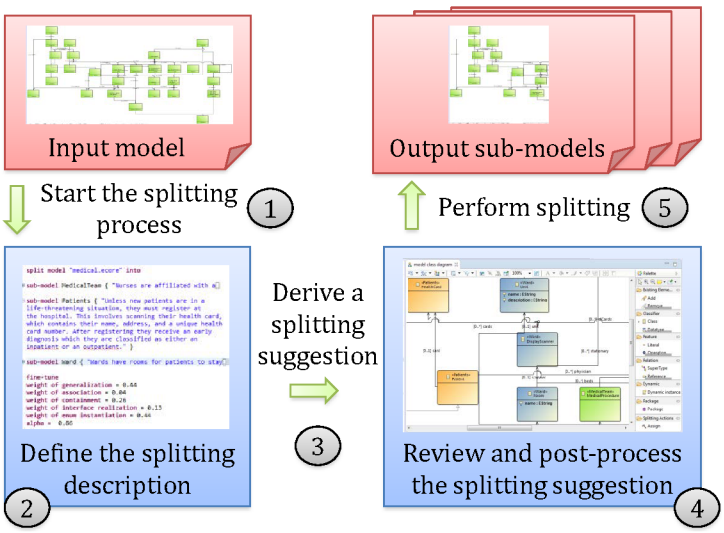

Figure 2.4: An overview of the provided tool support.

(2) Define the splitting description. Using the editor, the user assembles the descriptions of the target sub-models. For a comfortable user experience, the editor provides syntax highlighting, static validation, and folding capabilities. The textual editor is also used for configuration: Adding the keyword `partially` and defining a numerical threshold, the user can set the completeness condition in order to obtain a partial split. Furthermore, the user can fine-tune internal parameters used during the execution of the underlying technique. In Fig. 2.5, the weights assigned to different relationship types and the *alpha* exponent that shapes the scoring function are modified. However, parameter tuning is an optional feature: In Sec. 2.7, we identify a default combination of parameter values that, when applied to six independent class models, yields a considerable accuracy in comparison to hand-tailored decompositions.

(3) Derive a splitting suggestion. Using a context menu entry on the splitting description file, the user triggers the automated creation of a splitting suggestion. A splitting suggestion comprises a set of assignment entries, each holding a link to a model element, a link to a target sub-model, and the relevance score. To compute the splitting suggestion, the technique outlined in Section 2.5 is applied. The splitting suggestion is persisted to the file system.

(4) Review and post-process the suggestion. To obtain visual access to the splitting suggestion, the user can now open the model in a model

```
split model "medical.ecore" into

sub-model MedicalTeam { "Nurses are affiliated with a

sub-model Patients { "Unless new patients are in a
life-threatening situation, they must register at
the hospital. This involves scanning their health card,
which contains their name, address, and a unique health
card number. After registering they receive an early
diagnosis which they are classified as either an
inpatient or an outpatient." }

sub-model Ward { "Wards have rooms for patients to stay

fine-tune
weight of generalization = 0.44
weight of association = 0.04
weight of containment = 0.26
weight of interface realization = 0.13
weight of enum instantiation = 0.44
alpha =  0.86
```

Figure 2.5: Defining the splitting description.

editor. The user activates a dedicated layer called *model splitting*. This action triggers the color-coding of model elements corresponding to the splitting suggestion, shown in Fig. 2.6. As further visual aid, the assignment of a model element is also displayed textually above its name. For post-processing, the user may want to change some assignments for model elements that were not assigned to the proper target sub-model. This is done using the palette tool *Assign*. When the user reassigns a model element, the respective entry in the splitting suggestion is automatically updated. It is worth mentioning that if the user is not satisfied with the results, he or she may iterate Steps 2 to 4 as often as required, tweaking the descriptions and parameter settings. One important scenario for this is the discovery of new sub-models: The user can set the completeness condition to *partial* in Step 2 which leads to some model elements not being assigned in Step 3. The user inspects these elements in Step 4 to create new sub-model descriptions.

(5) Perform splitting. Given that the user is satisfied with the post-processed splitting suggestion, the actual splitting can be triggered by the user. The user may choose from two context menu entries: One for splitting the input model into multiple physical resources, the other for splitting it into sub-packages within the same resource.

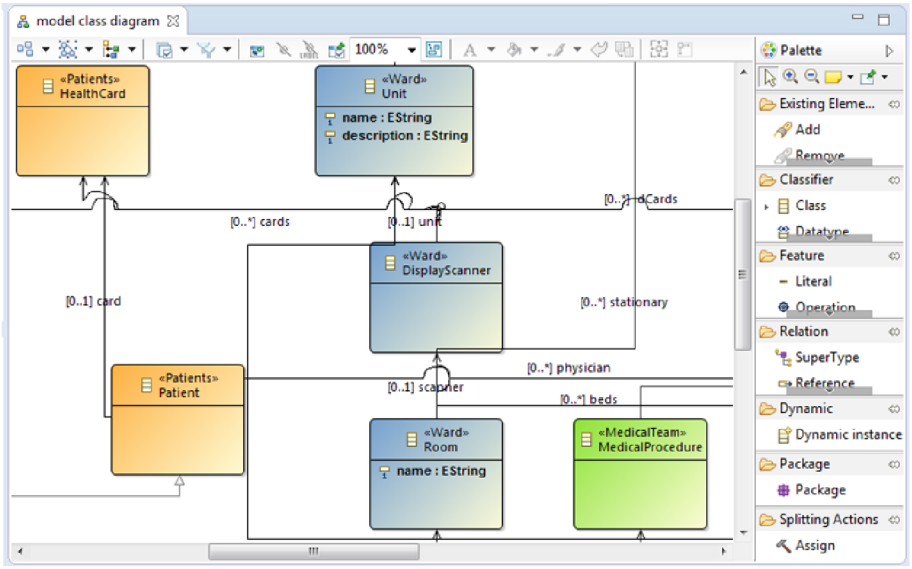

Figure 2.6: Reviewing and post-processing the splitting suggestion.

Design Goals

Two design goals were fundamental in the design of the proposed tool: Extensibility and usability.

Extensibility. The underlying technique possesses an innate extensibility that should be carried over to the end-user. It is applicable to models conforming to arbitrary meta-models, given that they fulfill two properties: (i) Model elements must have meaningful textual descriptions that a splitting description can be matched against. (ii) Except for trivial reconciliation, constraints imposed by the meta-model may not be broken in arbitrary sub-models. We address this design goal by using a framework approach: To customize the tool for a new meta-model, the user subclasses a set of base classes. For instance, to define how input models are converted to a generic graph representation used during crawling, they subclass a class named *GraphBuilder*.

Usability. The design of the tool is informed by *Cognitive Dimensions*, a framework for the human-centered design of languages and tools [42]: Providing an editable visual layer on top of a standard editor is a major step towards *visibility* – visual accessibility of elements and their relations – and away from *viscosity* – resistance to change. *Closeness of mapping* is implemented by a domain-specific language for splitting descriptions with custom editor support. *Premature commitment* is inhib-

ited and *progressive evaluation* is promoted by providing an incremental process that allows tweaking with input values while receiving instant feedback. For *traceability*, our file-based approach to user input allows the user to keep the splitting description and use it later, e.g., for documentation purposes.

Implementation

Eclipse Modeling Framework [106] is the de-facto reference implementation of the EMOF modeling standard [79]. Consequently, it was natural for us to design the tool as an extension for EMF. As such, it can be plugged into an existing Eclipse installation without further effort.

The prototype implementation for the splitting algorithm is written in Java. As input, it receives an input model and text files providing the sub-model descriptions and configuration parameters. For the IR phase, we used the LSA implementation from the open-source SemanticVectors library[3], treating class and interface names as queries, and sub-model descriptions as documents. The crawling phase is performed using a model-type agnostic graph-based representation allowing us to analyze models of different types. We thus transformed the input UML models into that internal representation, focusing only on the elements of interest described above. We disregarded existing package structures in order to compare our results against them. The output sub-models were then transformed back to UML by creating a designated package for each.

For the splitting description editor, we leveraged the powerful code generation facilities of Xtext[4]. We defined a simple domain-specific language for splitting descriptions. The editor with its syntax highlighting and code completion features was fully generated by Xtext. For customization, we added a couple of checks (e.g., forbidden characters, uniqueness of sub-model names). The visual splitting layer is an extension of EcoreTools 2.0[5] which is based on the Sirius[6] framework and part of the Eclipse release Luna 4.4. We decided to use this new technology as we benefit from its support for multiple viewpoints, allowing us to fully customize a splitting viewpoint to our needs.

[3] http://code.google.com/p/semanticvectors/
[4] https://www.eclipse.org/Xtext/
[5] https://www.eclipse.org/ecoretools/
[6] https://www.eclipse.org/sirius/

2.7 Evaluation

In this section, we introduce an evaluation of model splitting. Our goal is to study the applicability and the accuracy of model splitting when applied to real-life models. We focus on two research questions:

- **RQ1:** How useful is the incremental approach for model splitting?
- **RQ2:** How accurate is the automatic splitting?

First, we present the subject models that we considered and the measures taken to obtain splitting descriptions and manual decompositions for comparison. Second, we discuss the methodology and measurements in our evaluation. Third, we discuss our evaluation results, individually addressing each of the research questions.

Subjects

Example	Decomposition Type	SM	CI	AS	CS	AG	GE	IR
HSM	Diagram split	3	28	10	5	4	16	0
GMF	Sub-model decomposition	4	156	62	101	0	70	65
UML	Package decomposition	14	242	283	213	0	205	0
WASL	Package decomposition	4	30	18	13	0	14	0
WebML	Package decomposition	2	23	11	13	0	12	0
R2ML	Package decomposition	6	104	96	27	0	76	0

Table 2.1: Subject models, with numbers of sub-models (SM), Classes and Interfaces (CI), Associations (AS), Compositions (CS), Aggregations (AG), Generalizations (GE) and Interface Realizations (IR)

We chose subject models for our evaluation based on the following criteria: (1) the models should be splittable, as per Def. 6, modulo trivial pre- and post-processing; (2) we have access to an existing, hand-made splitting of the model which can be used for assessing our results; and (3) the splitting is documented, so that we can extract descriptions of the desired sub-models without introducing evaluator bias.

We selected six models that satisfy these criteria. The first four of these were known to the authors (convenience sampling); the last two were obtained by scanning the AtlanMod Zoo on-line collection of meta-models[7]. All models were either initially captured in UML or transformed from EMF to UML. The subjects are shown in Table 2.1 along with their particular decomposition types and metrics: The number of sub-models, classes and interfaces, associations, compositions, aggregations, generalizations, and interface realizations.

The first model, HSM [84], comprises three different diagrams and was already described in Sec. 2.1. Textual descriptions of the sub-models were extracted from [84]. The second, GMF[8], is a meta-model for the specification of graphical editors, consisting of four viewpoint-specific sub-models. Three out of four textual descriptions of the sub-models were obtained from the user documentation on the GMF website. One missing description was taken from a tutorial web site for Eclipse developers[9]. The UML meta-model[10] is organized into 14 packages. The descriptions of these packages were extracted from the overview sections in the UML specification. The description of the four WASL packages was extracted from [132]. The description of the two WebML packages was obtained from the online documentation. Finally, R2ML is a markup language designed for rule interchange between systems and tools. It comprises six packages, each documented in [130].

The second and the third columns in Table 2.1 list the decomposition type and the number of target sub-models for each of the subjects. The last four columns present the size of the subject models in terms of the number of classes and relationships.

In the IR step of our technique, we assume that all provided descriptions focus on explaining their target sub-model. However, in the obtained descriptions, we found that this was frequently not the case: the explanations incorporated explanations of neighboring sub-models. As these misattributed explanations were easy to spot, we manually discarded affected portions from the descriptions. To enable reproducibility, we provide all shortened descriptions online.[11] We recommend users of the technique to carry out the same manual preprocessing.

[7] http://www.emn.fr/z-info/atlanmod/index.php/Zoos
[8] http://www.eclipse.org/modeling/gmp/
[9] http://www.vogella.com/articles/EclipseEMF/article.html
[10] http://www.omg.org/spec/UML/2.5/Beta1/
[11] https://github.com/dstrueber/splittr/tree/master/de.uni_marburg.splittr.evaluation/input

Methodology and Measurement

To investigate **RQ1**, we performed a qualitative analysis using a case study. For **RQ2**, we performed a set of quantitative experiments. To evaluate the accuracy of our splitting technique, we used the following metrics:

1. *Expected*: the number of elements in the predetermined result, i.e., sub-model.
2. *Reported*: the number of elements assigned to the sub-model.
3. *Correct*: the number of elements correctly assigned to the sub-model.
4. *Precision*: the fraction of relevant elements among those reported, calculated as $\frac{Correct}{Reported}$.
5. *Recall*: the fraction of all relevant elements reported, calculated as $\frac{Correct}{Expected}$.
6. *F-measure*: a harmonized measure combining precision and recall, whose value is high if both precision and recall are high, calculated as $\frac{2 \times Precision \times Recall}{Precision + Recall}$. This measure is usually used to evaluate the accuracy of a technique as it does not allow trading-off precision for recall and vice versa.

Our technique relies on a number of configuration parameters described in Sec. 2.5: the calibration parameter α shaping the distribution of scores and the *weight* map w balancing weights of specific relationship types. We fine-tuned these parameters using the hill climbing optimization technique [83]. Our goal was to find a *single* combination of parameter values yielding the best average accuracy for all cases. The motivation for doing so was the premise that a configuration that achieved good results on most members of a set of unrelated class models might produce good results on other class models, too. The results are summarized in Table 2.2.

Results and Discussion

RQ1: How useful is the incremental approach for model splitting?

We evaluate this research question on a case study based on the Graphical Modeling Framework (GMF). GMF comprises four sub-models: *Do-*

Association	0.04
Aggregation	0.13
Composition	0.26
Generalization	0.44
Interface Realization	0.13
α	0.86

Table 2.2: Parameter assignment for class models.

main, *Graphical*, *Tooling*, and *Mapping*. While the sub-models of GMF are already known, they may not necessarily be explicitly present in historically grown meta-models comparable to GMF. We assume that the person in charge of splitting the model is aware of two major viewpoints, *Domain* and *Graphical*, and wants to discover the remaining ones. She provides the meta-model and describes the sub-models as follows:

"***Sub-model Domain*** *contains the information about the defined classes. It shows a root object representing the whole model. This model has children which represent the packages, whose children represent the classes, while the children of the classes represent the attributes of these classes.* ***Sub-model Graphical*** *is used to describe composition of figures forming diagram elements: node, connection, compartment and label.*"

The user decides to begin with an incomplete splitting, since her goal is discovery of potential candidates for new sub-models. An incomplete splitting creates suggestions for sub-models *Domain*, *Graphical* as well as a "*Rest*" – for elements that were not assigned to either of the first two because they did not score above a predefined threshold value. The user can control the size of the *Rest* part by adjusting the threshold value according to her understanding of the model. After a suitable splitting is obtained, the *Rest* part contains the following elements: *ContributionItem, AuditedMetricTarget, DomainElementTarget, Image, Palette, BundleImage, DefaultImage, ToolGroup, MenuAction, MetricRule, NotationElementTarget, ToolRegistry*. From the inspection of these, the user concludes that a portion of the monolithic model seems to be concerned with tooling aspects of graphical editors comprising different kinds of toolbars, menu items, and palettes aligned around the graphical canvas. She describes this intuition:

"***Sub-model Tooling*** *includes the definitions of a Palette, MenuActions, and other UI actions. The palette consists of basic tools being organized in ToolGroups and assigned to a ToolRegistry.*"

Run	Domain	Graphical	Tooling	Mapping
1	80%	77%	–	–
2	80%	84%	90%	–
3	86%	94%	90%	68%

Table 2.3: F-Measure during three runs of incremental splitting.

A next iteration of splitting is performed. This time, the *Rest* comprises only four items: *MetricRule, DomainElementTarget, NotationElementTarget, AuditedMetricTarget.* Three out of these four elements signify a notion of defining relationships between elements of already known sub-models. She concludes that a separate sub-model is required for defining the integration and interrelation of individual sub-models. She performs a third and last splitting after providing a final sub-model description:

"***Sub-model Mapping*** *binds the aspects of editor specification together. To define a mapping, the user creates elements such as NotationElementTarget and DomainElementTarget establishing an assignment between domain and notational elements.*"

To investigate **RQ1** we split it into two research questions: **RQ1.1:** Does the accuracy of splitting improve with each iteration? and **RQ1.2:** Does the approach assist the user in identifying missing sub-models?

RQ1.1: This question can be explored by considering the delta of each sub-model's F-measure during multiple incremental splitting steps. As shown in Table 2.3, the increase of accuracy is monotonic in all sub-models! The same threshold value was used for all splits. The discovery process not only helped the user to discover the desired sub-models but also to create short sub-model descriptions which can later be used for documentation.

RQ1.2: In the first query, the *Rest* part has 12 elements, whereas in the original model, its size was 139. All 12 elements actually belong to the yet undiscovered sub-models, *Tooling* and *Mapping*. Thus, we are able to conclude that the user was successfully guided to concentrate on discovering these sub-models without being distracted by contents of those sub-models she knew about upfront.

RQ2: How accurate is the automatic splitting?

We investigate **RQ2** by answering two research questions: **RQ2.1:** What is the overall accuracy of the splitting approach? and **RQ2.2:** What is the relative contribution of individual aspects of the splitting algorithm on the overall quality of the results?

RQ2.1: Column 4 in Table 2.4 presents average precision, recall and F-measure of our automated technique for each of the subject models. For five out of the six models, the achieved level of accuracy in terms of F-measure was good to excellent (74%-95%). However, the result for UML was not as good (48%). Detailed inspection of this model revealed that package organization of UML has a special, centralized structure: it is based on a set of global hub packages such as *CommonStructure* or *CommonBehavior* that provide basic elements to packages with more specific functionality such as *UseCase* or *StateMachine*. Hub packages are *strongly coupled* with most other packages, i.e., they have a low ratio of inter- to intra-relations. For example, the class *Element* is a transitive superclass for all model elements. This violation of the software engineering principle of *low coupling* hinders our topology-based approach for splitting.

To evaluate whether our algorithm produces meaningful results *except for hubs*, we derived a sub-model of UML which is restricted only to the functional packages. This sub-model, $\text{uml}_{\text{funct}}$, comprises 10 out of 14 packages and 188 out of 242 model elements of UML. As shown in Table 2.4, the accuracy results of $\text{uml}_{\text{funct}}$were similar to the five successful case studies (80%).

RQ2.2: Columns 1, 2 and 3 of Table 2.4 list contributions of individual steps of the algorithm and of the adjustments (A1-3) described in Sec. 2.5. The results after the IR phase are shown in column 1. Compared to the overall quality of the algorithm (column 4), the results are constantly worse in terms of the F-measure, due to low recall values. That is, IR alone is unable to find a sufficient number of relevant elements.

In column 2, we present the results of IR augmented with basic crawling which respects directionality, i.e., does not navigate relations from their inverse end. This version is similar to the crawling technique proposed by Suade but adjusted to operate on models rather than on code-level artifacts. The results are again worse than those of the overall

	1: IR Only			2: IR + Plain			3: IR + Undirected			4: Overall		
	Prec.	**Rec.**	**F-M.**	**Prec.**	**Rec.**	**F-M.**	**Prec.**	**Rec**	**F-M.**	**Prec.**	**Rec.**	**F-M.**
HSM	93%	42%	**56%**	93%	53%	**67%**	78%	78%	**75%**	90%	92%	**89%**
GMF	100%	9%	**17%**	99%	30%	**38%**	68%	72%	**68%**	86%	87%	**86%**
UML	57%	21%	**24%**	37%	20%	**22%**	34%	38%	**30%**	50%	58%	**48%**
WASL	88%	48%	**61%**	72%	29%	**38%**	68%	64%	**63%**	92%	91%	**89%**
WebML	100%	37%	**52%**	100%	40%	**56%**	88%	94%	**90%**	93%	97%	**95%**
R2ML	81%	22%	**32%**	74%	30%	**30%**	46%	49%	**42%**	75%	77%	**74%**
UML_{fn}	67%	22%	**30%**	76%	24%	**33%**	64%	66%	**61%**	84%	80%	**80%**

Table 2.4: Accuracy of model splitting.

technique due to low recall values. Interestingly, in some cases, e.g., WASL, the results are also worse than those of the plain IR technique in terms of both precision and recall, making the scoring schema related to this crawling strategy really inefficient.

Column 3 shows the results when crawling discards directionality, i.e., applies A1. This strategy results in a significant improvement in recall and the overall F-measure compared to the previous approach, but comes together with some decrease in precision.

Column 4 shows the results when the previous approach is extended with scoring modifications (A2-A3). This approach is clearly superior to the previous ones in terms of both precision and recall, and, as a consequence, of the overall F-measure.

We conclude that the basic crawling technique that worked well for code in case of Suade is not directly applicable to models, while our improvements allowed the algorithm to reach high accuracy in terms of both precision and recall.

Threats to Validity

Threats to external validity are most significant for our work: the results of our study might not generalize to other cases. Moreover, because we used a limited number of subjects, the configuration parameters might not generalize without an appropriate tuning. We attempted to mitigate this threat by using real-life case studies of considerable size from various application domains. The ability to select appropriate

sub-model descriptions also influences the general applicability of our results. We attempted to mitigate this threat by retrieving descriptions from publicly available documentation artifacts and documenting the used descriptions online.

2.8 Related Work

Formal approaches to model decomposition. A formally founded approach to model splitting was investigated in [58]. This approach uses strongly connected components (SCCs) to calculate the space of possible decompositions. The user of the technique may either examine the set of all SCCs or try to find reasonable unions of SCCs according to her needs. In [68], the same authors rule out invalid decompositions by providing precise conditions for the splitting of models conforming to arbitrary meta-models. Basic forms of model splitting and joining are considered in [31] where a global model view is split into two local ones with a common interface. These works are orthogonal to ours: Our technique requires a basic notion of a model being splittable, mostly motivated by the need to split class models and meta-models.

Annotation-based splitting. Garmendia et al. propose *EMF Splitter* [37], a tool that facilitates model decomposition as inspired by programming languages and IDEs: The whole model is organized as a project that is divided into packages. Packages are further decomposed into units. As a prerequisite, the tool requires that the input model is annotated manually. This technique is orthogonal to the approach proposed in this work, which provides a heuristic to recommend a specific splitting automatically regardless of the underlying decomposition mechanism. In [38], Garmendia et al. combine *EMF Splitter* with *SAMPLER*, a novel tool that provides visual support for the exploration of large models. It allows focusing on a selected scope of model elements of interest, while elements outside of this scope are combined into placeholder nodes. This technique is orthogonal to the proposed approach as well, which is not concerned with the visual representation of the created sub-models.

Graph clustering for meta-models and architecture models. *Graph clustering* is the activity of finding a partition of a given graph into a set of sub-graphs based on a given objective. Voigt [129] uses graph clustering to provide a divide-and-conquer approach for the matching of meta-models of 1 million elements. Of the different graph clustering

algorithms, the author chose Planar Edge Separator (PES) for its runtime performance, and then adapted it to meta-model matching. Like us, he provides weighting constants for specific relationships kinds; yet [129] only presents the values of these constants and does not evaluate their impact on the quality of the match. From a software engineering perspective, the main drawback of this approach is that the user cannot influence the decomposition in terms of the concepts represented in the resulting sub-models. The same objection may be raised for the meta-model splitting tool proposed in [119]. Our approach bases the decomposition on user description of the desired sub-models, thus avoiding the need for the user to comprehend and classify the resulting components.

Architecture restructing. The architecture restructuring technique by Streekmann [111] is similar to our approach. This technique assumes a legacy architecture that is to be replaced by an improved one. Similar to our technique, the starting point for the new organization is a set of target components together with a set of seeds ([111] calls them *initial mappings*) from which the content is derived. Yet, unlike in our approach, these seeds are specified manually by the developer. The clustering is performed by applying a traditional hierarchical clustering algorithm assigning model elements to components. The algorithm supports the weighting of different types of relationships; tuning these strongly impacts the quality of the decomposition. For the scenarios given in [111], the weighting differs from case to case significantly. In this work, in turn, we were able to find a specific setting of values that produced good results for an (albeit small) selection of unrelated class models. Streekmann also discusses algorithm stability w.r.t. arbitrary decisions made by it. During hierarchical clustering, once two elements are assigned to the same cluster (which, in the case of multiple assignment options, may be an arbitrary choice), this decision is not reversible. Arbitrary decisions in this style do not occur in our approach since we calculate relevance scorings for each sub-model individually.

Model slicing. *Model slicing* is a technique that computes a fragment of the model specified by a property. In the approach by Blouin et al. [13], slicing of a UML class model results in a sub-model which is either *strict*, i.e., it satisfies all structural constraints imposed by the meta-model, or *soft*, if conformity constraints are relaxed in exchange of additional features. For example, slicing a class model can select a class and all of its subclasses, or a class and its supertypes within radius 1, etc. Compared to model splitting, model slicing concentrates on computing a sub-model of interest, ignoring the remainder of the

model. In contrast, we use textual descriptions as input to IR to identify sub-models. The techniques are orthogonal and can be combined, as we plan to do in the future.

2.9 Conclusion

Splitting large monolithic models into disjoint sub-models can improve comprehensibility and facilitate distributed development. In this paper, we proposed an incremental approach for model splitting, supported by an automated technique that relies on information retrieval and model crawling. Our technique was inspired by code-level feature location approaches which we extended and adapted to operate on model-level artifacts. We demonstrated the feasibility of our approach and a high accuracy of the automated model splitting technique on a number of real-life case studies.

Chapter 3

Component Encapsulation for Composite Modeling

This chapter shares material with the FASE'13 paper "Towards a Distributed Modeling Process based on Composite Models" [120].

In this chapter, we address the problem of combining heterogeneous domain-specific modeling languages (DSMLs) used in the specification of large-scale systems. We propose a technique to establish explicit interfaces in a set of related models created using DSMLs. The technique is based on composite models [50], an extension of standard modeling platforms that allows introducing export and import interfaces in models. We give a formalization of the technique based on algebraic graph transformation. To discuss its usefulness, we embed it in a modeling process that gives rise to the systematic development of models.

3.1 Introduction

DSMLs are frequently developed using standard modeling platforms such as EMF [106] and Xtext [34]. In these platforms, models created using separate DSMLs can be combined using *remote references*, links between models. This mechanism is sufficient to establish a physical separation by distributing models over multiple files. Yet, it gives rise to several issues: First, as each model element is allowed to be a target of a remote reference, there is no support for a systematic and disci-

plined combination of models. Second, the target model elements of remote references are temporarily represented by proxy elements. These proxies are replaced by the actual model element on demand. In consequence, all involved models constitute one big model with unrestricted visibility. We conclude that remote reference are not suited to enable a refined, *logical* separation of concerns: Important engineering principles such as encapsulation and information hiding are not neglected.

Composite modeling [50] is an approach aiming to introduce a logical separation of concerns in models. Models are extended to exhibit *export* and *import* interfaces, subsets of model elements provided to or obtained from the environment. A model together with its export and import interfaces is a *model component*. A set of related model components is called a *composite model*. Each model component can have an arbitrary number of interfaces. The design space of possible interfaces is defined in the meta-model of the DSML used to create the model component. Composite modeling is supported by a comprehensive formal theory [51, 52, 53] and basic tooling [50].

We investigate the application of composite modeling in a scenario where a set of related DSMLs and conforming models already exist. In this scenario, we consider *component encapsulation* – the conversion of a set of plain models to a set of model components. Our approach is typing-based: Certain language *types*, meta-model elements, are stipulated as exported and imported. The technique automatically derives suitable export and import interfaces for the models created using the DSMLs. Thereby, these models are encapsulated to yield a composite model.

We briefly revisit the challenges introduced in Sec. 1.1.2 and discuss the prospective improvements.

- Addressing the *lack of systematic development* requires that DSMLs are devised using *"sound principles and methods"* [60]. Remote references, the default mechanism for establishing separation of concerns in standard modeling platforms, promotes an undisciplined combination of models: Fundamentally acknowledged principles such as *high coherence* and *low coupling* [107] are not accounted for. The proposed technique allows enforcing these properties by encapsulating substantial portions of a model behind interfaces. By replacing remote references with designated interfaces, an undisciplined combination of models with its implications for coherence and coupling properties is prohibited.

- Being able to *reuse* a model created using a DSML requires facilities to separate the model's behavior specification from its internal design decisions. In standard modeling platforms, this fundamentally accepted practice known as *information hiding* [80] is complicated by a lack of dedicated concepts. To this end, the proposed technique allows identifying certain types as relevant for the abstract behavior of a component. For instance, in the context of a web service DSML, *services* may be exposed, while *processes* and *tasks* related to internal design decisions are kept internal.
- *Information exposure* refers to the undesired visibility of certain portions of a model to the outside world. This issue is particularly relevant in contexts where secrecy is necessitated by business interests, security, or law. The proposed technique is suited to address this issue by allowing to designate certain types and their elements as exported. Non-exported types and their elements are made inaccessible to the environment. In the context of a corporate data model, an enterprise might be interested in providing its *products* to customers while maintaining secrecy with types such as, e.g., *building plan*.

We make the following contributions:

- We introduce a *component encapsulation technique* to derive the export and import interfaces of a model automatically.
- We formalize this technique in category theory, on the basis of composite graphs.
- We embed the proposed technique into a systematic *modeling process* to show its usefulness. We address questions of how the consistency of model components can be preserved during editing and how model components can be used for code generation.
- We provide a *case scenario* to demonstrate the application of the proposed technique and process.

The rest of this chapter is structured as follows: In Sec. 3.2, we give an overview on encapsulation, illustrating it using a running example. In Sec. 3.3, we describe a modeling process based on composite models, which is a part of the motivation for encapsulation. The underlying theory of composite models is recapitulated in Sec. 3.4. In Sec. 3.5, we consider component encapsulation on a formal level. In Sec. 3.6, we provide an algorithm implementing this formal construction. In Sec. 3.7, we discuss tool support. Sec. 3.8 is dedicated to an application scenario. We discuss related work and conclude in Secs. 3.9 and 3.10.

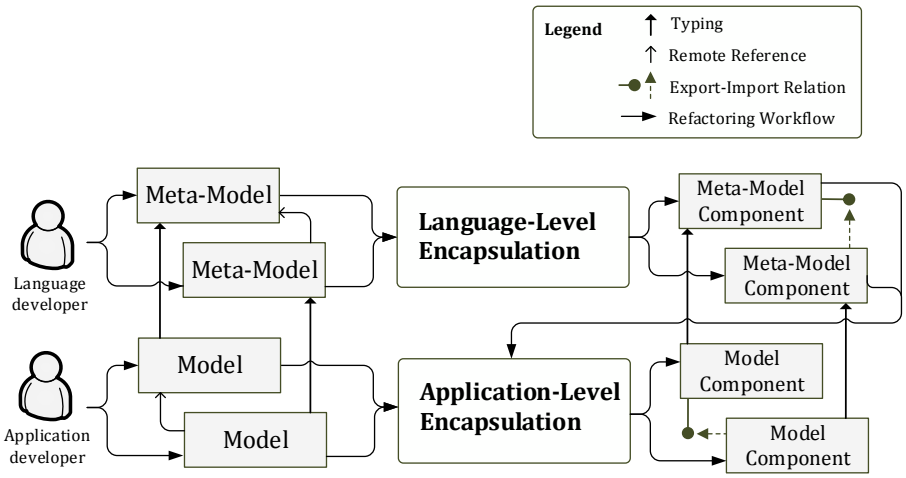

Figure 3.1: An overview of component encapsulation.

3.2 Overview

The key idea of component encapsulation is to derive an encapsulation of models from the encapsulation of their meta-models. A high-level overview of component encapsulation is given in Fig. 3.1. A language developer provides the meta-models to *language-level encapsulation*, an operation that encapsulates these meta-models automatically. It works by replacing all remote references with export and import interfaces. This step is performed once for each set of meta-models. The resulting meta-model components can be used to establish interfaces in any set of conforming models provided by an application developer. The operation producing model components based on these inputs is called *application-level encapsulation*.

The language and application developers in this overview do not necessarily have to be different developers. Specifically in scenarios involving the co-evolution of meta-models and their models, such as the one described in [128], the developer in charge of encapsulating the models may trigger the encapsulation of the meta-models as a prerequisite.

3.2.1 Example

As a running example, consider the model-driven development of web applications using DSMLs. This example is inspired by visual web

modeling languages such as WebML [19], UWE [61], and SWML [17]. A common design decision in these languages is their branching into a set of viewpoint-oriented sub-DSMLs – such as a structural data model, a presentation model and a navigation model. When a web application is developed by a team, contributors typically obtain responsibilities for the different viewpoints. For instance, one contributor acts as domain modeler and another one as presentation modeler. In a full model-driven development process, all specified models are transformed to a running software system using a code generator.

Fig. 3.2 provides the syntax for the *Simple Web Application Language (SWAL)* hypertext and data DSMLs used for the specification of web applications. SwalHypertext and SwalData are specified by means of two meta-models, comprising attributed meta-classes as nodes with directed meta-references as edges.

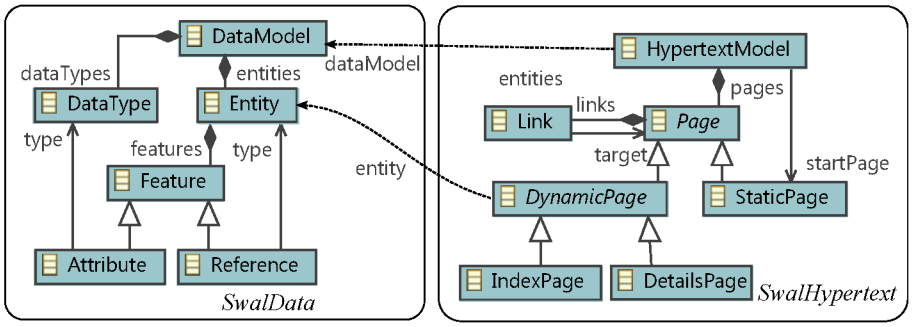

Figure 3.2: SWAL hypertext and data meta-models.

SwalData is used to specify a structural model of persistent data. In a *SwalData* model, a *DataModel* acts as overarching container for all contained elements. Persistent data is based on distinct *Entities* which are characterized by a number of *Features*, i.e. *Attributes* and *References*. An attribute is typed over a primitive *DataType*, a reference over an entity.

SwalHypertext is used to specify the interconnected pages in a web application, some of them displaying persistent data based on a *SwalData* model. In a *SwalHypertext* model, the class *HypertextModel* is used as overarching container. It contains a hypertext structure of interconnected pages and a link to a *DataModel*. The hypertext structure is based on *Pages* being interconnected through *Links*. Depending on its content, a page can either be dynamic or static. A dynamic page refers to an entity and can either be an *index page* displaying a list of available data records or a *details page* presenting a detailed view for a specific record.

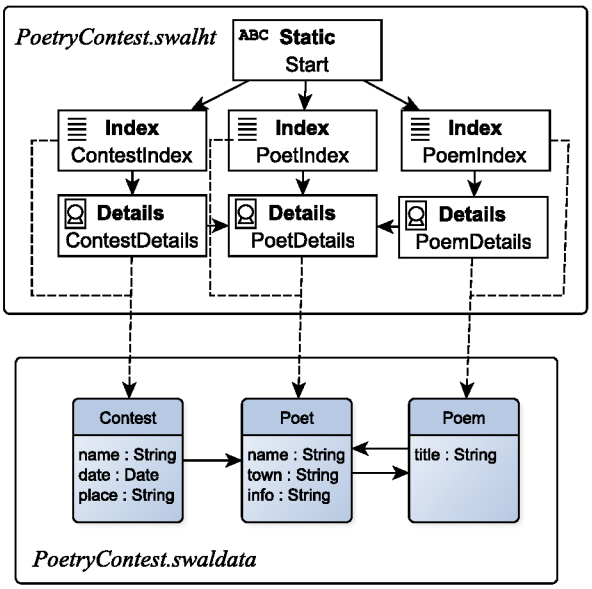

Figure 3.3: Poetry contest hypertext and data models.

In Fig. 3.3 a poetry contest web application is specified. Contest, poet, and poem entities are to be displayed on index and details pages. The concrete syntax given in the presentation facilitates convenient editing by hiding the *DataModel* and *HypertextModel* classes: pages and entities are visualized as nodes in different layouts. Hyperlinks, references, and links between pages and entities are visualized as arrows.

In Fig. 3.4, *SwalData* and *SwalHypertext* have been encapsulated manually. In *SwalData*, the *DataModel* and *Entity* classes, previously targets to remote references, are now part of an export interface. In *SwalHypertext*, delegate classes of these two classes have been created and added to a newly created import interface. The remote references have been replaced by internal references to these delegate classes. As a result, both meta-models are now self-contained in the sense that all semantic references run between classes within the same component. However, they are also interrelated as they are equipped with interfaces.

In Fig. 3.5, the poetry contest model has been encapsulated based on the meta-model encapsulation introduced in Fig. 3.4. Both meta-model components are instantiated by conforming model components. In particular, export and import interfaces are instantiated and used for the sharing of entities between both components.

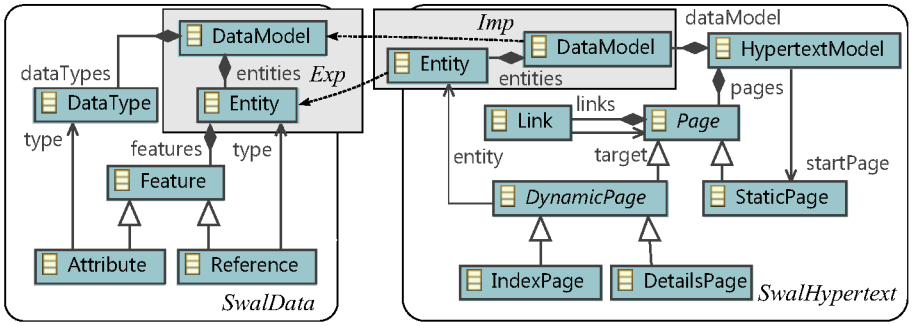

Figure 3.4: Result of encapsulating the SWAL meta-models.

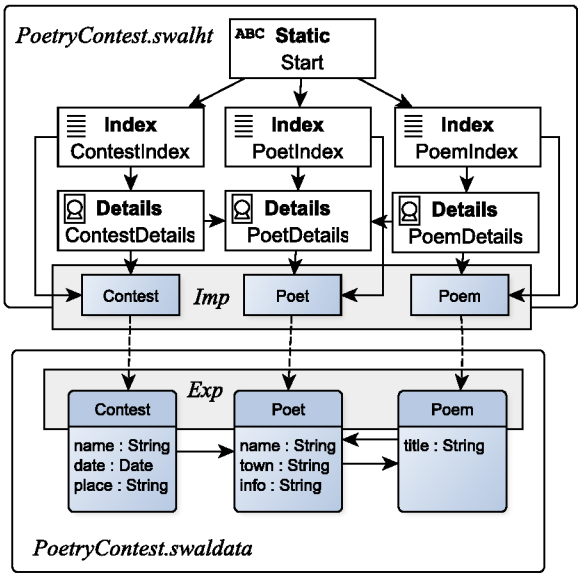

Figure 3.5: Result of encapsulating the poetry model.

3.2.2 Automated Technique

In this example, it is evident that a basic form of component encapsulation can be fully automated on language and application level: In the language meta-model, the encapsulation is based on replacing all remote references by export and import relationships. The encapsulation of the conforming models is determined by the typing of model elements and their assignment to meta-model components.

After performing language encapsulation automatically, it is advisable to include a manual post-processing step in which the language developers adjust the performed encapsulation: They may identify meta-classes not targeted by remote references, but still relevant for application developers to understand the context of specific model elements.

3.3 Modeling Process based on Composite Models

As a key motivation for component encapsulation, we put forward the claim that component encapsulation facilitates *"constructing large models in a systematic manner"* [60]. To substantiate this claim, we consider an overarching process for the model-driven development of software systems. The process uses component encapsulation to address the following three questions arising during model-driven development: (1) How can the structure of existing models be improved to support modularity? (2) How can the consistency of related models be ensured during collaborative modeling? (3) How can related models be used for code generation?

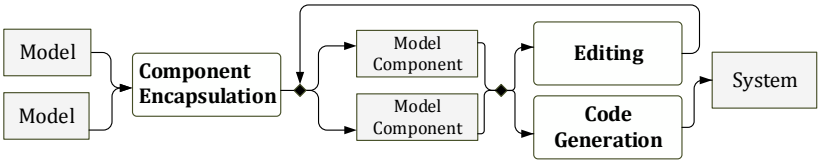

Figure 3.6: Systematic modeling process based on composite models.

Fig. 3.3 gives an outline of the process: It assumes that a set of models created using related DSMLs is provided as input. The models are then encapsulated to become model components. In the subsequent phase, editing steps are performed, possibly involving multiple developers collaborating on the components in parallel. Once that a stable revision of the model components has been established, code generation is performed to obtain the specified system.

An optional extension of this process involves the scenario where the starting point is a single language with a monolithic meta-model and conforming models. In this case, the meta-model needs to be split upfront: To this end, the splitting technique outlined in Chapter 2 can be applied. After the meta-model has been split, the conforming models have to be split along the meta-model decomposition. It is worth pointing out that, while neither splitting nor encapsulation are tied to a specific process, the process introduced in this chapter can be considered a methodological framework that allows combining them.

Encapsulation. As discussed in Sec. 3.1, component encapsulation can be performed automatically if an encapsulation of the meta-models for the input models is provided. To encapsulate models conforming to large meta-models, such as UML, it makes sense to proceed in several iterations of encapsulation. An example scenario for UML can look like

this: (1) encapsulate the structure component from the behavioral component, (2) encapsulate the structure component further into package and class structure components, (3) encapsulate the behavioral component into a basic action component and a behavior diagrams component, and (4) continue encapsulating this component until the well-known behavior diagrams are each separated in model components.

Editing. A crucial challenge of collaborative editing is to keep editing steps as independent as possible while preserving the consistency of models.[1] Component encapsulation enables an *inconsistency avoidance* strategy of giving editing steps at hand that are classified as either safe or critical to the consistency of models. To avoid dead-lock situations and foster the natural evolution of software projects, we provide the notion of a *relaxed* consistency avoidance that allows performing critical steps if necessary. This strategy stands in contrast to approaches incorporating an entirely liberal strategy of allowing arbitrary operations and providing tool support to detect and repair inconsistencies [40, 74]. Such strategies pose the risk that a later reconciliation is not possible without significant manual effort. Furthermore, it relies on the capability to perform a global consistency check, which may be unavailable for business, security, or legal reasons.

We consider asynchronous and synchronous editing: Components might be evolved by multiple developers in parallel or by one developer. For example, during the evolution of the model specified in Fig. 3.5, new requirements may emerge, e.g., the management of books. When domain and hypertext components are developed in an asynchronous manner, the domain modeler begins by adding this new entity to the body and export of the *SwalData* component. The hypertext modeler then adds the entity to the import interface and body of the *SwalHt* component and creates corresponding pages for the entity, resulting in the model shown in Fig. 3.7. This change could also be performed in a synchronous manner, using an editing command that adds an entity and corresponding pages to both components.

Synchronous and asynchronous editing steps can be expressed as model transformation rules on a composite model. Two example rules are shown in Fig. 3.8. *Del* and *New* tags denote nodes as being deleted from the model or newly created, respectively. Rule a specifies the synchronous addition of a new entity and corresponding index and details pages to both components. Rule b specifies an asynchronous editing

[1] Consistency refers to the *absence of inconsistencies*, i.e., contradictory information within a set of models.

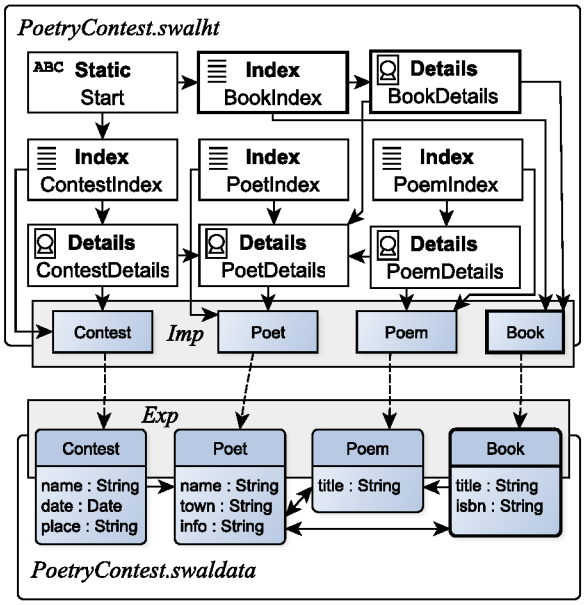

Figure 3.7: Model components after editing. Thick borders indicate newly added elements.

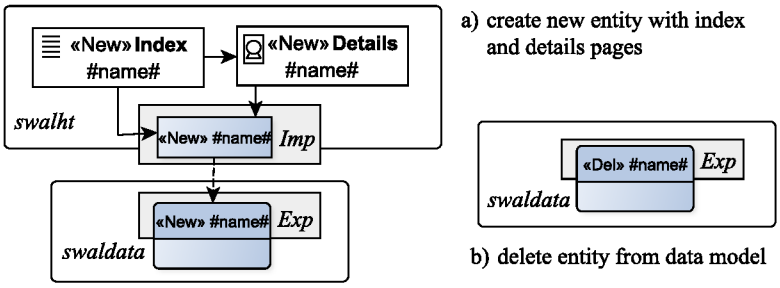

Figure 3.8: Synchronous and asynchronous composite model transformation rules.

step, the deletion of an entity from a body and an adjacent export. Rule a is neutral wrt. consistency: It introduces an intact export/import relation. The editing step specified by Rule b is to be considered a critical one since the export interface being edited might be referred from a remote import interface. A user performing this editing step may be warned and suggested to communicate the editing step to other team members.

Code Generation. To provide a full model-driven development process, code generation based on composite models needs to be inves-

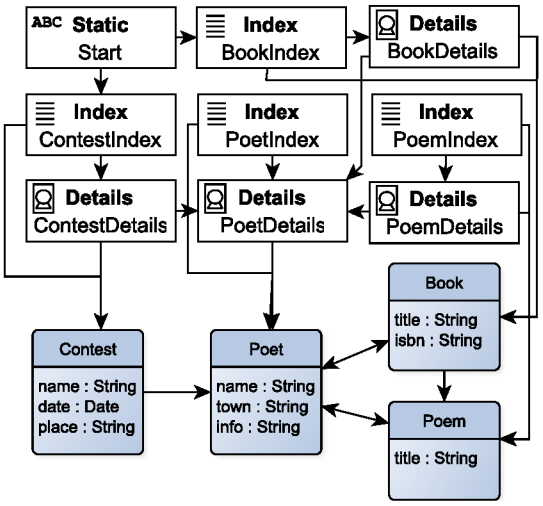

Figure 3.9: Merged application model.

tigated. Considering models that cannot be transmitted due to business, security or legal reasons, a *distributed* code generation technique is desirable. Such a technique may allow to generate the code based on each component individually, increasing the flexibility during component development. Yet, developing such a technique is a challenging task: Interrelations between model components need to be reflected in interrelations between the generated artifacts. To ensure that the code generated for a component yields syntactically correct code, the framework needs to enforce that these interrelations are intact. An alternative is to apply a *centralized* code generation, based on a code generator for the complete set of model components. In the case that a split has been performed as a preliminary step to component encapsulation, an existing code generator for the whole model may exist, the reuse of which is desirable. Given a composite model, such a model can be produced by merging it, using the information provided by export and import relations. Fig. 3.9 shows the result of merging the model shown in Fig. 3.7.

3.4 Preliminaries

In this section, we revisit the foundation of composite modeling. First, we consider composite models and their formalization based on composite graphs. Second, we revisit core concepts from algebraic graph transformation and their extension to composite graph transformation.

3.4.1 Composite Graphs

In a composite model, modularity is established through the definition of explicit export and import interfaces. Export and import interfaces identify model elements provided to and obtained from the environment. A model together with its export and import interfaces is a *model component*. A set of related model components is a *composite model*. The core model of a component is called its *body*. Export and import interfaces specify subsets of the body's model elements: Model elements contained in export and import interfaces are identified with model elements in the body. Import interface elements are also identified with elements in an export. An import is assigned to exactly one export interface. An export can correspond to an arbitrary number of import interfaces. An interface can hide structural complexity of the body, e.g., flatten its inheritance hierarchy. The design space of possible interfaces is defined in the meta-model of the component.

Graphs are well-suited to capture the structure of composite models. The basis of our formalization are typed graphs and graph morphisms as defined in Sec. 2.4. Graph morphisms are structure-preserving mappings between graphs. The typing relation between a model and its meta-model is formalized using a graph morphism called type morphism. Furthermore, our formalization of composite models will be based on category theory, a framework that allows considering graph structures in a very generic manner. Typed graphs and graph morphisms form the category GRAPHS_{TG} [33]. Due to the generic nature of category theory, it is also possible to use other kinds of graphs and morphisms as basic ingredients of composite graphs. For example, composite graphs over typed graphs with inheritance and containment as well as suitable graph morphisms are considered in [53].

Definition 8 (Composite network graph)**.** *A* composite network graph *is a graph G (Def. 1) typed over graph CNG (shown in Fig. 3.10) by a graph morphism $t : G \to CNG$ (Def. 3) such that the following properties hold:*

- *each* export node *is source of exactly one network edge running to a body node*
- *each* import node *is source of exactly two network edges, one edge is running to a body node and the other to an export node.*

If there are export nodes without outgoing edges, corresponding composite network graphs are called weak.

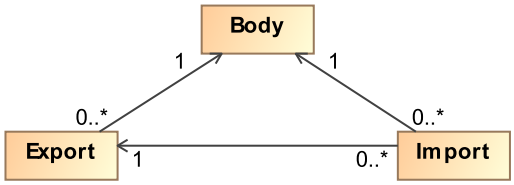

Figure 3.10: Type graph *CNG* for composite network graphs.

The notion of weak composite graphs allows us to consider settings with *restricted visibility*, a crucial concern in the motivating problems considered in this chapter. For instance, a developer team might be granted visibility for one full component and visibility of the export interfaces of all related components.

For example, just considering its network graph, the composite model in Fig. 3.5 comprises two body, one export, and one import node together with the required network edges. The body of the *swaldata* component might be hidden from visibility of the *swalht* developers, rendering their scope of visibility a weak composite network graph.

Definition 9 (Composite network graph morphism)**.** *Given two network graphs $g : G \to CNG$ and $h\colon H \to CNG$, an injective graph morphism $f\colon G \to H$ forms a valid* composite network graph morphism, *short* network morphism, *if $h \circ f = g$.*

Composite network graphs and network graph morphisms form a category, called CompoNetGraphs, that is co-complete [53]. Weak composite network graphs and their morphisms also form a category, however, this one does not have pushouts.

Definition 10 (Composite graph)**.** *Given a (weak) composite network graph G, a* (weak) composite graph $\hat{G}$ *over G is defined as $\hat{G} = (G, \mathcal{G}(G), \mathcal{M}(G))$ with*

- *$\mathcal{G}(G)$ being a set of graphs, called* local graphs, *of category* Graphs *with each graph uniquely refining a network node in G_N: $\mathcal{G}(G) = \{\hat{G}(n) |\ \hat{G}(n)$ is a graph and $n \in G_N\}$,*
- *$\mathcal{M}(G)$ being a set of graph morphisms, called* local (graph) morphisms, *each refining a network edge in G_E: $\mathcal{M}(G) = \{\hat{G}(e)\colon \hat{G}(i) \to \hat{G}(j) | \hat{G}(e)$ is a graph morphism and $e \in G_E$ with $s(e) = i$ and $t(e) = j\}$, and*
- *for all paths $\hat{G}(x) \circ \hat{G}(y), \hat{G}(z) : \hat{G}(A) \to \hat{G}(B)$ we have $\hat{G}(x) \circ \hat{G}(y) = \hat{G}(z)$ with $x, y, z \in G_E$. (commutative morphisms)*

For example, the composite model in Fig. 3.5 has one local graph for each of its four network nodes. The model is shown in a compact representation where corresponding elements in body and adjacent export or import graphs are only shown once, as part of the interface.

Definition 11 (Composite graph morphism)**.** *Given two (weak) composite graphs $\hat{G}$ and $\hat{H}$ with composite network graphs G and H, resp., a* (weak) composite (graph) morphism, *written $\hat{f}\colon \hat{G} \to \hat{H}$, is a pair $\hat{f} = (f, m)$ where*

- *$f : G \to H$ is a composite network graph morphism and*
- *m is a family of morphisms $\{\hat{f}(n) \mid n \in G_N\}$ such that*
  - *for all nodes $i \in G_N : \hat{f}(i)\colon \hat{G}(i) \to \hat{H}(f_N(i))$ is a graph morphism and*
  - *for all edges $e\colon i \to j \in G_E : \hat{H}(f_E(e)) \circ \hat{f}(i) = \hat{f}(j) \circ \hat{G}(e)$ (see Fig. 3.11).*

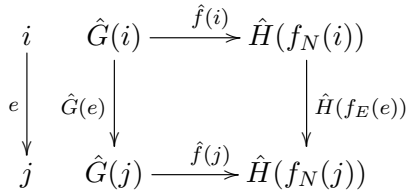

Figure 3.11: Illustration of a composite graph morphism.

If morphism f and all morphisms in m are inclusions (injective), $\hat{f}$ is called inclusion (injective). Given a graph $\hat{TG}$ and a composite morphism $\hat{t} : \hat{G} \to \hat{TG}$ is called typed composite graph*.*

Composite graphs and graph morphisms form a category, called CompGraphs, being co-complete. Weak composite graphs and weak composite morphisms form category CompGraphs^{weak}. Typed composite graphs and their morphisms form a category called CompGraphs$_{TG}$ [53].

This formalization induces that composite graphs are consistent in a certain sense: Since all morphisms have to be total, especially the ones between import and export interfaces, inconsistencies between components in the sense of unsatisfied imports may not occur. It is up to future work to adapt composite models such that temporary inconsistencies are tolerated, i.e., partial import mappings are allowed.

3.4.2 Rules and Composite Graph Rules

The systematic modification and evolution of composite models can be formalized using rules. In the following, we recall the main definitions of the rule-based algebraic approach to graph transformation called the *gluing approach*. In this approach, graph elements occurring in the left and right-hand sides of a rule, i.e., in an *interface graph*, are used to glue new elements to already existing ones.

Definition 12 (Rule)**.** *A (production) rule* $p = L \xleftarrow{l} I \xrightarrow{r} R$ *consists of graphs* L*,* I *and* R*, called* left-hand side, interface graph *and* right-hand side*, respectively, and two injective graph morphisms,* l *and* r*.*

An example for a simple rule is obtained by considering Fig. 3.8 without the rectangle labeled *"Exp"*. The *Del* annotation of the remaining node denotes it as being contained in $L \setminus I$. Since the intention of this rule is to specify a deletion, the I and R graphs are empty.

A rule is applied using a match m of its left-hand side to a given graph G. The application of a graph rule consists of two steps: First, all graph elements in $m(L - l(I))$ are deleted. Nodes to be deleted may have adjacent edges which have not been matched, so the rule application may produce dangling edges. Therefore, all matches m have to satisfy the *gluing condition*: If a node $n \in m(L)$ is to be deleted by the rule application, it has to delete all adjacent edges as well. Afterwards, unique copies of $R - r(I)$ are added. This behavior can be characterized by a double-pushout [32]. Given a rule and a match, the resulting rule application is unique [32].

Definition 13 (Rule application)**.** *Let a rule* $p = L \xleftarrow{l} I \xrightarrow{r} R$ *and a graph* G *with a total graph morphism* $m : L \to G$ *be given. A* rule application *from* G *to a graph* H*, written* $G \Rightarrow_{p,m} H$*, is given by the diagram in Fig. 3.12 where (1) and (2) are pushouts. We refer to* G*,* m *and* H *as a* start graph*, a* match*, and a* result graph*, respectively.*

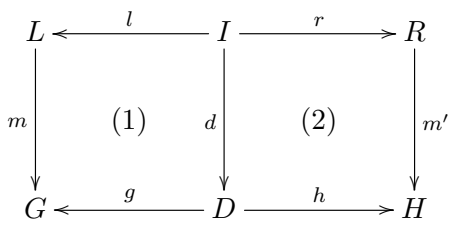

Figure 3.12: Rule application by a double pushout (DPO).

We now lift the established concepts of rules and their application to composite models. A composite graph rule is a rule that allows the transformation of a composite model.

Definition 14 (Composite graph rule)**.** *Given a distinguished graph $\hat{TG}$, called composite type graph, a* composite graph rule $\hat{p} = (\hat{L} \overset{\hat{l}}{\hookleftarrow} \hat{K} \overset{\hat{r}}{\hookrightarrow} \hat{R}, \hat{type})$ *consists of composite graphs $\hat{L}$, $\hat{K}$, and $\hat{R}$ typed over $\hat{TG}$ by the triple $\hat{type} = (\hat{type}_{\hat{L}} \colon \hat{L} \to \hat{TG}, \hat{type}_{\hat{K}} \colon \hat{K} \to \hat{TG}, \hat{type}_{\hat{R}} \colon \hat{R} \to \hat{TG})$ being composite morphisms and typed composite morphisms $\hat{l} \colon \hat{K} \hookrightarrow \hat{L}$ and $\hat{r} \colon \hat{K} \hookrightarrow \hat{R}$ being* inclusions *such that $\forall n \in K_N \colon \hat{p}(n) = (\hat{L}(n) \overset{\hat{l}(n)}{\hookleftarrow} \hat{K}(n) \overset{\hat{r}(n)}{\hookrightarrow} \hat{R}(n), \hat{type}(n))$ is a rule.*

Two composite graph rules are shown in Fig. 3.8. *Del* and *New* tags denote nodes as being contained in $\hat{L} - \hat{K}$ or $\hat{R} - \hat{K}$, respectively.

Definition 15 (Composite graph transformation)**.** *A* composite graph transformation (step) $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ *of a typed composite graph $\hat{G}$ to $\hat{H}$ by a (weak) composite graph rule $\hat{p}$ and a typed injective composite morphism $\hat{m} \colon \hat{L} \to \hat{G}$ is shown in Fig. 3.13. (1) and (2) are pushouts in the category of* CompGraphs$_{TG}$ *(*CompGraphs$^{weak}_{TG}$*).*

A composite graph transformation *is a sequence $\hat{G}_0 \Rightarrow \hat{G}_1 \Rightarrow \ldots \Rightarrow \hat{G}_n$ of direct composite graph transformations, written $\hat{G}_0 \overset{*}{\Rightarrow} \hat{G}_n$.*

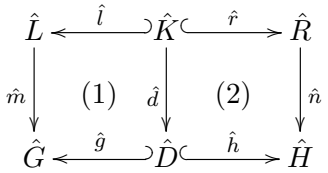

Figure 3.13: Illustration of a composite graph transformation step by means of two pushouts.

In general, a transformation step $\hat{G} \overset{\hat{p},\hat{m}}{\Longrightarrow} \hat{H}$ can be performed if $\hat{m}$ fulfills the *composite gluing condition*: The resulting structure must be a well-formed composite graph. Otherwise, it can happen that context edges dangle afterwards. The gluing condition has to be checked on the network and all local transformations. Moreover, for all deleted network nodes, the local graphs have to be fully determined by the match, and local graph elements may be deleted only if there are no preserved interface elements being mapped to them. Weak composite rules are not allowed to change stand-alone exports or to produce stand-alone exports by deleting their body. (For more details see [50].)

Transformation steps are performed component-wise, i.e., by performing the network transformation first and all local transformations for the involved network nodes afterwards if all composite morphisms are injective. Fig. 3.14 exemplifies this process. Rule $\hat{p}$ is applied on input model $\hat{G}$ by first applying the network rule p on the network graph G: p comprises just the "large" nodes, labeled B and E, and the edge between these nodes. It specifies the creation of an export node and a network edge connecting the export to an existing body node. In the second step, the object graphs of the existing body node and the new export node are updated to contain the specified nodes and edges. In this figure, numbers denote mappings as induced by the graph morphisms.

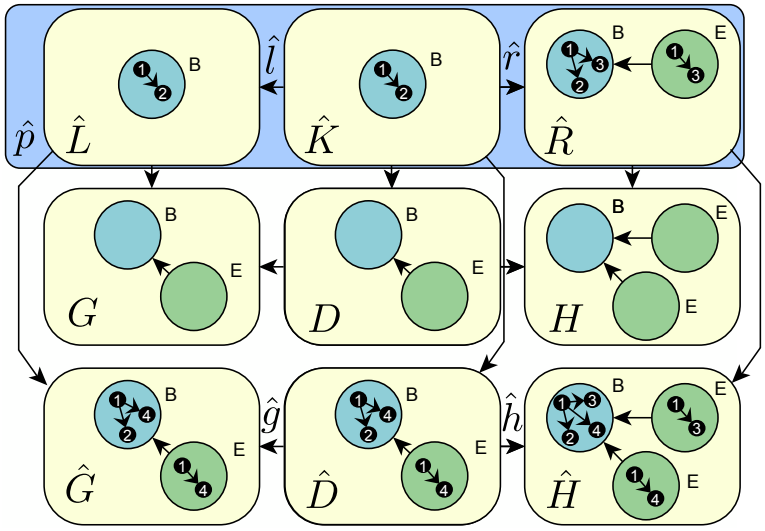

Figure 3.14: Component-wise application of a composite graph rule.

Applying Rule a shown in in Fig. 3.8 to the model in Fig. 3.5, nothing is deleted. Consequently, the composite gluing condition is obviously fulfilled. To obtain the composite model in Fig. 3.7, variable "name" has to be instantiated by "book". Additional references between book, poem, and poet entities have to be added by another editing step.

Merging a Composite Graph

Merging as exemplified in Fig. 3.9 can be formalized using a colimit construction.

Definition 16 (Graph merge)**.** *Given a composite graph, there is a unique graph containing its merge result: Considering a composite graph C as a diagram in category* GRAPHS_{TG}*, its colimit consists of a simple graph G and a family of graph mophisms from all local graphs of C to G. The colimit construction is uniquely determined.*

3.5 Framework

In this section, we provide a formal framework for component encapsulation, following the overview shown in Fig. 3.1: We define its two components, *language-* and *application-level encapsulation*, by specifying their input and output precisely. Language-level encapsulation provides a certain degree of freedom with regard to its instantiation. Application-level encapsulation, in contrast, is uniquely determined by the provided input models and language-level encapsulation. We later discuss our instantiation of both components. We consider the binary case of encapsulating two related models. Three or more related models can be encapsulated by successively applying binary encapsulation.

Proposition 2 (Binary encapsulation of graphs)**.** *Given graph G and two subgraphs G_1 and G_2 with inclusions $g_1 \colon G_1 \to G$ and $g_2 \colon G_2 \to G$, their interconnecting interfaces can be uniquely determined such that the resulting diagram forms a valid composite graph with two components.*

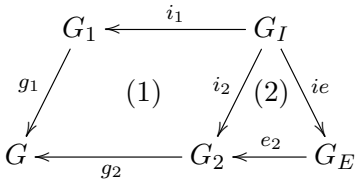

Figure 3.15: Binary encapsulation of a composite graph.

Proof. In Fig. 3.15, let square (1) be a pullback and (2) an epi-mono-factorization. Then, graph G_I and morphisms i_1 and i_2 are uniquely determined, up to isomorphism. The epi-mono-factorization splits morphism i_2 into a surjective and an injective part. Graph G_E and morphisms e_2 and ie are uniquely determined by this factorization. Diagram i_1, ie, and e_2 forms a valid composite graph with two components. Its network graph is well typed over the component network graph defined in Def. 8. □

For example, consider the *SwalData* and *SwalHypertext* meta-models in Fig. 3.2. We are free to consider additional model elements not contained in *SwalHypertext* as part of G_1: Choosing *SwalHypertext* including its outgoing remote references, their target classes and the reference between these classes as G_1, *SwalData* as G_2, and the graph yielded by the union of all classes and references as G, we obtain export and import interfaces G_I and G_E as shown in Fig. 3.4.

We provide a formalization of application-level encapsulation along a given language-level encapsulation. We show that the result of the construction is uniquely determined.

Proposition 3 (Binary encapsulation of typed graphs)**.** *Given a type graph TG with subgraphs TG_1 and TG_2 and their binary encapsulation as in the upper part in Fig. 3.16. Moreover, graph G with subgraphs G_1 and G_2 and typings t, t_1, t_2 over TG, TG_1, TG_2 are given so that (t_1, g_1) is a pullback over (t, t_{g1}) and (t_2, g_2) is a pullback over (t, t_{g2}). There is a unique binary encapsulation of G_1 and G_2 being type compatible with the resulting composite type graph. That is, all morphisms in Fig. 3.16 exist and form a commuting diagram. The result is a composite graph typed over the encapsulation of $TG1$ and $TG2$.*

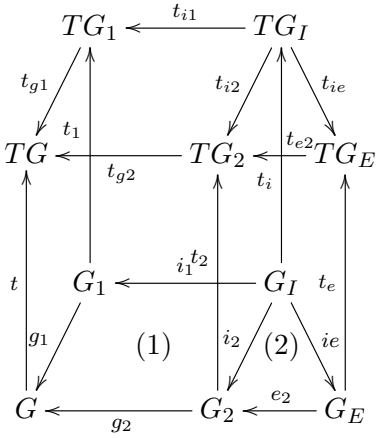

Figure 3.16: Binary encapsulation of typed composite graph.

Proof. The following steps can be performed:

1. (t_i, i_1) is constructed as pullback over (t_{i1}, t_1).
2. Morphism $i2$ is the induced morphism by pullback (t_2, g_2) and morphisms $g1 \circ i_1$ and $t_{i2} \circ t_i$ such that $t_2 \circ i_2 = t_{i2} \circ t_i$ and $g1 \circ i_1 = g2 \circ i_2$.
3. (e_2, t_e) is constructed as pullback over (t_2, t_{e2}).
4. Morphism ie is the induced morphism by the pullback (e_2, t_e) and morphisms i_2 and $t_i \circ t_{ie}$ such that $i_2 = e_2 \circ ie$ and $t_{ie} \circ t_i = t_e \circ ie$.

□

In the example, encapsulating the poetry application model in Fig. 3.3 over the meta-model encapsulation shown Fig. 3.4 gives us the composite model illustrated in Fig. 3.5.

3.6 Instantiation

In this section, we introduce our instantiation of component encapsulation. The key idea is to establish a *conservative* encapsulation in which interfaces only contain elements required to combine the models, i.e., elements previously targeted by remote references. To include additional elements, the interfaces may be post-processed manually.

The basic procedure, shown as Algorithm 2, works in the same way for both components: It implements language-level encapsulation as per Prop. 3.15 and application-level encapsulation as per Prop. 3.16. The main difference concerns the way in which the types for certain copied objects are determined: On the language level, this is a trivial issue since meta-models only contain elements of the type "meta-class" (we formalize meta-models using simple graphs in Prop. 3.15). On the application level, the meta-model encapsulation needs to be inspected (cf. Prop. 3.16). The procedure assumes two input models: Remote references running from the *right* to the *left* model are replaced by export and import relationships. To perform this substitution in both directions, the procedure can be applied a second time, providing the input models in swapped order. To encapsulate a set of more than two models, the procedure can be applied on each pair of models successively.

The main idea of this procedure is to consider each reference in the right model and, in case that a remote reference targeting the left model is detected, restructure both models accordingly. To this end, new export and import interfaces are created in lines 2–3. In lines 4–7, elements in the left model targeted by remote references are identified. For each such element, corresponding export, import, and delegate objects are determined in lines 8-10: If we consider an element for the first time, we create these objects in the target body or interface. If we considered it at an earlier point, we just refer to these existing elements. The methods used to create the new objects ensure that the typing of these objects is correct. On language level, we only consider meta-classes and -references (cf. Prop. 3.15). On application level, the types are retrieved from the available body, export, and import metamodels (Prop. 3.16). The target of the remote reference is updated in line 11 to refer to the delegate object. In line 12, we add interconnecting references between the newly created elements. In 13–15, we make sure that we do not add empty export or import interfaces to a component.

For example, applying this procedure on the models shown in Fig. 3.2, using the *SwalData* meta-model as left and the *SwalHypertext* meta-

Algorithm 2 Component encapsulation.

```
Input: left: Left model.
Input: right: Right model.
 1 procedure ENCAPSULATECOMPONENTS(left, right)
 2     var export = new export(left)
 3     var import = new import(right, export)
 4     for each e ← right.elements
 5         for each r ← e.references
 6             if r.target ∈ left.elements then
 7                 var remoteElem ← r.target
 8                 var exp ← export.findOrAddCopy(remoteElem)
 9                 var imp ← import.findOrAddCopy(remoteElem,exp)
10                 var delegate ← right.findOrAddCopy(imp)
11                 r.target ← delegate
12     addInterconnectingReferences(body, export, import)
13     if (!export.isEmpty && !import.isEmpty) then
14         left.addExport(export)
15         right.addImport(import)
16     return
```

model as right model, yields the model in Fig. 3.4: In lines 2–3, the export and import interfaces are initialized. The two remote references *dataModel* and *entity* are identified in line 4–6. Consequently, new meta-classes corresponding to the *DataModel* and *Entity* classes are created and added to the interfaces and body in lines 7–10. These meta-classes are set as new targets for the remote references in line 11. The reference between *DataModel* and *Entity* is added in line 12. Since the interfaces are not empty, they are added to their models in lines 13-15.

Applying the procedure on the models shown in Fig. 3.3, using the poetry data model as left and the hypertext model as right model, yields the model in Fig. 3.5: The interfaces are created in lines 2–3. All six visible remote references are considered once in the loops in line 4–6. One additional reference not visible in the concrete syntax representation is considered, running between the *HypertextModel* and *DataModel* objects. In lines 7–10, the first time each of the four target objects – *Contest*, *Poet*, *Poem*, and *DataModel* – is considered, corresponding objects are created. The sub-procedures determine the typing for these objects: The meta-classes created during the language-level encapsulation are used as types. The new object in the body is set as reference target in line 11. References between the data model and its entities are added in line 12. The interfaces are added in lines 13–15.

3.7 Tool Support

The core processing of composite models is supported by an existing editor environment based on the Eclipse Modeling Framework [106]. For a set of related models, wizard are provided allowing to perform component encapsulation as proposed in this chapter. Export and import interfaces are implemented as separate resources with special references, supported by a delegation mechanism that replaces EMF's proxy concept. Furthermore, we have implemented a model transformation language and tool set allowing the specification and execution of editing steps. Our basic implementation comprises a rule editor and interpreter engine suite. Using this implementation, it is possible to deploy transformation rules as editing steps, e.g. refactorings, within existing editors such as Papyrus. The transformation language can also be used for model-to-model transformation, e.g., to support a cleanup step before code generation. The tool set is open source being provided at `http://www.uni-marburg.de/fb12/swt/forschung/software` along with examples and a tutorial.

3.8 Application Scenario

In this section, we investigate the claim that component encapsulation can facilitate the systematic development of models. We provide an application scenario concerned with the collaborative development of a web application using purpose-tailored DSMLs. Inspired by a similar scenario first introduced in [8, 57], the application under development is an administrative web application for a vehicle rental company. The models in the scenario are based on the domain and hypertext DSMLs introduced in Sec. 3.2.1. We assume that these languages have been subject to encapsulation, yielding the meta-model components as provided in Fig. 3.4. We discuss the scenario in the light of the following research questions:

- **RQ1**: Is component encapsulation suitable to facilitate independent editing while ensuring consistency of model components?
- **RQ2**: Is the same language-level encapsulation useful when applied to different application scenarios?

First, we demonstrate the scenario in detail, considering a sequence of six revisions during collaborative development. Second, we discuss observations concerning the research questions.

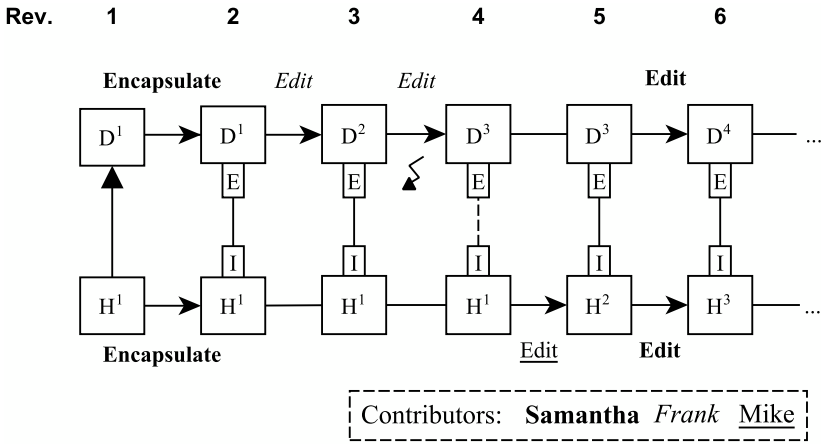

Figure 3.17: Overview of the application scenario, indices denoting revision numbers of the individual models.

Demonstration

The scenario, shown in Fig. 3.17, features three team members, Samantha, Frank, and Mike. Samantha is the team manager; she assigns the remaining team members to viewpoints: Frank becomes the domain modeler. Mike becomes the hypertext modeler.

The starting point of the scenario are existing data and hypertext models D and H, being connected using remote references. In the first revision, Samantha encapsulates along the given language-level encapsulation, replacing the remote references with export and import interfaces. The internal details of remote components are hidden to both developers respectively: Frank's scope of visibility is restricted to his assigned component being D. Mike's scope of visibility comprises his assigned component being H, and remote component D's export (cf. the notion of weak composite graphs introduced in Sec. 3.4).

Samantha, Frank, and Mike perform a series of asynchronous and synchronous editing steps, reflected in increasing revision numbers. One editing step introduces an inconsistency, being reconciled in a subsequent revision. We consider all six revisions in detail.

Revision 1. (Fig. 3.18) This revision comprises the initial state of the involved models. We show the revisions in a textual notation, which provides an alternative concrete syntax to the one shown earlier. As in-

```
1. data VehicleRentalData {
2.   entity Customer {
3.     att name : String
4.     ref creditCard : CreditCard
5.     ref address : Address }
6.   entity Address {
7.     att street : String
8.     att postalCode : Integer
9.     att city : String }
10.  entity CreditCard {
11.    att number : String }
12.  entity Car {
13.    att type : String }
14. }
```

```
1. hypertext VehicleRentalHypertext {
2.   index page carindex shows
       VehicleRentalData.Car {
3.     link to page cardata }
4.   data page cardata shows
       VehicleRentalData.Car { }
5.   index page customerindex shows
       VehicleRentalData.Customer { }
6.   index page creditcardindex shows
       VehicleRentalData.CreditCard { }
7.   static page indexpage {
8.     link to page carindex
9.     link to page customerindex
10.    link to page creditcardindex }
11.  start page is indexpage
12. }
```

Figure 3.18: Revision 1: Models with remote references.

Figure 3.19: Revision 2: Models after *component encapsulation*.

dicated by the four occasions in the hypertext model where model element names are prefixed with the name of the data model *VehicleRentalData*, the two models are connected using remote references.

Revision 2. (Fig. 3.19) As triggered by Samantha, component encapsulation of the models along the encapsulation of the DSMLs has been performed. Each model now contains a dedicated interface: The data model contains an export interface. The hypertext model contains an import interface referencing this export interface. The model elements exchanged between these interfaces are named explicitly. The remote references have been replaced by plain references to the imported elements.

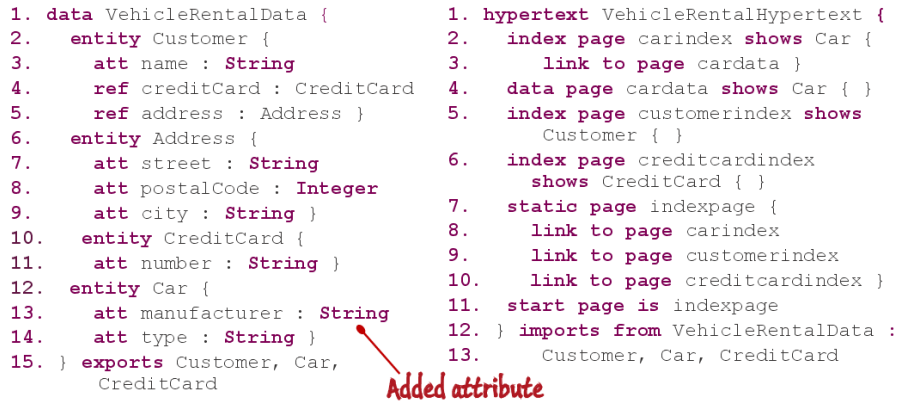

Figure 3.20: Revision 3: Models after performing *add attribute*.

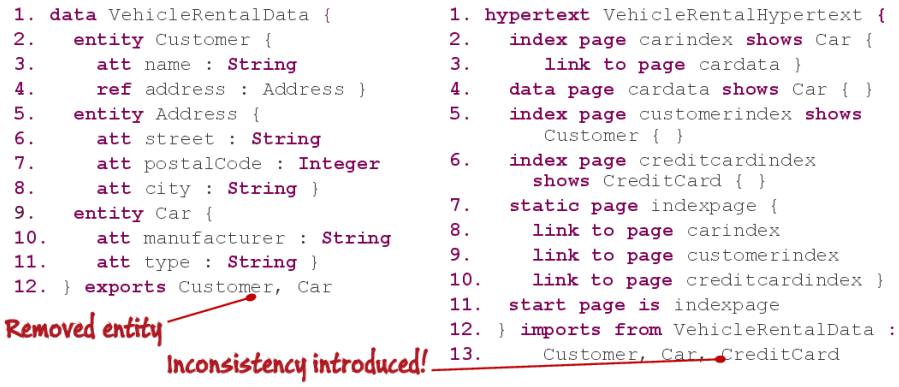

Figure 3.21: Revision 4: Models after performing *remove entity*.

Revision 3. (Fig. 3.20) Frank has introduced a new attribute for maintaining the manufacturer of a car, an asynchronous editing step only affecting one component. This step is neutral to inter-model consistency and does not require conflict handling.

Revision 4. (Fig. 3.21) As required by a law change prohibiting the storage of credit card information, Frank has removed the entity *CreditCard* from the data model. Since this editing step involves the deletion of an exported model element, it is a critical editing step threatening consistency. Frank receives a warning. His options are: to manually establish communication to Mike clarifying the change, to let a default message be delivered to Mike, to take back the change or to do nothing. In the two former cases, Mike can react by performing an editing step to retain consistency.

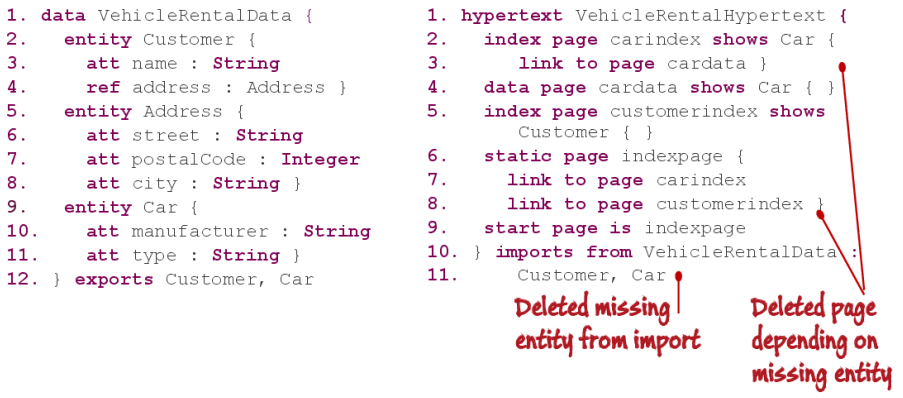

Figure 3.22: Revision 5: Models after reconciling the inconsistency introduced in Revision 4.

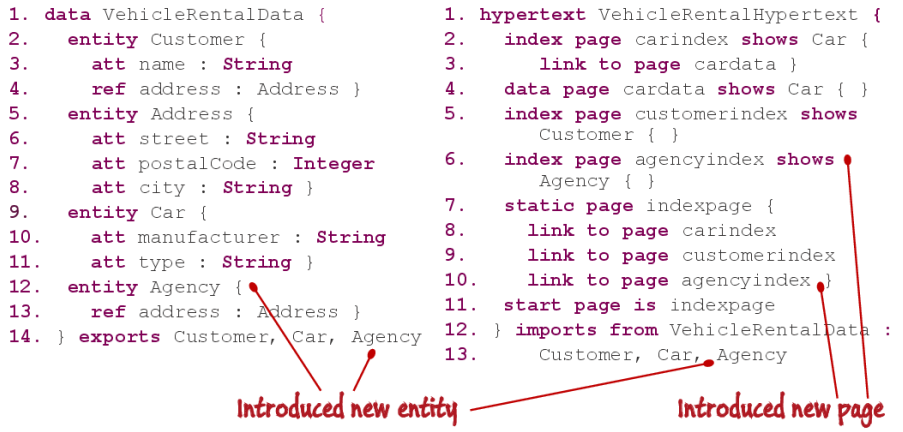

Figure 3.23: Revision 6: Models after performing *add corresponding entity and index page*.

Revision 5. (Fig. 3.22) Frank has decided to inform Mike about the change, so that Mike has been able to react immediately. Mike has removed the page that shows the credit card information from the hypertext model.

Revision 6. (Fig. 3.23) To support the management of agencies, a new requirement requested by the customer, Samantha has performed a synchronous editing step changing both components in parallel. Eventually, Samantha decides that the model components have accomplished a stable state and should be used for code generation. A global consistency check may be performed before the code generation to ensure a valid result. If at some later point in time new requirements are added, the components can be further evolved.

Discussion

Below, we discuss the two research questions outlined in the beginning of this section.

RQ1: Is component encapsulation suitable to facilitate independent editing while ensuring consistency of model components?

In accordance with the original motivation, the maintenance of explicit export and import interfaces supported a smart and relaxed conflict avoidance in this scenario: At all times, developers were aware whether their editing is either safe or critical to inter-model consistency. In the case of critical steps, further intervention became necessary. An automatized conflict detection and resolution algorithm can be considered complementary and might be applied at any time throughout the development process. Especially a conflict detection step right before the code generation step is highly desirable.

RQ2: Is the same language-level encapsulation useful when applied to different application scenarios?

The language encapsulation of the *SwalData* and *SwalHypertext* metamodels, originally introduced in Fig. 3.4 together with a different example model to illustrate the approach, was sufficient to enable consistent editing most of the time while helping to reconcile an inconsistency immediately in one occasion.

Threats to validity and limitations

In this section, we considered only one scenario for the proposed process, a potential threat to external validity. In addition, the considered scenario is an artificial one. A more diverse, preferably empirical validation is required to ensure the utility of our approach in realistic settings.

A threat to construct validity involves our notion of "usefulness" of the language-level encapsulation. As one caveat, we identify the sit-

uation where the application model developers need to inspect additional model elements to understand the context of imported elements. For instance, in order to understand the intention of a specific class, it might be reasonable to introduce references and attributes as well. This requirement can only be satisfied by extending the language-level encapsulation.

3.9 Related Work

Modularization Techniques. Amálio et al. [3] proposed Fragmenta, a theory for introducing modularity in models based on the notion of *fragments*. Fragments are small models that can be combined into larger ones by constructing *clusters* of fragments. Relations between fragments are established using proxy objects. The resulting formalization can be considered a formal underpinning of the proxy-based modularization strategy provided by the Eclipse Modeling Framework [106]. In contrast, the idea of component encapsulation is to *replace* this modularization strategy by the refined one of composite models, accounting for engineering principles such as information hiding.

In [131], Weisemöller et al. propose to extend meta-models by export and import interfaces to introduce information hiding. This work can be considered a predecessor of composite modeling, notably, since it is also based on concepts from algebraic graph transformation such as graphs and graph morphisms. Yet, this work does not consider concrete models based on the extended meta-models and does not provide a technique to introduce modularity automatically.

Distributed Editing. As for distributed editing, an elaborated strategy for inconsistency detection is provided by Macromodeling [97]. Macromodeling allows integrating multiple models of different modeling languages on type and instance layers. A major objective of macromodeling is the check of global consistency conditions based on logical formulas. Existing model editors such as Papyrus[2] or MagicDraw[3] implement a strategy for inconsistency avoidance by locking selected model parts for modification. Property locking as proposed by Chechik at al. [20] is a generalization of this approach. Instead of locking specific portions of the model, locks are assigned to properties, which can be

[2] `http://www.eclipse.org/papyrus/`
[3] `http://www.nomagic.com/products/magicdraw.html`

expressed by logical formulas over the model. These approaches assume a scenario where all involved modelers in principle have access to the complete involved model, while the developers in the scenarios considered in this chapter have differing scopes of visibility.

Connected Data Objects (CDO)[4] is a collaboration framework facilitating collaborative editing on top of EMF. As CDO does not provide support for explicit interfaces, it is orthogonal to composite modeling. An integration of composites with CDO exists.

3.10 Conclusion

In this chapter, we propose *component encapsulation*, an automated technique that allows deriving export and import interfaces for a set of related models automatically, thereby turning it into a set of model components. To demonstrate that the technique enables a disciplined development workflow for models created using domain-specific modeling languages (DSMLs), we embed it in the context of a systematic modeling process. By allowing to establish sound engineering principles such as *information hiding* and a notion of *visibility*, the proposed technique enables a useful improvement in the state-of-the-art of DSML development.

[4] `http://projects.eclipse.org/projects/modeling.emf.cdo`

Chapter 4

Rule Merging for Variability-Based Model Transformation

*This chapter shares material with the FASE'15 paper "A Variability-Based Approach to Reusable and Efficient Model Transformations" [116] and the FASE'16 paper "*RuleMerger*: Automatic Construction of Variability-Based Model Transformation Rules" [115].*

In this chapter, we tackle the maintainability- and performance-related scalability issues of model transformations by introducing *variability-based model transformation*. We formally define variability-based rules and contribute a novel match-finding algorithm for applying them. We introduce rule merging, a three-component framework that allows creating efficient variability-based rules automatically. We prove correctness of this approach by showing the equivalence of the created rules to their classical counterparts and demonstrate its benefits in two realistic transformation scenarios.

4.1 Introduction

Model transformation is a key enabling technology for Model-Driven Engineering, pervasive in all of its activities, including the translation, optimization, and synchronization of models [101]. Algebraic graph transformation (AGT) is one of the main paradigms in model transformation. It allows the specification of rules in a high-level, declarative

manner [27]. Recently, many complex transformations have been implemented using AGT [46, 71, 35]. AGT is gaining further importance due to its use as an analysis back-end for imperative model transformation languages [89].

Transformation systems often contain rules that are substantially similar to each other. Yet, until recently, various model transformation languages lacked constructs suited to capture these similar *rule variants* in a compact manner [63]. The most frequently applied mechanism for creating variants remains cloning: developers produce rules by copying and modifying existing ones, a process related to various drawbacks for the maintainability and performance of the resulting rule sets.

In this chapter, we introduce *variability-based model transformation*, an approach inspired by product line engineering (PLE) principles [21, 26]. A variability-based (VB) rule encodes a set of similar variants in a single-copy representation, explicating their common and variable portions. By introducing VB rules into a transformation system, the drawbacks of cloning as outlined Sec. 1.1.3 can be addressed:

- Representing a set of mutually similar rules in a single-copy representation is promising to improve their *maintainability*: It is considerably easier to maintain consistency between rules if the developer is not required to perform the same change in each rule variant individually. This approach is also less prone to subtle errors introduced during rule creation by cloning, where errors are copied and must be fixed in all subsequently created rules. Finally, the effort to maintain the overall rule system may be smaller when the overall number of rules is smaller.
- To improve the *performance* of the transformation engine during execution of these rules in batch mode, we introduce a novel algorithm for resolving variability automatically during the rule matching process, i.e., determination of application sites in the input model. Our central idea is to find matches for the common parts of all rule variants first and then to use them as starting points for the matching of the variable parts.

Despite the apparent benefits of its outcome, creating VB rules manually is a tedious and error-prone task. This task relies on the precise identification of (i) the set of rules that should be unified into a single VB rules; (ii) rule portions that should be merged versus portions that should remain isolated. The choices made during these steps have a substantial impact on the quality of the produced rules.

We introduce *rule merging* as an approach to automate this task. Rule merging comprises a three-component framework: It applies *clone detection* to identify overlapping fragments and *clustering* to assign rules to groups of mutually similar rules. A third component called *merge construction* selects the overlapping portions used as basis for creating the output VB rules. Each of these components can be instantiated and customized in various ways, accounting for user-specified quality goals.

We make the following contributions:

- We provide a formalization of variability-based rules, investigating their syntax and application semantics on the basis of graph transformation. We prove equivalence to the application of the corresponding classic rules.
- We propose a match-finding algorithm aiming to achieve a performance gain when compared to matching the rules individually.
- We present a novel merge-refactoring approach for AGT-based model transformation rules. Our instantiation of the approach provides a novel *merge construction* algorithm and harnesses state-of-the-art clone detection and clustering techniques.
- We formally prove the correctness of the approach, showing the equivalence of the produced VB rules to their classical counterparts.
- We empirically show that the produced VB rules are superior to their classical counterparts in terms of several performance- and maintainability-related characteristics.

The remainder of this chapter is structured as follows: In Sec. 4.2, we give an overview of the approach and demonstrate two motivating scenarios. In Sec. 4.3, we recall the necessary background required by the approach. In Sec. 4.4, we define the foundations of variability-based model transformation. We investigate variability-based transformation formally and describe the algorithm for directly applying variability-based transformations. In Sec. 4.5, we outline the approach and argue for its correctness. Sec. 4.6 reports on our instantiation of rule merging. In Sec. 4.7, we discuss our implementation in detail. Sec. 4.8 presents our evaluation. In Sec. 4.9 and 4.10, we discuss related work and conclude.

4.2 Overview

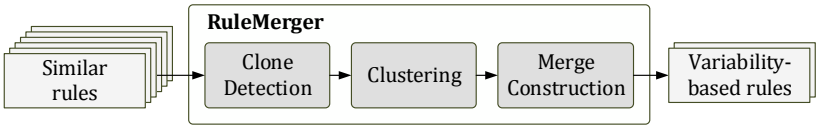

Figure 4.1: An overview of rule merging.

In this section, we present rule merging, a novel approach for automating the merge-refactoring of model transformation rules. The approach includes three components (see Fig. 4.1). It applies *clone detection* [91] to identify overlapping portions between rules and *clustering* [133] to identify disjoint groups of similar rules. During *merge construction*, common portions are unified and variable ones are annotated to create variability-based (VB) rules. Each component can be instantiated and customized with respect to specific quality goals, e.g., to produce rules optimized for background execution or easy editing. Since the framework guarantees that all created rule sets are semantically equivalent, we envision a system that enables users to edit rules in a convenient representation and automatically derives a highly efficient one.

The distinguishing factors of this approach, compared to merge-refactoring approaches in the PLE domain [92, 137, 96], are its ability to detect overlapping *portions* rather than pairs of similar elements and to create *multiple* output VB rules rather than one single-copy representation of all rules. These factors allow us to address the performance and maintainability issues related to cloning.

4.2.1 Examples

Example 1: Variability-intensive refactoring rules.

Consider a set of model transformation rules aiming to improve the structure of an existing code base based on *class model refactorings*. Fig. 4.2 shows six refactoring rules expressed in an abstract syntax notation [27]. The rules describe several ways of relocating a method between two classes. Each rule comprises two parts called left-hand side (LHS) and right-hand side (RHS): if the LHS matches a place in the input model, then the RHS is applied, thereby changing the model. We present the rules in an integrated form, with the LHS and RHS of a rule being represented in one graph. The LHS comprises all *delete* and *preserve* objects. The RHS contains all *preserve* and *create* objects.

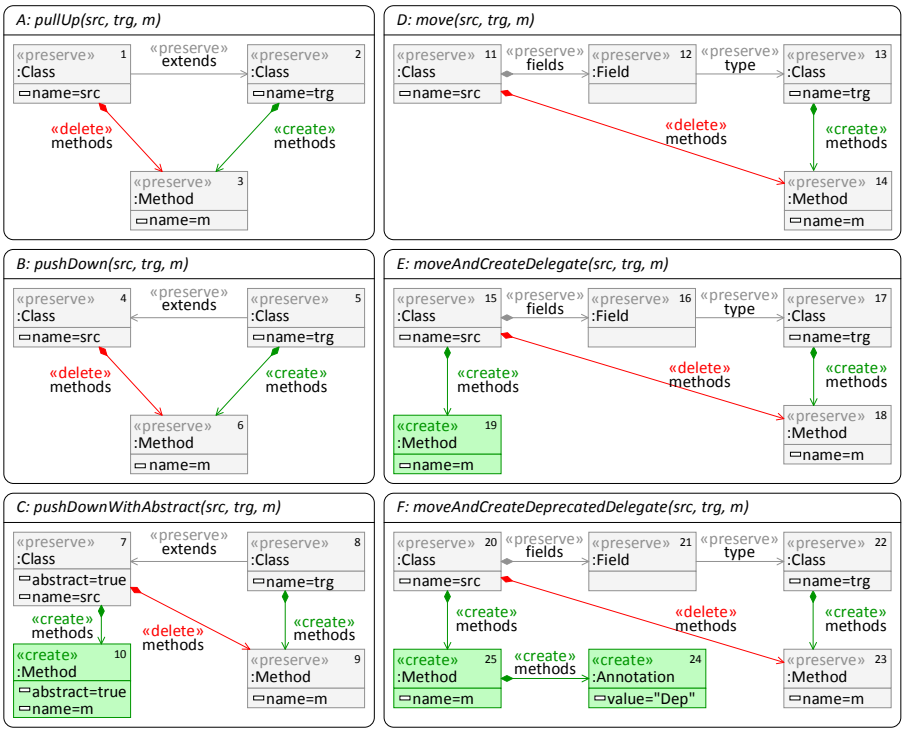

Figure 4.2: Refactoring rules for class models.

Rule A takes as input two classes, one of them sub-classing the other, and a method. Each of these input objects is specified by its name. The rule moves the method from a sub-class to its super-class, by deleting it from the sub-class and adding it to the super-class. Similarly, Rule B moves a method from the super-class to one of its sub-classes. Rule C also moves a method from the super- to a sub-class, but, in addition, creates an abstract method with the same name in the super-class. Rules D, E, and F move a method across an association. The latter two rules also create a "wrapper" method of the same name in the source class. Rule F uses an annotation to mark this method as deprecated. Such rule sets are often created by cloning, that is, copying a rule and modifying it to fit the new purpose.

We consider the merge-refactoring of a rule set created using cloning. The result is a rule set with variability-based (VB) rules in which the common portions are unified and the differences are explicated, as shown in Fig. 4.3. Specifically, Rules B and C are merged, producing a new VB Rule B+C. Rules D, E, and F are merged into D+E+F. Rule A remains as is.

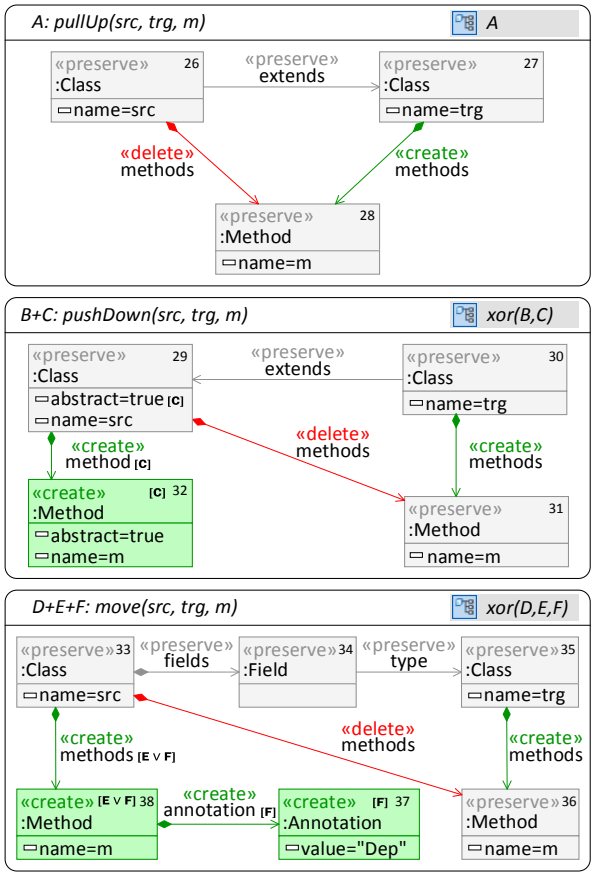

Figure 4.3: Variability-based refactoring rules for class models.

Each VB rule has a set of *variation points*, corresponding to the names of the input rules: Rule B+C has the variation points *B* and *C*. In addition, each rule has a *variability model* specifying relations between variation points, such as mutual exclusion: Rule B+C has the variability model *xor(B,C)*. VB rules are *configured* by binding each variation point to either *true* or *false*. Portions of VB rules are annotated with *presence conditions*. These portions are removed if the presence condition evaluates to *false* for the given configuration. Element #32 and its incoming edge are removed in the configuration {*C*=*false*, *B*=*true*}. For conciseness, we omit the presence condition *true*, e.g., for nodes #26-#28.

In this example, the VB rules are configured individually, either manually by the user or automatically by client code, e.g., a model editor. The result of configuration is a "flat" rule again – a process similar to that in product-line engineering approaches [26]. Alternatively, *all* rules of a rule set may be executed in *batch mode*, i.e., considered simultaneously. The following example deals with such a scenario.

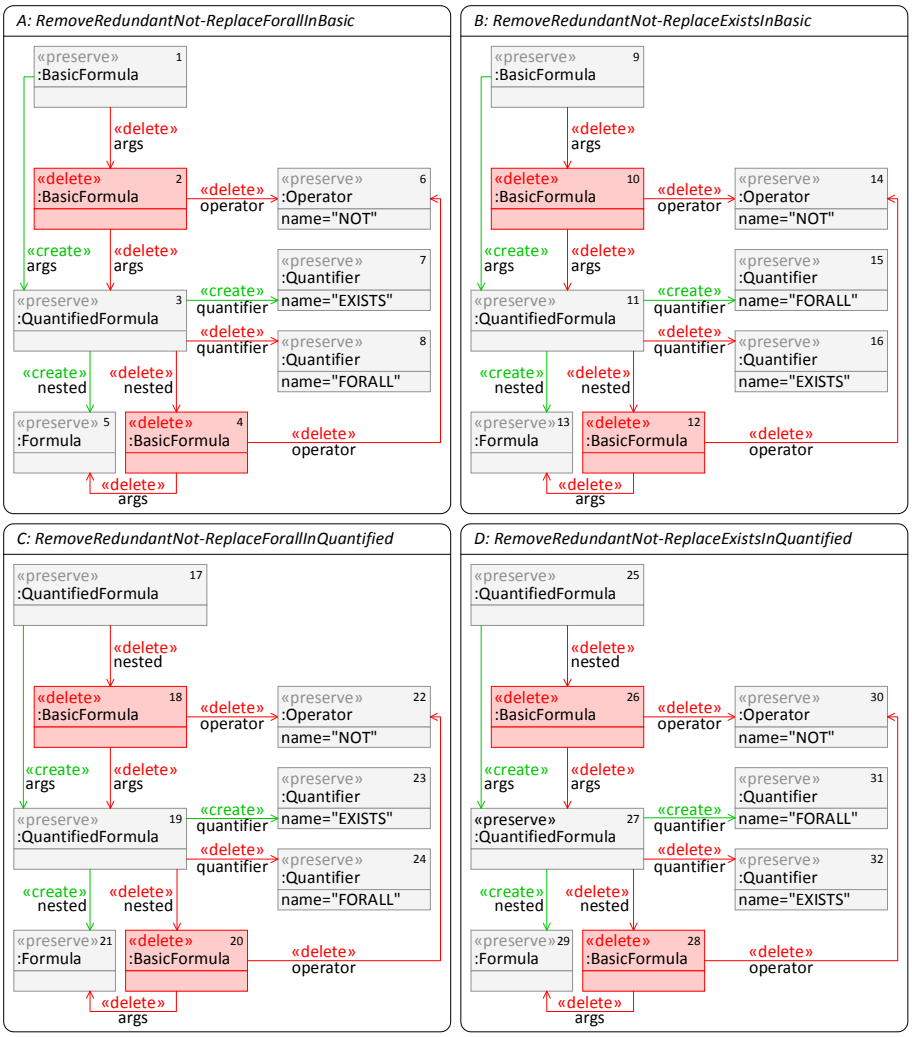

Figure 4.4: *Remove Double Negation* optimization rules.

Example 2: Variability-intensive optimization rules.

Consider a second example inspired by a set of real-life rules for optimizing and simplifying first-order logic expressions [7], aimed to improve performance of engines that process these expressions, e.g., theorem provers or SAT solvers. Fig. 4.4 shows four transformation rules that simplify first-order logic formulas by removing redundant *not* symbols and thus reducing the "depth" of a formula. Again, we present the rules in an integrated form, with the left- and right-hand sides of the transformation being represented in a graph labeled with the actions *preserve*, *delete*, and *create*.

Rule A removes a $\neg\forall\neg$ segment of a formula and transforms it into an $\exists$ segment. This is done by removing nodes #2, #4 and their corresponding edges, replacing the quantifier of node #3 to be "exists" (node #7) rather than "forall" (node #8), and connecting the modified quantifier to the enclosing and the enclosed formulas – nodes #1 and #5, respectively. Similarly, Rule B removes a $\neg\exists\neg$ segment and transforms it into a $\forall$ segment. Rules C and D differ from A and B in the type and adjacent edges of the topmost enclosing formula (nodes #1,9,17,25): basic vs. quantified. A BasicFormula has an operator and a set of argument formulas, whereas a QuantifiedFormula has a quantifier and nests exactly one other formula. Note that there exists a third kind of formula, PredicateFormula, that encloses no other formulas.

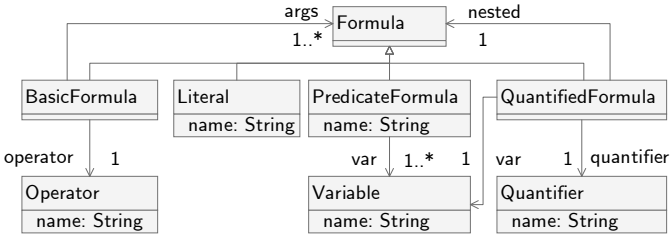

Figure 4.5: Meta-model for first-order logic formulas.

Fig. 4.5 provides the meta-model for first-order logic formulas as used by these model transformation rules. Formulas are expressed as a set of atomic formulas – being either named predicate formulas or literals – that are combined through universal and existential quantifiers and Boolean operators. There are four kinds of formulas:

- Literals, being either *TRUE* or *FALSE*,
- Predicate formulas, comprising a name and a set of variables (e.g. name *F* and variable *X* constituting predicate formula *F(X)*),
- Basic formulas, linking a set of argument formulas through an operator (such as *AND*, *OR*, *NOT*), and
- Quantified formulas, nesting a formula while binding one of its variables to a quantifier (*EXISTS* or *FORALL*).

Fig. 4.6 shows an example first-order logic formula $\phi = (\neg\forall x \cdot \neg F(x)) \wedge$ *true* that can be simplified using one of the rules, namely, Rule A. The formula is represented as a model created using the meta-model shown in Fig. 4.5. The left-hand side of the figure depicts formula-specific elements. The right-hand side presents a library of "generic" reusable

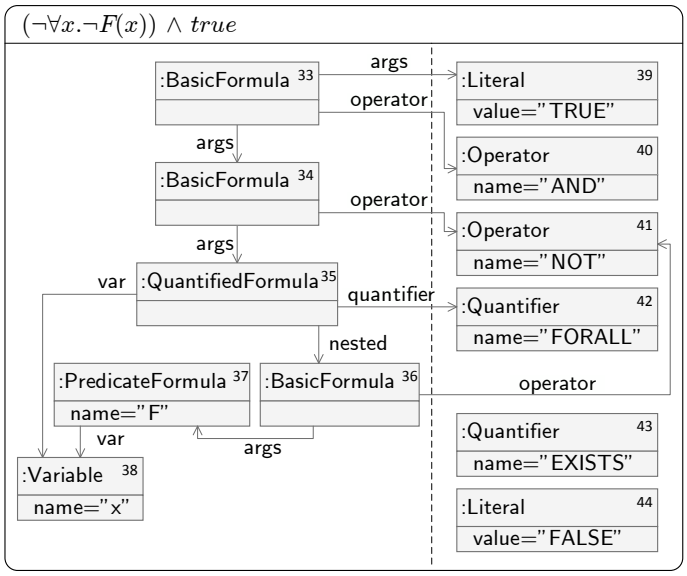

Figure 4.6: Example first-order logic formula ϕ.

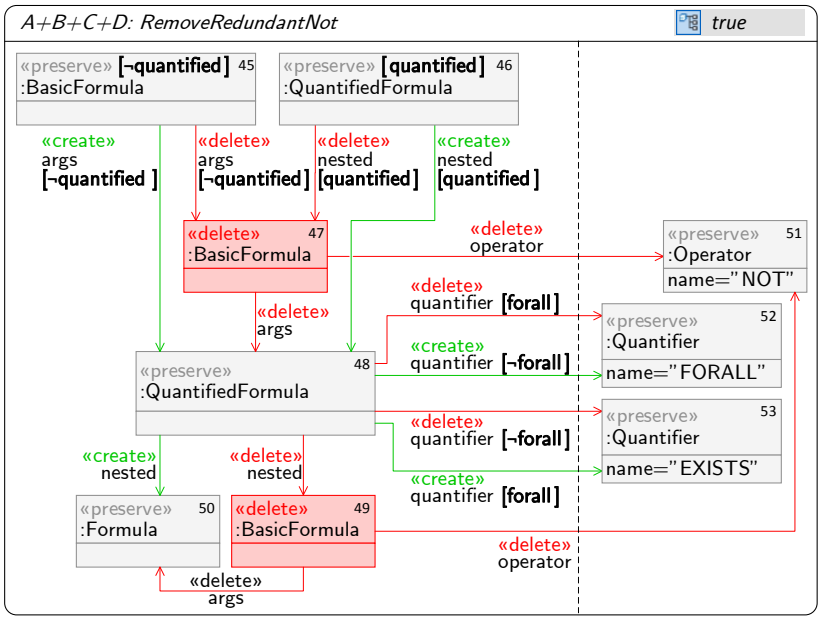

Figure 4.7: Variability-based *Remove Double Negation* optimization rule.

first-order logic operators. Elements #1-#8 of Rule A match with the corresponding elements #33–#37, #41, #43, #42. We call this assignment a match m_A. Finding m_A triggers the application of Rule A, producing the formula $(\exists x \cdot F(x)) \wedge true$. Note that m_A is a valid match because PredicateFormula (node #5 in Fig. 4.6) is a sub-type of Formula.

Fig. 4.7 shows a compact *variability-based rule* that represents all four individual rules in Fig. 4.4. The differences between the classic rules are explicitly captured and represented by *variation points*. Rule elements are *annotated* with *presence conditions* – Boolean formulas over the variation points. In the visual representation, annotations are appended in square brackets to the names of their corresponding nodes and edges. Again, for the simplicity of presentation, we omit the presence condition *true*, e.g., for nodes #2-#8.

In this example, there are two variation points: (1) The *forall* variation point controls the direction of the quantifier inversion. When set to *true*, it corresponds to the $\neg\forall\neg$ to $\exists$ inversion, as in rules A and C; when set to *false*, it corresponds to the $\neg\exists\neg$ to $\forall$ inversion, as in B and D. (2) The *quantified* variation point controls the enclosing formula and its adjacent edges. When set to *true*, it corresponds to a formula of the type QuantifiedFormula with outgoing nested edges, as in C and D; when set to *false*, it corresponds to a formula of the type BasicFormula with outgoing args edges, as in rules A and B. Note that this variation point cannot be captured using node sub-typing, as it affects edges with different types. The variability model in this example is *true* since all configurations obtained by binding configuration points to *true* or *false* are valid.

A variability-based rule can be *configured* by setting variation point values and then selecting all elements whose presence conditions evaluate to *true* while removing those whose presence conditions evaluate to *false*. In our example, configuring the rule with {*forall=true*; *quantified=false*} produces Rule A in Fig. 4.4 while the configuration {*forall=false*; *quantified=true*} produces Rule D.

4.2.2 Automated Technique

In this chapter, we introduce rule merging, an automated technique that takes as input a set of mutually set of similar rules such as the ones shown in Fig. 4.2 or Fig. 4.4. and produces a representation using VB rules such as illustrated in Fig. 4.3 or Fig. 4.7.

Effect on maintainability. Conceptually, a variability-based rule is equivalent to a set of rules for all its valid configurations. Yet both examples demonstrate several benefits of VB rules related to maintainability: The amount of redundancy is reduced, ensuring consistency between variants during changes; subtle error produced during

rule creation are fixed in one place. The total number of rules is smaller, possibly allowing to navigate the transformation system with smaller cognitive effort for the developer and decreased computational effort for the rule editor. The latter may increase responsiveness of the editor and thereby help the developer focus on the maintenance task at hand.

Effect on performance. The match-finding algorithm for a variability-based rule proposed in this chapter performs matching of all its valid configurations at once, thus positively affecting the performance of the transformation system. The algorithm *automatically* detects a configuration that induces a valid match using a two-step process. In the first step, it matches the *base rule* – the portion of the rule annotated with *true* and representing common parts of all individual rules. For the example in Figs. 4.7 and 4.6, this results in exactly one match, m_{base}, assigning elements #47-#53 to #34-37,41-43 and connecting edges accordingly. In the second step, the algorithm enumerates the valid configurations and tries to match them using m_{base}. This yields exactly one match for Rule A: m_A. The result of match-finding is m_A paired with the configuration {*forall=true*; *quantified=false*} enabling m_A.

4.3 Preliminaries

As preliminaries for variability-based transformation, we briefly revisit the fundamental concepts of algebraic graph transformation [32] as outlined in Section 2.4. Graphs can be used to represent the underlying structure of visual models. Their conformance to a meta-model can be formally represented by typed attributed graphs mapped to type graphs. For simplicity, our treatment here uses basic graphs without types and attributes. However, since the introduced notion of variability is orthogonal to these features, our implementation and evaluation use the full power of typed attributed graphs with inheritance [45].

A graph comprises a set of nodes and a set of edges connecting these nodes. Structure-compatible mappings between graphs can be expressed in terms of *graph morphisms* which are compatible to the source and target functions for the edges. Graph transformations are expressed using rules. For example, in the *gluing approach* to graph transformation, graph elements occurring in the left- and right-hand sides of a rule, i.e., in the *interface graph*, are used to glue new elements to already existing ones. Rule A from Fig. 4.4 is a rule that can be applied to the typed attributed graph of Fig. 4.6 in order to transform it.

A rule is applied using a match, a morphism of its left-hand side to the input graph. The application proceeds in two steps: First, all specified deletions are performed. By checking the gluing condition, it is ensured that no dangling edges are produced. Then, the specified modifications and creations are performed. In the example, the example match m_A comprises mappings of elements #1-#8 of Rule A to elements #33–#37, #41, #43, #42. By the application of Rule A, nodes #34 and #36 are deleted, together with all of their incoming and outgoing edges. The edge between nodes #35 and #42 is deleted as well. As no dangling edges are left behind, the gluing condition is satisfied. Edges between #33 and #35, #35 and #37, as well as between #35 and #43 are created yielding the graph structure for the formula $\phi' = (\exists x \cdot F(x)) \wedge \mathit{true}$.

We further consider connected rules, rules being constituted by connected left-hand and right-hand side graphs.

Definition 17 (Connected rule)**.** *A rule $r = L \xleftarrow{le} I \xrightarrow{ri} R$ is* connected *iff, treating all edges as undirected, $\forall G \in \{L, R\}$ there is a path between each pair of nodes in G.*

For example, none of the rules in Fig. 4.4 is connected, since element #6 and its counterparts isolated in their R graphs.

We define an operation called *flattening* based on multi-pullbacks and -pushouts [70], categorical constructions generalizing the intersection and union of objects. The category SETS is complete and co-complete, i.e., all limits and co-limits, specifically multi-pullbacks and -pushouts, exist [33]. The same holds for GRAPHS, since it is a comma category of SETS [41]. The category $\text{GRAPHS}_{TG'}$ of typed graphs and total clan morphisms *without type refinement* is co-complete since pushouts exist and the empty typed graph is an initial object [33]. We do not need type refinement for variability-based rules. Moreover, we show that the multi-pullback in GRAPHS is well-typed, i.e., it is an object in $\text{GRAPHS}_{TG'}$. Note that this object is not necessarily a multi-pullback in $\text{GRAPHS}_{TG'}$.

Proposition 4. *Let a type graph TG, a graph G_r and a set of graphs $\mathcal{G} = (G_i | i \in I)$ with a family of morphisms $g'_i = G_i \to G_r$ be given. Further, let G_r and each $G \in \mathcal{G}$ be typed graphs over TG. For multi-pullback G_t (see Fig. 4.8), there exists a well-defined and unique type morphism $type_t : G_t \to TG$.*

Proof. The type morphism $type_t$ exists and is uniquely determined: $\forall i, j$ with $1 \leq i, j \leq n$, we have $type_i \circ g_i = type_r \circ g'_i \circ g_i = type_r \circ g'_j \circ g_j = type_j \circ g_j$. □

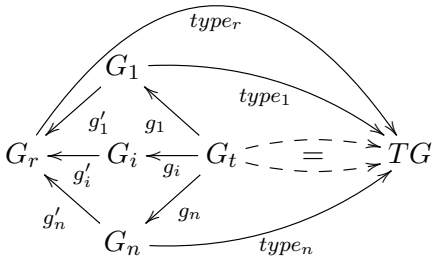

Figure 4.8: Typing of the multi-pullback in GRAPHS with $1 \leq i \leq n$.

4.4 Variability-Based Model Transformation

In this section, we introduce variability-based transformation rules and show how to apply them.

4.4.1 Variability-Based Rules

We denote variability using variability expressions, propositional expressions over a set of *variation points*. We consider two kinds of variability expressions: The variability model and variability conditions. The variability model is used to express relationships between variation points, such as mutual exclusion or implication. Variability conditions are used to specify when specific variants expressed using variation points shall be active. The set of variation points and the variability model are fixed for the set of rules and not changed by transformation steps.

A *subrule* encapsulates a subset of actions on a substructure of a set of rules. If we want to identify substructures of the same rule, we talk about subrule *embeddings*. To identify common substructures of multiple rules, we talk about *subrule morphisms*.

Definition 18 (Subrule morphism)**.** *Given a pair of rules* $r_0 = (L_0 \xleftarrow{le_0} I_0 \xrightarrow{ri_0} R_0)$ *and* $r_1 = (L_1 \xleftarrow{le_1} I_1 \xrightarrow{ri_1} R_1)$ *(Def. 12) with injective mappings* le_i, ri_i *for* $i \in \{0, 1\}$*, a* subrule morphism $s : r_0 \to r_1$*,* $s = (s_L, s_I, s_R)$ *consists of injective mappings* $s_L : L_0 \to L_1$*,* $s_I : I_0 \to I_1$*, and* $s_R : R_0 \to R_1$ *s.t. in the diagram in Fig. 4.9* (1) *and* (2) *commute. In addition,*

- *the intersection of* $s_L(L_0)$ *and* $le_1(I_1)$ *in* L_1 *is isomorphic to* I_0*.*
- *the intersection of* $s_R(R_0)$ *and* $ri_1(I_1)$ *in* R_1 *is isomorphic to* I_0*.*
- $L_1 - (s_L(L_0) - s_L(le_0(I_0)))$ *is a graph (Def. 1).*

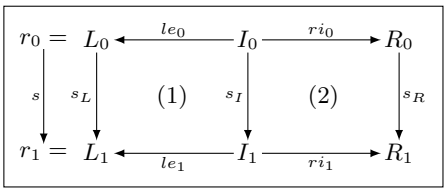

Figure 4.9: A schematic depiction of subrule morphisms.

The conditions prefaced by "in addition" ensure that a subrule performs the same actions on related elements as the original rule and that the larger pattern of the original rule does not prevent a subrule to be applied.

Definition 19 (Subrule embedding)**.** *Subrule mapping s is called a* subrule embedding *if all of its morphisms s_L, s_I, and s_R are inclusions. Given two subrule embeddings $s : r_0 \to r_1$ and $s' : r'_0 \to r'_1$, we have that $s \subseteq s'$ if there are subrule embeddings $t_0 : r_0 \to r'_0$ and $t_1 : r_1 \to r'_1$ with $s' \circ t_0 = t_1 \circ s$.*

In the example rule shown in Fig. 4.7, the rule comprising all nodes and edges with no visible presence condition, i.e., #47–53 and their interrelating edges, is a subrule of the entire rule.

Definition 20 (Language of variability expressions)**.** *Given a set of variation points V, $\mathcal{L}_V$ is the set of all propositional expressions over V, called* language of variability expressions*.*

In the example, *forall* and *quantified* are variation points. The language of variability expressions includes the words *forall* $\wedge$ *quantified*, $\neg$ *forall*, and *true*.

Definition 21 (Variability model)**.** *Given a language of variability expressions $\mathcal{L}_V$, a variability model vm is an element of $\mathcal{L}_V$. A total function cfg : $V \to \{true, false\}$ is a* variability configuration*. A variability configuration cfg is* valid *wrt. to a given variability model vm iff if vm evaluates to true when each variable v in vm is substituted by cfg(v).*

In the example, the *forall* and *quantified* variation points are orthogonal, hence, the variability model is *true* and all four possible configurations are valid. If Rule C did not exist, *forall* and *quantified* might be mutually exclusive. The variability model would then be *forall xor quantified*, rendering the configuration *{forall=true, quantified=true}* invalid.

Definition 22 (Variability condition)**.** *Given a language of variability expressions $\mathcal{L}_V$ and a variability model vm, a* variability condition *is an element of $\mathcal{L}_V$, i.e., a propositional expression over V. A variability configuration*

cfg satisfies *a variability condition* vc *if* vc *evaluates to true when each variable* v *in* vc *is substituted by* $cfg(v)$. *A variability condition is* valid *if there is a valid variability configuration satisfying it. A variability condition* X *is* stronger *than* Y *iff* $X \implies Y$.

In the example, $V = \{forall, quantified\}$. Valid variability conditions include *true*, $\neg$*quantified*, and *forall* $\wedge$ *quantified*; *forall* $\wedge$ $\neg$*forall* is not valid.

Definition 23 (Variability-based (VB) rule)**.** *Given* $\mathcal{L}_V$, *a* VB rule $\check{r} = (r, S, v, vc)$ *consists of a rule* r, *a set* S *of subrule embeddings to* r, *a variability model* v *and a function* $vc : S \cup \{r\} \to \mathcal{L}_V$. *Function* vc *defines* variability conditions *for subrules s.t.* $vc(r)$ *is* true *and* $\forall s \subseteq s' : vc(s') \implies vc(s)$. *The* base rule *is determined by the intersection of all subrule embeddings.*

For example, Fig. 4.7 shows a VB rule in a compact representation: Elements instead of subrules are annotated. Rule r is the entire rule ignoring annotations, the variability model vm is *true*. Set S and function vc are derived easily by creating a subrule s_{conj} for each conjunction $conj = \bigwedge_{i \in V} l_i$ of literals over V, and setting $vc(s_{conj}) = conj$. Each subrule s_{conj} includes the elements whose presence condition is implicated by *conj*. For instance, subrule $s_{\neg quantified \wedge forall}$ comprises the elements annotated with $\neg$*quantified*, *forall*, and *true*, i.e., it is isomorphic to Rule A in Fig. 4.4.

We may further include arbitrary subrules in S. Doing so will be useful to us; it allows us to improve the performance of the application of VB rules. In the example, we add one additional rule to S: The base rule s_{true}, comprising all elements annotated *true*. We then have $S = \{s_{quantified \wedge forall}, s_{\neg quantified \wedge forall}, s_{quantified \wedge \neg forall}, s_{\neg quantified \wedge \neg forall}, s_{true}\}$.

Application of Variability-Based Rules

We now show how to apply variability-based rules: (1) either by flattening them to a set of classic rules and matching and applying these rules in the classic way, or (2) directly, using a suitable variability configuration to identify a corresponding match. We then prove the equivalence of these two approaches.

Variability-based transformation through flattening

We begin by showing how a variability-based rule can be *flattened*, i.e., represented by a set of classic rules.

Definition 24 (Configuration-induced rule)**.** *Let a VB rule* $\check{r} = (r, S, v, vc)$ *over* $\mathcal{L}_V$ *be given. For a valid variability configuration* c, *there*

exists a unique set of subrule embeddings $S_c \subseteq S$ s.t. $\forall s \in S : s \in S_c$ iff c satisfies $vc(s)$. The configuration-induced rule *r_c (Fig. 4.10) comprises multi-pushouts L_c, I_c and R_c over multi-pullbacks L_t, I_t and R_t over L_i, I_i and R_i of each rule r_i in S_c. The multi-pullback and -pushout constructions can be performed in* GRAPHS. *Prop. 4 and the co-completeness of* $\text{GRAPHS}_{TG'}$ *ensure that these objects are also contained in* $\text{GRAPHS}_{TG'}$. *Morphisms le_t and ri_t exist since L_t and R_t are limits, le_c and ri_c exist since I_c is a colimit. r_c can be embedded to r since L_c, I_c and R_c are colimits.*

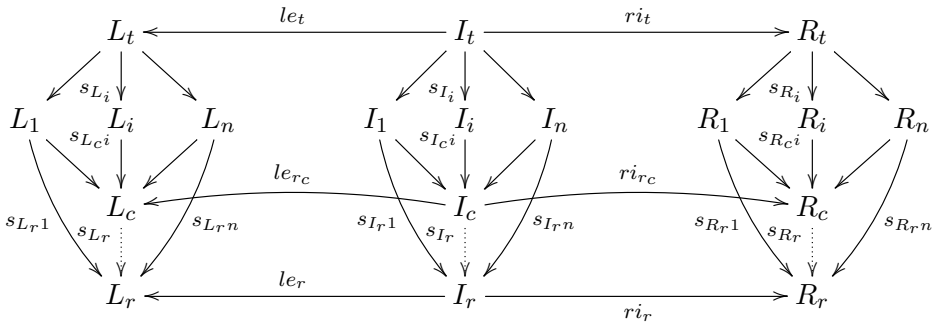

Figure 4.10: Construction of configuration-induced rule r_c for configuration c, $1 \leq i \leq n$ with $n = |S_c|$.

For example, for the rule in Fig. 4.7, configuration {*forall=true*, *quantified=true*} yields a configuration-induced rule being isomorphic to Rule C in Fig. 4.4.

Definition 25 (Flattening of a VB rule)**.** *The flattening of $\check{r}$ is the set of all configuration-induced rules over all valid configurations: Flat($\check{r}$) = $\{r_c \mid c : V \to \{true, false\} \ \wedge \ c \text{ is valid}\}$.*

In the example, flattening the rule in Fig. 4.7 yields a set containing four rules isomorphic shown to the ones in Fig. 4.4.

Definition 26 (Application of a rule set)**.** *Given a rule set $\mathcal{R}$ and a graph G, the application of $\mathcal{R}$ to G is the set of rule applications: Trans($\mathcal{R}, G$) = $\{G \Rightarrow_{r,m} H\}$ with $r \in \mathcal{R}$ and a match $m : L \to G$ (Def. 13).*

For example, applying the flattening of the rule in Fig. 4.7 on the graph of formula ϕ in Fig. 4.6 yields a set of rule applications containing exactly one element: The application of the rule isomorphic to Rule A using the match m_{A}.

Direct application of variability-based rules.

In the following, we consider the direct application of variability-based rules by finding a suitable variability-based match on-the-fly. The cen-

tral task is to find variability configurations that induce a match for the left-hand side of one variant contained in the variability-based rule. If the resulting morphism of the left-hand side to graph G satisfies the gluing condition for the corresponding flat rule, the rule application can take place.

Definition 27 (Variability-based match family)**.** *Given a variability-based rule $\check{r}$ over $\mathcal{L}_V$, a graph G, and a valid variability configuration c, yielding the set of subrule embeddings $S_c \subseteq S$ s.t. $\forall s \in S : s \in S_c$ iff c satisfies $vc(s)$. A* variability-based match family *is a family of matches $m_{s \in S_c} : L_s \to G$ s.t. $\forall m_i, m_j \in m_s, 1 \leq i, j \leq |S_c|$ the following holds: $\forall x \in dom(m_i) \cap dom(m_j) : m_i(x) = m_j(x)$ (Fig. 4.11).*

The condition ensures that matches within a family are compatible: An element contained in multiple subrules is always mapped to the same element in graph G.

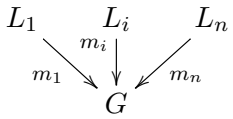

Figure 4.11: Variability-based match family, $1 \leq i \leq n$ with $n = |S_c|$.

This definition entails that the identification of matches can terminate as soon as one of the subrules cannot be matched, enabling a performance benefit of VB rule application: We can match the base rule s_{true} first and only need to go on if this subrule can be matched.

To match the rule in Fig. 4.7 to the graph for formula ϕ in Fig. 4.6, we choose the variability configuration *{quantified=false; forall=true}*, which yields the subrule embeddings s_{true} and $s_{\neg quantified \wedge forall}$. The left-hand side of each of this subrule embeddings can be matched to G, yielding a variability-based match family.

Definition 28 (Variability-based match)**.** *Given a variability-based match family $m_{s \in S_c}$ for a variability-based rule $\check{r}$, a configuration c and a graph G, a* variability-based match *$\check{m}$ is a pair (m_c, c) where $m : L_c \to G$ is obtained by the colimit property of L_c (Fig. 4.12).*

In the example, again considering the configuration *{quantified=false; forall=true}*, a VB match is obtained from considering the VB match family described after Def. 27 and gluing their mappings together. This VB match has the same mappings as m_A.

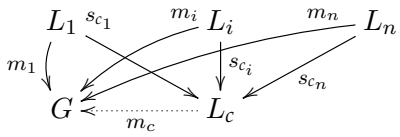

Figure 4.12: Variability-based match, $1 \leq i \leq n$ with $n = |S_c|$.

Definition 29 (Application of a variability-based rule)**.** *Given a match $\check{m} = (m_c, c)$ for variability-based rule $\check{r}$ and graph G, the* application of $\check{r}$ at $\check{m}$ *is the classic rule application $G \Rightarrow_{r_c,m_c} H$ of the configuration-induced rule r_c to m_c. To obtain all applications of $\check{r}$, we consider all variability-based matches: DirectTrans($\check{r}$, G) = $\{G \Rightarrow_{r_c,\check{m}} H \mid c$ is a valid configuration, $\check{m} = (m_c, c)$ is a variability-based match$\}$.*

For example, applying the rule in Fig. 4.7 to the graph of formula ϕ in Fig. 4.6 at the VB match with the same mappings as m_A yields the graph structure of formula ϕ' described at the end of Sec. 4.3.

Now, we show that the set of all applications of a variability-based rule $\check{r}$ to a graph G is equal to the set of classic rule applications obtained from flattening $\check{r}$ and applying these rules to G.

Theorem 1 (Equivalence of rule applications)**.** *Given a variability-based rule $\check{r}$ and a graph G, the following holds: DirectTrans($\check{r}$, G) = Trans(Flat($\check{r}$), G).*

Proof. Since *DirectTrans*($\check{p}$, G) and *Trans*(*Flat*($\check{r}$), G) are both constructed over all valid configurations, we can assume a particular valid configuration $c : V \rightarrow \{true, false\}$ without loss of generality.

From *Trans*($Flat(\check{r}), G$), we consider the rule application $G \Rightarrow_{r_c,m} H$ of a configuration-induced rule $r_c \in Flat(\check{r})$ to a match $m : L_c \rightarrow G$. The colimit property of L_c ensures that $m_{s \in S_c} : L_s \rightarrow G$ is a variability-based match family. Thus, match m paired with configuration c is a variability-based match $\check{m}$, being applied with rule r_c.

From *DirectTrans*($\check{r}$, G) , we consider rule application $G \Rightarrow_{r_c,\check{m}} H$ of a configuration-induced rule r_c to a variability-based match $\check{m}$. r_c is an element of $Flat(\check{r})$. $\check{m}$ provides a match $m_c : L_c \rightarrow G$, being applied with rule r_c.

□

4.4.2 Variability-Based Matching Algorithm

In this section, we describe an algorithm for implementing the concept of variability-based match (Def. 28). Our guiding intuition is to find matches for the base rule first and then expand these matches for the variable parts.

The first step, matching the base rule (see Def. 25), yields matches for the common parts that we store in a collection called *baseMatches*. Function FINDMATCHES, shown in Alg. 3, extends *baseMatches* to find matches for the variable parts. It enumerates all valid variability configurations, derives the corresponding rules and matches them classically. FINDMATCHES receives an input model, a variability-based rule, the *baseMatches* set, and two intermediate parameters: a data structure *bindings* that assigns each of the variability expressions used in the rule (i.e., the variability model and all used presence conditions) to one of the literals *true*, *false* or *unbound*, and a set to accumulate variability-based matches. The binding for the variability model is set to *true*, while all presence conditions are set to *unbound*. The accumulative set is initially empty. The function outputs this set of variability-based matches.

An execution of FINDMATCHES systematically binds all presence conditions, starting on Line 2 with an arbitrary one that we call pc_0. To enumerate all valid configurations, we first set pc_0 to *true* and then to *false* (Lines 3-4 and 5-6). In both calls to FINDMATCHESINNER, we first consider those presence conditions that were previously unbound and now are either contradicting or implied by the current bindings. On Lines 10 and 11, we compute them using a SAT solver, calling the results $bindings_{\not\rightarrow}$ and $bindings_{\rightarrow}$ (for *false* elements and *true* elements, respectively). We update the bindings accordingly on Line 12. If all presence conditions are now bound, the problem becomes classic matching. We determine the classic rule to be matched by removing rule elements with a *false* presence condition on Line 14. The classic match-finder tries to bind the rule elements contained in the derived rule, but not in the base rule. The computed matches are translated into variability-based matches, being pairs of a classic match and the current variability configuration, on Lines 15-16. If some presence conditions have not been bound, we call FINDMATCHES again on Line 18. On Lines 7 and 19, we reset temporary bindings of variables to clean up before backtracking.

To exemplify our algorithm, we continue with the scenario at the end of Sec. 4.2.1. First, we create and match the base rule, comprising the

Algorithm 3 Pseudocode for recursive function FINDMATCHES.

Input: *model*: Input model
Input: *rule*: Variability-annotated rule
Input: *baseMatches*: Classic matches of the base rule
Input: *bindings*: {Variability expressions used in *rule*} $\rightarrow$ {*true*, *false*, *unbound*}
Input: *matches*: Accumulated variability-based matches
Output: *matches*: Accumulated variability-based matches

```
1  function FINDMATCHES(model, rule, baseMatches, bindings, matches)
2      pc_0 = bindings.select(unbound).get(0)
3      bindings.set(pc_0, true)
4      FINDMATCHESINNER(model, rule, baseMatches, bindings, matches)
5      bindings.set(pc_0, false)
6      FINDMATCHESINNER(model, rule, baseMatches, bindings, matches)
7      bindings.set(pc_0, unbound)
8      return matches
9  function FINDMATCHESINNER(model, rule, baseMatches, bindings, matches)
10     bindings_↯ = bindings.select(unbound).select(p | bindings.contradicts(p))
11     bindings_→ = bindings.select(unbound).select(p | bindings.implies(p))
12     bindings.setAll(bindings_↯ → false, bindings_→ → true)
13     if bindings.select(unbound).isEmpty() then
14         classicRule = rule.removeAll(x | x.pc ∈ bindings.select(false))
15         classicMatches = Matcher.find(model, classicRule, baseMatches)
16         matches.addAll(createVariabilityBasedMatches(classicMatches))
17     else
18         FINDMATCHES(model, rule, baseMatches, bindings, matches)
19     bindings.setAll(bindings_↯ → unbound, bindings_→ → unbound)
20     return
```

elements annotated with *true*, by classic match-finding. The computed *baseMatches* set contains exactly match m_{base}. We arbitrarily select a presence condition $\neg$*qualified* and set it to *true* on Line 3, thus deriving *qualified* to be *false* on Lines 10-12. To bind the rest of the presence conditions, we call FINDMATCHES again on Line 18. We then select *forall* and set it to *true*, thus setting $\neg$*forall* to *false* and completing the binding of presence conditions. On Line 14, we remove all rule elements labelled $\neg$*forall* or $\neg$*qualified* to derive Rule A. Calling the classic match finder on this rule on Line 15 yields m_{A}. We pair this classic match with the current bindings to create a variability-based match. The remaining three configurations are determined analogously; however, they do not yield any additional matches.

Complexity of our algorithm is determined by the number of configurations which grows exponentially with the number of variation points. Of course, the configurations determine rules that in the classic approach would be matched individually. Thus, complexity of our algorithm is the same as that in classic matching. Yet, since we save matching effort by precomputing base matches and then extending them, we predict our algorithm to perform better than the classic one.

4.5 Framework

Given a rule set with similar rules, rule merging, outlined in Fig. 4.13, aims to find an efficient representation of these rules using a set of variability-based (VB) rules. In this section, we define a formal framework of three components called *clone detection*, *clustering* and *merge construction*: We specify the input and output of each component and show correctness of rule merging based on these specifications. Each component may be instantiated in various ways, as long as its specification is implemented.

Clone Detection

Clone detection allows identifying overlapping portions between the input rules. We use clone detection as a prerequisite for both clustering and merge construction: To cluster rules based on their similarity, we consider rules as similar if they share a large overlap. Merging overlapping portions rather than individual elements allows us to preserve the essential structural information expressed in the rules. Moreover, the performance of merged rules in terms of their execution time can be considerably improved by restricting clone detection to *connected* portions: Matching connected patterns is a problem that can be handled much more efficiently than that of multiple independent patterns [123].

Formally, given a set of rules, a *clone* is a largest subrule that can be embedded into a subset of this rule set. To support instantiations of clone detection that are restricted to connected portions, we analogously define *connected clones* based on largest connected subrules. To establish a well-defined merge construction, we define a *compatibility* relation and a *reduction* operation. Compatibility ensures that two clones never assign the same object contained in one rule to diverging objects contained in another. Reduction allows discarding irrelevant mappings.

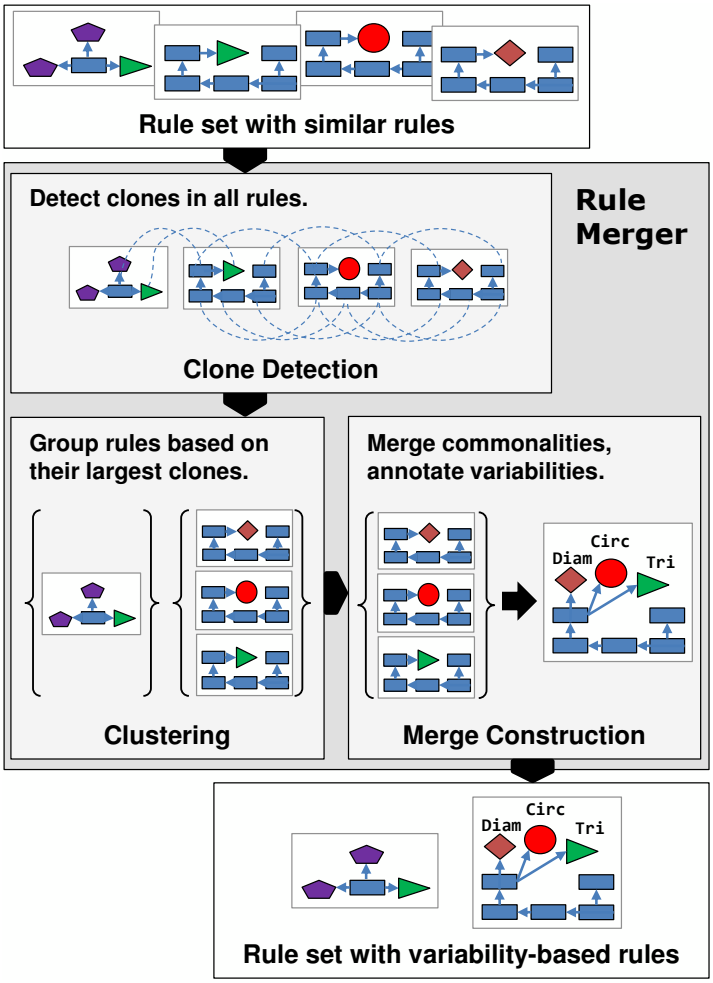

Figure 4.13: Refined overview of rule merging.

Definition 30 (Clone group). *Given a set* $\mathcal{R} = \{r_i | i \in I\}$ *of rules, a* (connected) clone group $CG_{\mathcal{R}} = (r_c, \mathcal{C})$ *over* $\mathcal{R}$ *consists of a (connected) rule* r_c*, called* clone*, and set* $\mathcal{C} = \{c_i | i \in I\}$ *of subrule mappings* $c_i : r_c \to r_i$ *iff there is no set* $\mathcal{C}' = \{c'_i | i \in I\}$ *of subrule mappings* $c'_i : r'_c \to r_i$ *with a subrule mapping* $i : r_c \to r'_c$ *where* r'_c *is a (connected) rule.*
$CG_{\mathcal{R}}$ *is* reduced *to* $\mathcal{R}' \subseteq \mathcal{R}$*, written* $Red(CG_{\mathcal{R}}, \mathcal{R}') = (r_c, \mathcal{C}')$*, by* $\mathcal{C}' = \mathcal{C} \setminus \{c_j | r_j \notin \mathcal{R}'\}$*. Two clone groups* $CG_{\mathcal{R}} = (r_c, \{c_i | i \in I\})$ *and* $CG_{\mathcal{R}'} = (r'_c, \{c_j | j \in J\})$ *with* $\mathcal{R}' \subseteq \mathcal{R}$ *and* $J \subset I$ *are* compatible *if there is a subrule mapping* $in : r_c \to r'_c$ *with* $c_j = c'_j \circ in$ *for all* $j \in J$*.*

Table 4.1 shows the result of applying clone detection to the classic rules in the example shown in Fig. 4.2. Each row denotes a clone group, comprising a set of rules and a clone (indicated by its size) present in each

Name	Rules	Size
CG1	{*E*, *F*}	10
CG2	{*D*, *E*, *F*}	8
CG3	{*C*, *E*, *F*}	7
CG4	{*B*, *C*}	6
CG5	{*A*, *B*, *C*, *D*, *E*, *F*}	5

Table 4.1: Clone groups, as reported by clone detection.

of these rules. The rows are ordered by the size of the clone, calculated as the number of involved objects and interrelations. CG2 in particular represents objects #15-18, #20-23 and their interrelations. CG1 incorporates objects #19 and #25 and their incoming relationships in addition. Clone groups CG1 and CG2 are compatible: The clone of CG2 extends the one of CG1. CG2 can be reduced to rule set $\{E, F\}$ by discarding the subrule embedding into rule D. CG2 and CG3 are not compatible: their rule sets are not in subset relation. All clone groups in this example are connected.

The output of clone detection is a set of clone groups – in the example, all rows of Table 4.1. These clone groups may be pair-wise incompatible.

Clustering

As a prerequisite for merge construction, we introduce *clustering*, an operation that splits a rule set into a cluster partition based on the similarity between rules. Its input are a set of rules and a set of clone groups over these rules.

Definition 31 (Cluster). *A* cluster Cl *over a set* $\mathcal{R}$ *of rules is a set of clone groups* $CG_{\mathcal{R}'}$ *over each subset* $\mathcal{R}' \subseteq \mathcal{R}$*. Given a partition* $\mathcal{P}$ *of* $\mathcal{R}$*, a* cluster partition *is a set* $Par(Cl)_{\mathcal{P}}$ *of clusters over* Cl *where for each* $P \in \mathcal{P}$ *there is a cluster* $Cl_P \in Par(Cl)_{\mathcal{P}}$ *comprising clone groups* $Red(CG_{\mathcal{R}'}, P)$ *and* $CG_{P'} \subseteq CG_P$ *over subsets* P' *of* P*. Each cluster* $Cl_P \in Par(Cl)_{\mathcal{P}}$ *is called* a sub-cluster *of* Cl*.*

In the example, there is a cluster partition over the rule set with sub-clusters over {*A*}, {*B*, *C*}, and {*D*, *E*, *F*}. We consider the sub-cluster over {*D*, *E*, *F*}: The clone groups over this set are obtained by reducing the mappings of {CG2, CG5} to rules *D*, *E* and *F*, i.e., discarding all mappings not referring to either rule. To obtain the clone groups over its

subset {*E*, *F*}, we include CG1 and CG3 as well and further reduce the mappings of {CG1, CG2, CG3, CG5} to *E* and *F*.

The output of clustering is a clustering partition over the original set of rules.

Merge Construction

Merge construction takes a cluster partition over the entire rule set as input. Each sub-cluster becomes a VB rule in the output. The available information on overlapping, given by clone groups, is considered to merge corresponding elements. Merging requires that the clone groups over each sub-cluster are compatible. Incompatible clone groups have to be discarded before merging, a non-trivial task requiring a strategy to determine what to discard. The instantiation in Sec. 4.6 provides such a strategy. To maintain traceability between original and new rules, we define a variation point for each original rule. The variability model is set over the variation points, specifying that exactly one of them is valid at a time.

Definition 32 (Cluster merge)**.** *Given a cluster partition $Par(Cl)_{\mathcal{P}}$ over a cluster Cl over $\mathcal{R}$, each sub-cluster $Cl_P \in Par(Cl)_{\mathcal{P}}$ is merged to a variability-based rule $\hat{r} = (r, S, v, vc)$ by merging all rules in $P = \{r_j | j \in J\}$ over compatible clone groups in Cl_P. The result is a rule r. $S = \{s_i : r_i \to r\}$ consists of all resulting subrule embeddings. Variation points V are determined by the rules in P: $V = \{v_j | j \in J\}$. Moreover, $v = \text{Xor}_{j \in J}(v_j)$ and $vc(s_j) = v_j$. We use the notation $Merge(Cl_P)$ to indicate $\hat{r}$ and $Merge(Cl) = \{Merge(Cl_P) | Cl_P \in Par(Cl)_{\mathcal{P}}\}$.*

Rules are merged over compatible clone groups by gluing those rule elements that are in relation via subrule mappings. This relation is extended to an equivalence relation, so in particular, the transitive closure is considered as well. All elements not in the relation are merged in disjointly.

In the example, considering all clone groups identified for the sub-cluster over {*D*, *E*, *F*}, CG1–2 are compatible; since we consider the reduction to {*D*, *E*, *F*} they are incompatible to CG3 and CG5. Merging the sub-cluster based on clone groups CG1–2 yields a VB rule isomorphic to *D*+*E*+*F* in Fig. 4.3. The variability model v is set to $xor(cfg(v_D), cfg(v_E), cfg(v_F))$. In the compact representation of VB rules shown in Fig. 4.3, the presence condition of an element is the disjunction of all variation points whose corresponding subrules contain the element.

As a proof of well-definedness, we show that merging a rule set and then flattening it produces the original set.

Theorem 2 (Correctness of rule merging)**.** *For any cluster Cl over a set $\mathcal{R}$ of flat rules, we have Flat(Merge(Cl)) = $\mathcal{R}$.*

Proof. Given a cluster Cl over $\mathcal{R}$, for any partition of Cl we have *Merge*$(Cl) = \{$*Merge*$(Cl_P)|Cl_P \in$ *Par*$(Cl)_\mathcal{P}\}$. We show that for any subcluster $Cl_P \in$ *Par*$(Cl)_\mathcal{P}$, we have *Flat*(*Merge*$(Cl_P)) = P$ and assume $P = \{r_j|j \in J\}$. *Merge*(Cl_P) yields $\hat{r} = (r, S, v, vc)$ as defined in Def. 32. Next, we consider *Flat*$(\hat{r})$. Since $v =$ *Xor*$_{j\in J}(v_j)$, all valid configurations bind exactly one v_j to true. We consider a fixed $j \in J$, yielding configuration c_j with $c_j(v_j) =$ *true* and $c_j(v_i) =$ *false* for all $v_i \in J \setminus \{v_j\}$. $s_j \in S_P$ is in S since c satisfies $vc(s_j) = v_j$. Since there is exactly one subrule embedding s_j for each c_j, no further merging of subrule embeddings is needed. The resulting subrules are the flat rules forming P. □

Note that the opposite operation, first flattening a VB rule set and then merging the resulting flat rules, may not yield the same VB rule set: In general, there are several VB rules with the same flattening. In fact, Thm. 2 ensures that *all* VB rule sets created by instantiations of rule merging have the same flattening, i.e., they are semantically equivalent.

4.6 Instantiation

In this section, we present our instantiation of the rule merging framework based on state-of-the-art clone detection and clustering tools and a novel merge construction algorithm. We describe two input parameters that enable customizations with respect to specific quality goals.

Clone Detection

We considered the applicability of three techniques for clone detection, each of them allowing to identify *connected clones* as per Def. 30. First, we applied *gSpan*, a general-purpose graph pattern mining tool [134]. Using this tool, we experienced heap overflows even on small rule sets. Second, we re-implemented eScan [82], which terminated with *insufficient memory* errors for larger rule sets. While our implementation could be flawed, [29] reports on a similar experience with their re-implementation of eScan. Finally, we applied ConQAT [29], a heuristic technique which delivers fast performance at the expense of precision.

It was able to analyze rule sets of 5000 elements in less than 10 seconds while reporting a large portion of relevant clones. We used ConQAT in our experiments on realistic rule sets.

We provide a customization to increase the speed-up produced by the constructed rules: The performance-critical task in rule application, *matching*, considers just the rule left-hand sides. Consequently, performance is optimized when rules are merged based on their overlap in left-hand sides. To this end, a Boolean parameter *restrictToLhs* allows restricting the rule portions considered by clone detection. When set to *true*, it only finds and reports clones for left-hand sides.

Clustering

From a large variety of approaches to cluster a set of objects based on their similarity [133], we chose *AverageLinkage*, an hierarchical agglomerative method, due to its convenient application to our approach. It assumes a distance function – a measure of similarity between the clustered elements. We consider the similarity of rule pairs, defining it as the size of the rules' largest common clone divided by their average size. In the example, similarity of rules *E* and *F* is calculated based on CG1, evaluating to $\frac{10}{11} = 0.91$. It further assumes a customizable *cutting-level threshold* parameter that we describe in what follows.

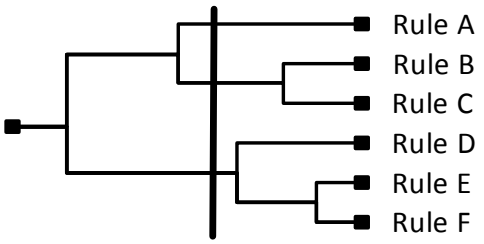

Figure 4.14: Cluster dendrogram, as reported by clustering.

The method builds a cluster hierarchy, often visualized using a *dendrogram* – a tree diagram arranging the input elements, as shown in Fig. 4.14. Tree nodes describe proximity between rule sets. The "lower" in the tree two nodes are connected, the more similar are their corresponding rules. For example, rule *D* is similar to *E* and *F*, but the similarity is not as strong as that between just *E* and *F*. The clustering result is obtained by "cutting" using the cutting-level threshold, marked by a vertical bar in Fig. 4.14, and collecting the obtained subtrees.

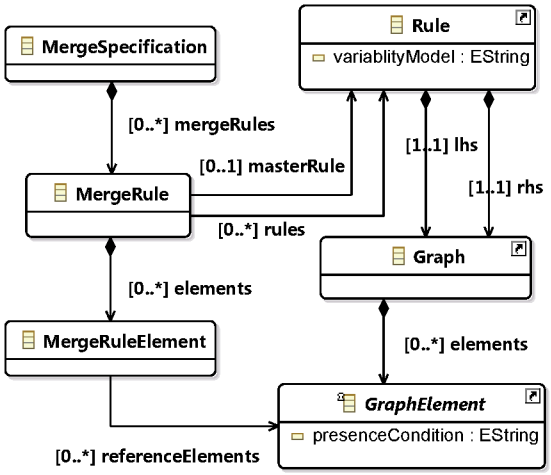

Figure 4.15: Merge specification metamodel.

Merge Construction

We propose a custom algorithm for merge construction. It proceeds in two steps: determining *what* is to be merged and *how* to do the merging. The first step, called *merge computation*, takes as input the cluster partition created by clustering (Def. 31). To ensure a well-defined merge, merge computation refines the given cluster partition by discarding incompatible clone groups (Def. 30), retaining sub-clusters for which a set of compatible clone groups is available. To this end, we apply a greedy strategy that aims to capture a high degree of overlap. Each sub-cluster becomes a `MergeRule` in the output of merge computation, a `MergeSpecification`. The second step, *merge refactoring*, creates VB rules according to this `MergeSpecification` as per Def. 32.

Fig. 4.15 specifies a metamodel for the interface between merge computation and merge refactoring. `MergeSpecification`, corresponding to the overall rule set, acts as an overarching container for a set of `MergeRules`. One `MergeRule` identifies a sub-cluster that is to be merged into a VB rule. In order to preserve the graphical layout of the contained rules, one rule is stated as *masterRule*; this rule is used as a starting point in creating the VB rule. To retain as much layout information as possible, it is best to select the largest input rule as the *masterRule*. A `MergeRule` specifies all elements to be unified in the created VB rule. For each element in the resulting rule, a `MergeRuleElement` is defined, referring to the elements to be represented by it. In a consistent specification, each rule element is referred to by exactly one `MergeRuleElement`.

Algorithm 4 Pseudocode for merge computation.

```
 1 function COMPUTEMERGE(cl : Cluster[])
 2     var mergeSpecification = ∅
 3     for each c ← cl
 4         var cg = c.cloneGroups
 5         while cg ≠ ∅                                  ▷ Create a new sub-cluster
 6             var top = FINDTOPCLONEGROUP(cg)
 7             var mergeRule = CREATEMERGERULE(top)
 8             var considered = {top}
 9             while HASCOMPATIBLE(considered, cg)
10                 var comp = FINDTOPCOMPATIBLE(cg)
11                 var temp = CREATEMERGERULE(comp)
12                 INTEGRATE(mergeRule, temp)
13                 considered.ADD(comp)
14             mergeSpecification.rules.ADD(mergeRule)
15             cg.REMOVEMAPPINGS(mergeRule.rules)
16             cg.REMOVEALLEMPTY
17             cg.REMOVEALL(considered)                  ▷ Done with current sub-cluster
18     return mergeSpecification
```

Algorithm 4 sketches merge computation. The output `MergeSpecification` is created in line 2 and incrementally filled by considering each cluster. In each iteration of the loop starting in line 5, a new sub-cluster is constructed. We apply a greedy strategy to integrate as many compatible clone groups as possible, starting with the *top* – the largest available – clone group in lines 6-8 and incrementally adding the next largest compatible ones in 9-13. For each clone group, we temporarily create a new `MergeRule`, integrating its contents with the result `MergeRule` in line 12. When no more compatible clone groups are found, we add the `MergeRule` to the result and discard mappings that concern its rules from the remaining clone groups, from which we remove all empty and already considered clone groups, in lines 14-17. We repeat this process until no clone groups are left to consider.

In the example, considering cluster {*D*, *E*, *F*} containing clone groups CG1, CG2, CG3, and CG5, the largest one CG1 is chosen as top group in line 6. In line 7, a `MergeRule` is created based on CG1, specifying the merge of the involved rules *E* and *F*. One `MergeRuleElement` is created for each pair of clone elements and for each non-clone element, e.g., one for {#15, #20} and one for {#24}. In lines 9-13, CG2 is identified as the next largest compatible clone. Its temporary merge rule, specifying the merge of rules *D*, *E* and *F*, is created. The two merge rules are integrated by establishing that each rule element finally belongs to exactly one `MergeRuleElement`, which involves the deletion of redundant `MergeRuleElement`s. Then, as no compatible clone groups

can be found, the `MergeRule` comprising the information of CG1 and CG2 is added to the resulting `MergeSpecification`. In lines 15–16, the mappings of CG3 and CG5 for *D*, *E* and *F* are removed, leaving them empty and leading to their discarding.

Based on a given `MergeSpecification`, the merge refactoring procedure is a straightforward implementation of Def. 32. The merge refactoring procedure is shown in Algorithm 5. The following steps are performed: in line 3, each graph element not common to all rules gets a presence condition. We merge all non-common rule elements into the specified master rule in line 4. In lines 5 and 6, the variability model of the master rule is set and the non-master rules are removed from the overall rule set.

Algorithm 5 Merge refactoring.

```
1 procedure MERGEREF(ms:MergeSpecification)
2     for each merge rule mr in ms
3         SETPRESENCECONDITIONS(elements)
4         MERGELHSRHSGRAPHS(masterRule,rules)
5         SETVARIABILITYMODEL(masterRule)
6         REMOVENONMASTERRULES(rules)
```

In the example, the presence conditions shown in Fig. 4.3 are created in line 3. As specified by `MergeRuleElements`, most rule elements in *D*, *E* and *F* have corresponding elements in all other rules, rendering their presence condition to be *true*. Exceptions are objects #37, #38, and their connecting edges that now receive a non-*true* presence condition. Since Rule *F* already contains all target elements required for implementing the `MergeSpecification`, line 4 has no effect. Otherwise, elements from other rules might have been added in the specified places in the rule. Lines 5 and 6 set the variability model to mutual exclusion between variation points *D*, *E* and *F* and remove rules *D* and *E*.

4.7 Implementation

We implemented variability-based rules and rule merging on top of Henshin [6], a model transformation language for the Eclipse Modeling Framework [106]. The architecture of our rule merging implementation follows the abstract representation given in Fig. 4.13. In what follows, we discuss its components.

Clone detection: We used publicly available implementations of *gSpan*[1] and ConQat[2]. We re-implemented *eScan* since no existing implementation was available. To apply these techniques, we first normalized Henshin rules to become labeled graphs so that we could translate them to the custom graph representation expected by each technique. The label for each element is comprised of its action and the name of its type. To capture variability at the granularity of attributes (for instance, see the variability-annotated *name* attribute in element #29 of Fig. 4.3), our translation accounted for attributes: We represented each attribute as a pair of a custom node and edge. Since the clone detection techniques consider the set of input rules as one large graph, the reported clone groups may include multiple clones within the same rule. For a well-defined merge, we arbitrarily select one instance and discard others. We consider additional use-cases for our implementation of clone detection elsewhere [113].

Clustering: We used a publicly available implementation of *AverageLinkage*[3]. A concern about the lack of scalability of this approach concerning large data-sets [133] did not manifest itself. We used clustering solely for the grouping of rules and not of the contained objects.

Merge construction: We implemented the algorithm sketched in Sec. 4.6. The created VB rules were amenable to the publicly available implementation of VB rule application[4] that we used in our experiments.

4.8 Evaluation

In this section, we evaluate rule merging by comparing the created rules to the corresponding classic rules and to rules that were merged manually. We focus on two research questions:

- **RQ1**: How well does rule merging achieve its goal of creating high-quality rule sets?
- **RQ2**: What is the impact of design decisions made by rule merging on the quality of the created rules?

To answer these questions, we applied our instantiation of rule merging on rule sets from two real-life model transformation scenarios, called

[1] https://www.cs.ucsb.edu/ xyan/software/gSpan.htm
[2] https://www.cqse.eu/en/products/conqat/install/
[3] https://github.com/lbehnke/hierarchical-clustering-java
[4] https://github.com/dstrueber/varhenshin

OCL2NGC and FMRECOG, and one adapted from literature, called COMB. The main quality goal in these scenarios is performance: In our communication with the developer of the OCL2NGD [7], the developer pointed out that the bad performance of the rule set was an obstacle to its usefulness. FMRECOG is an automatically derived rule set used in the context of model differencing [18], a task that necessitates low latency. COMB was introduced as a benchmark in [127]. Thus, we optimized the two input parameters described in Sec. 4.6 for performance.

We assess the quality of the produced rules with respect to performance and reduction in redundancy. To quantify *performance*, we applied the rule sets ten times on all input models and measured cumulative execution time on all input models. To quantify *redundancy reduction*, we measured the relative decrease in the number of rule elements, based on the rationale that we produce semantically equivalent, yet syntactically compacted rules (Thm. 2). As discussed in Sec. 4.2, reducing redundancy in rules is related to benefits for their maintainability.

In what follows, we first describe each of these three scenarios in detail. Afterwards, we explain our research methodology. Finally, we present and discuss our results and address potential threats to validity.

Scenarios

The first scenario, OCL2NGC, is an Object Constraint Language (OCL) to Nested Graph Constraints translator [7]. In the rule set, comprising 54 rules in total, we focused on a subset of 36 rules that are applied non-deterministically as long as one of them is applicable. We call it a *bottleneck rule subset* (BRS) as it causes a significant performance bottleneck during translation. For our experiments, we have refactored BRS automatically, using the automated approach, and manually, allowing to compare both approaches to merging. For the manual merging, we clustered the input rules relying on naming similarities between the rules and merged them based on symmetries that we recognized in their diagrammatic representations, a daunting and time-consuming task spanning over three days. To measure performance, we applied all rule sets on ten OCL invariants from [7] designed for high coverage of the translation rules. The input model in each run included the actual invariant paired with the OCL standard library, yielding 1850 graph elements on average.

In the second scenario, we considered a rule set taken from a product-line evolution scenario [18]. The rule set, FMRECOG, contains 53 rules and specifies recognition rules for detecting applications of certain edit operations on a feature model. Its rules are applied on pairs of revisions of the same feature model. In order to detect edit operations after they were applied – a crucial activity in revision management – we need to find all matches for all rules – a highly performance-intensive task. To measure performance, we applied the rules in FMRECOG on nine feature models with 100 to 300 features each. The feature models were automatically generated using BeTTY [99] with parameterization profiles rendered after real-world feature models. For details, please see [18]. To create revisions, editing operations were applied randomly. Moreover, we preprocessed the rules in FMRECOG to remove instances of two advanced transformation features – rule amalgamation and negative application conditions – that are outside the scope of this work.

The third scenario is based on Varró et al.'s widely-known graph transformation benchmark *Comb Pattern* [127]. In the original benchmark, the task was to find occurrences of a small pattern – the *comb pattern* – in a large grid. The benchmark has two parameters: the size of the grid and the size of the comb. We extended the task to contain variability so the new task was to find combs of variable size *k*, where *k* can represent any integer in the range $[m_1, m_2]$. For our measurements, we considered the range $[3, 8]$, which was small enough to create the included rules manually, but large enough to expect an observable difference. We created the 6 comb pattern rules required in the classic approach. We measured performance on 10 different grids, spanning from 20x20 to 200x200 elements, which allowed us to consider a variety of input models of different sizes. We considered both sub-tasks described in the original paper: COMBNOMATCH and COMBALLMATCHES. In the former, the grid is constructed to contain no occurrences of the comb pattern. In the latter, the grid is constructed to contain many such occurrences.

Methods and Set-Up

To address RQ1, we investigated three subquestions: **RQ1.1:** *How do VB rules created by rule merging compare to the equivalent classical rules?* **RQ1.2:** *How do VB rules created by rule merging compare to those created manually?* **RQ1.3:** *How do the VB rules created by rule merging scale to large input models?* For RQ1.1, we considered all three rule sets. For RQ1.2, we considered the scenario where a manually created rule set

		Size		Execution time (sec.)			
Scenario	*Rule Set*	#Rules	#Elements	Total	Sd	Median	Sd
OCL2NGC	Classic	36	3045	916.6	96.3	46.0	7.1
	Manual Merge	10	1018	181.8	27.1	10.8	2.4
	Automatic Merge	12	2147	5.8	0.4	0.4	0.1
FMRECOG	Classic	53	4626	799.9	41.4	63.2	3.5
	Automatic Merge	12	2790	211.4	46.0	15.9	0.3
COMB	Classic	6	252	1.39	0.09	0.12	0.01
NOMATCH	Automatic Merge	1	62	0.24	0.09	0.02	0.01
COMB	Classic	6	252	10.4	0.18	0.83	0.02
SEVERALMATCHES	Automatic Merge	1	62	14.2	0.26	1.07	0.05

Table 4.2: Results for RQ1.1 and RQ1.2: Quality characteristics of the rule sets.

was available: OCL2NGC [116]. For RQ1.3, we considered the COMB scenario, as it features a procedure to increase the input model automatically (increasing the size of the input grid [127]); we measured the impact of model size on execution time until we ran out of memory.

To address RQ2, we investigated two questions: **RQ2.1** *What is the impact of clone detection?* **RQ2.2** *What is the impact of clustering?* For RQ2.1, we randomly discarded 25%–100% of the reported clone groups. For RQ2.2, we replaced the default clustering strategy by one that assigns rules to clusters randomly. We measured the execution time of the rules created using the modified input.

As clone detection techniques, we applied ConQat [29] on OCL2NGC and FMRECOG, as it was the only tool scaling to these scenarios. We applied gSpan [134] on the COMB rule set as it allowed us to consider all clones instead of an approximation. The input parameters were optimized independently for each scenario by applying the technique repeatedly until the execution time was minimized. Moreover, the Henshin transformation engine features an optimization concerning the order of nodes considered during matching. To avoid biasing the performance of the FMRECOG rule set by that optimization, we deactivated it. We ran all experiments on a Windows 7 workstation (3.40 GHz processor; 8 GB of RAM).

Results and Discussion

RQ1: How well does rule merging achieve its goal of creating high-quality rule sets?

Table 4.2 shows the size and performance characteristics for all involved rule sets. Execution time is provided in terms of the total and median amount of time required to apply the whole rule set on each test model, each of them paired with the standard deviation *(SD)*. The number of elements refers to edges and nodes, including both left-hand and right-hand side of the involved rules.

RQ1.1 The execution time observed for OCL2NGC after the rule merging treatment showed a decrease by the factor of 158. This substantial speed-up can be partly explained by the merging component of rule merging that eliminates the anti-pattern *Left-hand side not connected (LhsNC)* [123]: In the automatically constructed VB rules, connected rules are used as base rules, while in the classic rules, we found multiple instances of *LhsNC*. In the FMRECOG and COMB rule sets, the speed-up was less drastic, amounting to the factors of 4.5 and 5.8, respectively. When applying the COMB rule set on the SEVERALMATCHES scenario, which involves an artificial input model with many possible matches [127], execution time increased by the factor 1.36, showing a limitation of VB rules: If the number of base matches is very high, the initialization overhead for extending the base matches outweighs the initial savings. This overhead may be reduced by extending the transformation engine implementation. The amount of redundancy was reduced by 29% in OCL2NGC, 40% in FMRECOG, and 75% in COMB.

RQ1.2 In OCL2NGC, we found a speed-up by the factor of 36. To study this observation further, we inspected the manually created rules, again finding several instances of the *LhsNC* antipattern. This observation gives rise to an interesting interpretation of the manual merging process: While the designer's *explicit* goal was to optimize the rule set for performance, they implicitly performed the more intuitive task of optimizing for compactness. Indeed, the amount of reduced redundancy in the manually created rules (67%) was significantly greater than in those created by rule merging (29%), highlighting an inherent trade-off between performance- and compactness-oriented merging: Not including overlap elements into the base rule leads to duplications in the variable portions.

RQ1.3 As shown in Fig. 4.16, the last supported input model was a 480x480 grid for both rule sets. We observed that the ratio between the

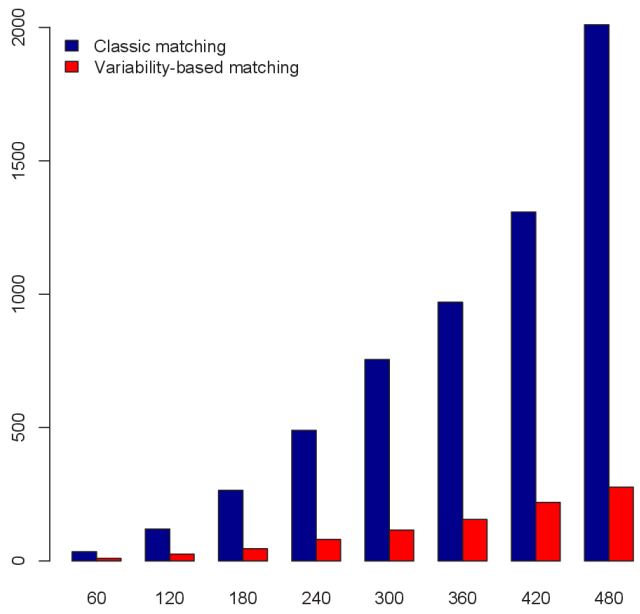

Figure 4.16: Results for RQ1.3: Execution time in sec. (y) related to length of grid (x).

execution time of applying the classic (left-hand bars) and the VB rules (right-hand bars) stayed the same in each iteration, independent of the size of the input grid: The VB rules were always faster by the factor of 6. In terms of the total execution time, the speed-up provided by the VB rules became more important as the size of input models increased.

RQ2: What is the impact of design decisions made by rule merging on the quality of the created rules?

RQ2.1 As presented in Table 4.3, the execution time for the FMRECOG rule set increased monotonically when we increased the amount of discarded overlap, denoted as d. OCL2NGC behaved almost monotonically as well. The slightly decreased execution time reported for d=0.25 can be explained by the heuristic merge construction strategy. While the merge of rules based on their largest clones might be adequate in general, in some cases it may be preferable to discard a large clone in favor of a more homogeneous distribution of rules. The reported execution time for d=0.75 was higher than that for the set of classic rules. In this particular case, small clones were used during merging, leading to small base rules, which resulted in many detectable matches and

thus in a high initialization overhead for extending these matches. To mitigate this issue, one could define a lower threshold for clone size.

	Discarded portion (d)				
Scenario	0.0	0.25	0.5	0.75	1.0
OCL2NGC	5.8	5.6	251	981	917
FMRECOG	211	252	604	690	800

Table 4.3: Results for RQ2.1: Impact of considered overlap on execution time (sec.).

RQ2.2 As indicated in Table 4.4, the employed clustering strategy had a significant impact on performance, amounting to factors of 13.7 for the OCL2NGC and 3.7 for the FMREGOC rule set. Interestingly, in OCL2NGC, random clustering still yielded better execution times than manual clustering did (see Table 4.2) – this is related to the fact that rule merging removed the *LhsNC* antipattern. In FMRECOG, randomly clustered rules were comparable to the classic ones.

Threats to validity and limitations

Factors affecting *external validity* include our choice of rule sets, test models and matching strategy, and the capability to optimize the two input parameters. While the considered rule sets represent three heterogeneous use cases, more examples are required to confirm that our approach works sufficiently well in diverse, potentially larger scenarios. To ensure that our test models were realistic, we employed the models used by their developers or described in the original benchmark. The performance of rule application depends on the chosen matching strategy, in our case, mapping this task to a constraint satis-

	Clustering strategy	
Scenario	AvLinkage	Random
OCL2NGC	5.8	80
FMRECOG	211	788

Table 4.4: Results for RQ2.2: Impact of clustering strategy on execution time (sec.).

faction problem [95]. We aim to consider the effect of alternative strategies in the future. Parameter tuning requires the existence of realistic test input models – still, given a rule set designed for productive use, it is reasonable to assume that such models exist.

With regard to *construct validity*, we focus on one aspect of maintainability, the amount of reduced redundancy. Giving a definitive answer on how to unify rules for optimal maintainability is outside the scope of this work. Specifically, our approach increases the size of individual rules, a potential impediment to readability [110]. We believe that this limitation can be mitigated by tool support. In the future, we aim to investigate providing *editable views* to developers, representing portions of a VB rule that correspond to configurations as selected by the user. Kästner proposes a related approach to address the readability issues associated with *preprocessors*, an annotative variability management mechanism [54].

4.9 Related Work

Merge-refactoring in product line engineering. Rule merging is related to a number of approaches in software product line engineering, specifically approaches that create feature-annotated representations from individual products. Nejati et al. [75] introduced the matching of Statechart models based on commonalities in their structure and behavior and applied it to merge models of telecommunication features. Ryssel et al. [96] proposed an approach for re-organizing product variants given in Matlab into annotative representations while identifying variation points. Rubin et al. [92, 94] defined a formal merge framework and instantiated it to class models and state machines, defining a number of desired qualities of the resulting model and studying how these can be best obtained. Ziadi et al. [137] proposed a language-independent approach for the reverse-engineering of product lines. These approaches operate on the basis of an element-wise comparison using names and as well as structural and behavioral similarities. In model transformation rules, the essential information lies in structural patterns. To our knowledge, our approach is the first that utilizes structural clone detection to identify such patterns.

Optimization of model transformation rule application. Our work can be considered a performance optimization for the NP-complete problem of transformation rule matching [9]. Earlier approaches in this

area are mostly complementary to ours as they focus on the matching of single rules [126, 47, 62, 1]. Mészáros et al. [73] first explored the idea of considering overlapping portions in multiple rules. Their custom technique for detecting these sub-patterns, however, did not scale up to complete rule sets. Instead, they considered just two rules at a time, which enabled a moderate performance improvement of 11%. In our approach, applying clone detection and clustering techniques gives rise to an increased speed-up. The incremental graph pattern matching approach in [125] succeeds in mitigating the memory concern of Rete networks by considering shared sub-patterns. Yet, the authors report on deteriorated execution times: The index tables that map sub-patterns to partial matches grow so large that performance is impaired. Multi-query optimization has also been investigated for relational databases [100]. In the more related domain of graph databases, all optimization approaches we are aware of focus on single-query optimization [135].

Clone refactoring. Circumstances under which clones can and should be eliminated are the subject of an ongoing discussion [92]. Based on empirical observations, Kim et al. [59] identified three types of clones: *short-lived clones* vanishing over the course of few revisions, *"unfactorable" clones* related to language limitations, and *repeatedly changing clones* where a refactoring is recommended. We second the idea that an aggressive refactoring style directed at short-lived clones should be avoided. Instead, targeting clones of the two latter categories, we propose to apply our approach on stable revisions of the rule set. Specifically, clones of the second category that were previously "unfactorable" due to the lack of suitable reuse concepts may benefit from the introduction of VB rules. An approach complementary to clone refactoring is *clone management*, based on a tool that detects and updates clones automatically [76]. This approach has a low initial cost, but requires constant monitoring.

Refactoring of model transformation rules. Multiple refactoring techniques have been proposed to refactoring transformation rules towards best practices. Taentzer et al. present an approach to specify refactorings for graph transformation systems based on pre-defined patterns [122]. Syriani et al. devise a plan to create a design patterns catalog for model transformations [121]. Rentschler proposes a modularization technique tailored at textual model transformation languages such as ATL [87]. In [88], Rentschler et al. employ a clustering-based strategy to identify interface elements during the introduction of interfaces into legacy transformation rules. Cuadrado et al. present a reuse concept based on abstract transformation rules that can be instantiated for

variants of similar meta-models [24]. The abstract transformation rules are reverse engineered from existing transformation rules. All of these refactoring approaches are complementary to ours, since none of them considers clones.

Variability in model transformations. The variability-based rules introduced in this chapter are inspired by annotative representations of product lines [26, 55, 92] and augment annotative representations proposed in earlier works, e.g., [56, 103, 124]. While these earlier approaches allowed specifying a product line of transformation rules, they did not provide an automated refactoring technique to create such representations. Furthermore, they did not provide any benefits for their performance. Finally, the achieved level of expressiveness was either lower than that of the proposed approach or so high that an efficient handling is prohibited. As for performance, [103, 56] report on a trade-off between better variability management and a performance overhead, the latter caused by the derivation of rules. In contrast, variability-based rules and matching improve both the compactness *and* the performance of a transformation system. As for expressiveness, [103] and [124] are based on creating refinement rules for the variable parts and assigning them to one feature (or variation point). In turn, we support propositional presence conditions over variation points. In this respect, [56] goes even further by allowing users to annotate a rule element with embedded C++ code, which, however, would produce an extremely large search space for variability-based matching.

Rule refinement. Several model transformation languages implement *rule refinement* [63] – an important mechanism for reuse inside the same transformation system. In such languages, a base rule is refined by a set of sub-rules modifying it. Then, some approaches [4, 49] flatten the rules for application, i.e., compile them into simpler rules. The translational semantics in the approach proposed in RubyTL [25] is closest to ours – it applies the base rules first and then applies the refinement rules on the target model of the transformation. In contrast, our approach aims to efficiently find matches in the *source* model.

4.10 Conclusion

In this chapter, we proposed variability-based (VB) model transformation, a novel approach to improve maintainability and performance in model transformation systems. Moreover, we introduced rule merg-

ing, an approach for constructing VB rules automatically. Our experiments showed that the approach is effective: The created rules always had preferable quality characteristics when compared to classical rules, unless the number of expected matches was very high. Notably, the created rules were more effective when applied to transforming large input models. It is apparent that using the approach, the performance of model transformation systems as well as redundancy-related maintainability concerns can be considerably improved, making the benefits of VB rules available while imposing little manual effort.

Chapter 5

Conclusions and Outlook

In this chapter, we summarize the key conclusions of this thesis and outline a selection of directions for future research based on these conclusions.

5.1 Conclusions

Refactoring is a cornerstone of software quality assurance. It aims to improve the quality of a software system by reorganizing its structure without affecting its behavior. In this thesis, we considered a set of refactorings in the context of large-scale Model-Driven Engineering (MDE) that are typically performed by hand, thereby imposing a significant effort on developers. We provide a set of techniques that allow automatizing these tasks.

We provided *model splitting* as a refactoring technique that allows splitting a monolithic model into a set of sub-models. This technique is useful to support collaborative model-driven scenarios: By assigning each collaborator a set of sub-models under their responsibility, the likelihood of editing conflicts is considerably decreased. The technique can also be applied to split a large diagram into multiple diagram views, reducing the effort required to navigate and comprehend large diagrams.

In our evaluation of model splitting, we found that our technique can achieve an average accuracy of 80% when compared to manual split-

ting. We provide elaborated tool support that enables the user of the technique to post-process the results, so that false assignments become easy to reconcile by hand. Furthermore, we contributed a supporting process that allows discovering the desired sub-models in an incremental manner. Together, these contributions give rise to a considerable improvement in collaborative modeling scenarios.

We introduced *component encapsulation* as a refactoring technique to turn a set of related models into a set of model components by deriving export and import interfaces for these components. This technique enables the systematic development of models in a context where heterogeneous domain-specific modeling languages are used to develop a large-scale systems in a model-driven fashion.

We demonstrated the usefulness of component encapsulation by embedding it into the context of a systematic modeling process based on heterogeneous domain-specific modeling languages (DSMLs). In this modeling process, the derived interfaces are highly valuable: They allow to distinguish between *safe* editing steps that can be performed without coordination, and *critical* editing steps demanding a further course of action. Precisely defined basic operations on composite models, formalized using category theory, provide a solid basis for a principled combination of heterogeneous DSMLs.

We proposed *rule merging* as a refactoring technique to eliminate redundancy in transformation rule systems. This redundancy was often an undesired consequence of the lack of adequate reuse concepts, giving rise to maintainability and performance drawbacks in the concerned rule sets. To satisfy the need for an adequate reuse concept, we introduced *variability-based (VB) model transformation rules* allowing to capture commonalities and differences in multiple rule variants explicitly. Rule merging then employs state-of-the-art clone detection and clustering techniques to create VB rules automatically.

In our evaluation of rule merging, the speed-up we observed compared to the redundancy-intensive classic rules was considerable, ranging between a factor of 4.5 and 158. We also compared the rules created by our technique against hand-created VB rules in one scenario. In this scenario, we observed a speed-up by the factor of 36. From inspection of the rule set, we concluded that the technique works specifically well in sets of rules that exhibit an anti-pattern called *disconnected left-hand side*. We also considered the effect of rule merging on the contained amount of redundancy, which has considerable implications for maintainabil-

ity. In the scenario where a comparison against hand-created rules was available, we observed that the automatically created rules were not as compact as the hand-crafted rules: Hand-crafting the rules achieved a reduction of redundancy by 67%, while the automatic creation reduced the redundancy by 30%. The reduced amount of redundancy can be further increased by devising more refined clone detection techniques.

5.2 Outlook

Based on the outlined conclusions, it is apparent that this work represents a step forward towards the vision of *MDE in the large*. Still, a variety of research directions remains to be explored. Together with the open challenges as touched on in Sec. 1.1, addressing these directions will help to fully implement the guiding vision of this thesis.

The discussion of the future research directions is driven by three principles:

- **Extension**: The scope of the contributions of this thesis may be widened by taking into account a larger variety of modeling languages, languages concepts, and tool components.
- **Combination**: When applied to suitable scenarios, the combination of individual contributions may give rise to additional advantages, but also to new challenges.
- **Actualization**: The usefulness of the contributions may be further demonstrated and enhanced by applying complementary research methods, specifically, such methods that allow to gather and account for systematic empirical evidence on how the contributions are used by developers.

In Sec. 5.2.1–5.2.3, we discuss extensions for each of the three main contributions of this thesis. In Sec. 5.2.4, we discuss an opportunity for combining two of these contributions. In Sec. 5.2.5, we discuss actualization in terms of an empirical study in the context of rule merging.

5.2.1 Model Splitting for Various Languages

We exemplified and evaluated the model splitting technique introduced in Chapter 2 on class models. Class models are a particular

relevant kind of model: First, they are a default means to specify the structure of a system. Second, they are also used in the context of meta-modeling to specify various modeling languages. Yet, in the context of UML and even more so in the context of domain-specific languages, a great variety of modeling (sub-)languages exists. Many of these languages are used to create large models. For the reasons outlined in Sec. 1.1.1, the use of these large models in collaborative scenarios creates a need for adequate splitting techniques.

Devising a generic, language-independent strategy for model splitting is a challenging task: Each model language has its own idiosyncrasies that should be taken into account to allow for a splitting that closely reflects the user intention. As a first step in this direction, the technique proposed in this thesis allows assigning different weights to different relationship kinds contained in the language. In the case of class models, this sub-contribution alone allowed to increase the average accuracy from 62% to 80% in terms of F-measure (see Table 2.4).

Yet, the proposed technique has a set of assumptions that do not hold for arbitrary modeling languages.

First, the information retrieval (IR) component of the technique assumes that model elements are named. In some modeling languages, elements do not carry names by default: One example are model transformation languages (for an example, see Sec. 4.2.1). Instead, the IR component may be replaced by a manual seed identification step, as proposed in the approach by Streekmann [111].

Second, the graph analysis in the model crawling component assumes a notion of *relatedness* between neighboring elements in the input model. Different kinds of languages might require a different kind of graph analysis. For instance, behavioral modeling languages often exhibit a sequential semantics that might benefit from analysis techniques such as *program slicing*.

Third, the technique assumes a property called *splittability*: Constraints satisfied by the input model must not be broken in sub-models. In several languages, this property can be implemented by means of a trivial reconciliation step. For instance, the constraint *"every activity diagram must have a start and end activity"* may be established by inserting these activities in each sub-model. To support languages where trivial reconciliation is not available, the technique might be extended to find an *optimal splitting satisfying all constraints*.

Fourth, the technique assumes that the input model is expressed using a formal modeling language. Yet, as splitting may be required in the beginning of a software project to determine a suitable system decomposition, it may be the case that the models are not expressed in a conventional modeling language yet. In *flexible modeling*, models are expressed using general-purpose graphical drawing tools [139] or text-based visual notations [114]. To be useful for such settings, model splitting may be combined with an upfront type inference [138] step.

5.2.2 Distributed Code Generation for Composite Models

In the state of the art as outlined in Sec. 3.3, a composite model can be used for code generation by merging all model components to become one large model. This model can then be used as input for a classical centralized code generator. Unfortunately, this approach to code generation defeats the purpose of composite modeling to a certain extent: A key motivation for composite models are situations where transmitting a full model is prohibited, e.g., for business, legal, or security reasons.

Consequently, when a software system is developed using composite models, a desirable approach is *distributed code generation*. A distributed code generation framework comprises a set of specialized code generators. Each generator is responsible for the code generation for one of the components included in the composite model. The code generation for a model component can be triggered as soon as the component has settled for a stable revision, allowing a flexible component-based code generation.

Such a framework needs to deal with several challenges: Usually, the interrelations between model components are to be reflected in interrelations between the generated code artifacts. To ensure that the code generated for a component yields syntactically correct code, the framework needs to enforce that these interrelations are intact: One approach is to forbid the code generation in case that the component depends on other components for that no up-to-date code generation result exists. A second approach is to generate stubs as replacements for missing artifacts. Both approaches require determining which artifacts are created by other components, a highly nontrivial task: It requires a semantic inspection of the involved generator templates. In particular, such an inspection involves *parsing* these templates, a feature not implemented in state-of-the-art code generator frameworks [11].

5.2.3 Extensions of variability-based model transformation

In Chapter 4, we have provided evidence that variability-based (VB) model transformation is a promising approach to improve several maintainability and performance characteristics of redundancy-intensive rule sets. We predict two further milestones that will render VB transformation fully applicable to real-world transformation systems: First, developers need to be furnished with adequate tooling to deal with the complexity introduced by variability annotations. Second, VB model transformation needs to account for all main features of state-of-the-art transformation languages.

VB rules increase the amount of information per rule, potentially producing visual clutter that may impair developer performance. We believe this limitation can be mitigated by tool support: In our recent work [118], we investigate providing *editable views* to developers, representing parts of a VB rule that correspond to a specific configuration selected by the user. Changes performed in an editable view are propagated back to the VB rule. Kästner proposes a related approach to address the readability problem associated with *preprocessors*, an annotative mechanism to manage source code variability [54].

Furthermore, we plan to enhance the expressiveness of VB rules. The current formalization and implementation of VB rules is tailored to simple typed and attributes rules. Covering several other important transformation features, such as negative application conditions [43] and rule amalgamation [33], will make our approach applicable to other rule-based model transformation languages such as [10, 39, 2].

5.2.4 Combining Variability-Based Transformation and Composite Modeling

Two of the techniques considered in this work aim to introduce new paradigms in existing artifacts: Component encapsulation turns a set of related models into a *composite model*, a set of model components. Rule merging turns a rule system into a *variability-based (VB) rule system.* If the claim holds that both of these paradigms are beneficial to the maintainability and overall quality of the involved models and transformation systems, the question arises of how these paradigms can be combined to improve these qualities even more.

VB model transformation can be used to capture the variability involved in the editing rules for the collaborative development of a composite model. Consider the rule delete entity from data model in Fig. 3.8 (Sec. 3.3). While we originally introduced this rule as a single isolated one, it can be considered an instance of a *family* of rules in the context of synchronous and asynchronous editing. All of these rules have the same *local* effect: The entity of the specified name is removed from the data model. Yet, the *global* effect of the involved rules differs: In case that the class to be deleted was part of an export interface, it should be removed from that interface. In case it was part of an import relationship, it should be removed from the import interface. Otherwise, there should be no effect on interfaces. All of these variants can be specified using a *variability-based composite graph rule*: The network base rule involves just the body. There is one object base rule which specifies the deletion of the class from the body object graph. Export and import interfaces are modeled as variant-specific nodes in the network rule. The corresponding object rules are variant-specific as well.

5.2.5 Rule Merging in the Wild

The rule merging technique provided in Chapter 4 is based on a *greedy strategy*: To provide a very compact representation of the involved rule variants, it aims to reduce the amount of clones to the largest extent possible. Still, while we provided evidence that the technique achieves its goal of reducing the amount of clones and also provides a considerable performance improvement, we did not provide empirical evidence that the design decisions are comprehensible to the human developers who will have to *maintain* the created rule sets.

The question of how rules are merged *in the wild* – in real-world settings, by developers with specific intentions – is one that deserves further attention and can only be addressed by accounting for empirical evidence. An experimental set-up to investigate this research question might look as follows: Developers of a specific expertise level (e.g. model transformation novices or experts) are provided a set of equivalent VB rule sets that only differ in their compactness and amount of contained redundancy. To determine which VB rule set provides the most “natural“ representation, the subjects are asked to participate in comprehension tasks and questionnaires where they rate the comprehension effort for each VB rule set. A methodological framework for preparing and reporting such an experiment is described e.g. in [48].

The results can be used to create a refined revision of rule merging informed by empirical evidence.

Bibliography

[1] Vlad Acretoaie and Harald Störrle. Efficient Model Querying with VMQL. In *Proc. of Workshop on Combining Modelling with Search- and Example-Based Approaches*, pages 7–16. CEUR-WS.org, 2015.

[2] Vlad Acretoaie, Harald Störrle, and Daniel Strüber. Transparent Model Transformation: Turning Your Favourite Model Editor into a Transformation Tool. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 121–130. Springer, 2015.

[3] Nuno Amálio, Juan de Lara, and Esther Guerra. Fragmenta: A Theory of Fragmentation for MDE. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*. Springer, 2015. pending publication.

[4] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 340–355, 2014.

[5] Thorsten Arendt. *Quality Assurance of Software Models-A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project*. PhD thesis, Philipps-Universität Marburg, 2014.

[6] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformation. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[7] Thorsten Arendt, Annegret Habel, Hendrik Radke, and Gabriele Taentzer. From Core OCL Invariants to Nested Graph Con-

straints. In *Proc. of International Conference on Graph Transformation*, pages 97–112, 2014.

[8] Thorsten Arendt, Gabriele Taentzer, and Alexander Weber. Quality Assurance of Textual Models within Eclipse using OCL and Model Transformations. In *Proc. of OCL and Textual Modeling Workshop*, pages 1–12, 2013.

[9] Mikhail J. Atallah. *Algorithms and Theory of Computation Handbook*. CRC press, 2002.

[10] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST*, 1, 2007.

[11] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[12] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. EMF model refactoring based on graph transformation concepts. *Electronic Communications of the EASST*, 3, 2007.

[13] Arnaud Blouin, Benoit Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 62 – 76, Wellington, New Zealand, 2011.

[14] Dominique Blouin, Alain Plantec, Pierre Dissaux, Frank Singhoff, and Jean-Philippe Diguet. Synchronization of Models of Rich Languages with Triple Graph Grammars: an Experience Report. In *Proc. of International Conference on Theory and Practice of Model Transformations*, 2014.

[15] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, (2):211–221, 1986.

[16] Jan Bosch. Architecture in the age of compositionality. In *Software Architecture*, pages 1–4. Springer, 2010.

[17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[18] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. Reasoning about Product-Line

Evolution using Complex Differences on Feature Models. *Journal of Automated Software Engineering*, pages 1–47, 2015.

[19] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33(1-6):137 – 157, 2000.

[20] Marsha Chechik, Fabiano Dalpiaz, Csaba Debreceni, Jennifer Horkoff, István Ráth, Rick Salay, and Daniel Varró. Property-Based Methods for Collaborative Model Development. In *Proc. of Workshop on The Globalization of Modeling Languages*, 2015. pending publication.

[21] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[22] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version Control with Subversion*. O'Reilly Media, Inc., 2004.

[23] Jesús Sánchez Cuadrado and Juan de Lara. Streaming model transformations: Scenarios, challenges and initial solutions. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 1–16. Springer, 2013.

[24] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Reverse Engineering of Model Transformations for Reusability. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 186–201. Springer, 2014.

[25] Jesús Sánchez Cuadrado and Jesus Garcia Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. *IEEE Transactions on Software Engineering*, 35(6):825–840, 2009.

[26] Krzysztof Czarnecki and Michał Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of International Conference on Generative Programming and Component Engineering*, pages 422–437. ACM, 2005.

[27] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[28] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

[29] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. Model Clone Detection in Practice. In *Proc. of Workshop on Software Clones*, pages 57–64. ACM, 2010.

[30] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[31] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, and Ulrike Prange. Consistent integration of models based on views of visual languages. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 62–76. Springer, 2008.

[32] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. *Fundamenta Informaticae*, 74(1):31–61, 2006.

[33] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[34] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proc. of Object Oriented Programming Systems Languages and Applications Companion*, pages 307–309. ACM, 2010.

[35] Michalis Famelis, Levi Lucio, Gehan Selim, Rick Salay, Marsha Chechik, James R. Cordy, Juergen Dingel, Hans Vangheluwe, and Ramesh S. Migrating Automotive Product Lines: A Case Study. In *Proc. of International Conference on Theory and Practice of Model Transformations*. Springer, 2015.

[36] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[37] Antonio Garmendia, Esther Guerra, Dimitrios S Kolovos, and Juan de Lara. EMF Splitter: A Structured Approach to EMF Modularity. In *Proc. of Workshop on Extreme Modeling*, page 22. CEUR-WS.org, 2014.

[38] Antonio Garmendia, Antonio Jiménez-Pastor, and Juan de Lara. Scalable model exploration through abstraction and fragmenta-

tion strategies. In *Proc. of Workshop on the Scalability of Model-Driven Engineering*, page 21. CEUR-WS.org, 2015.

[39] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of International Conference on Graph Transformation*, pages 383–397. Springer, 2006.

[40] Michael Goedicke, Torsten Meyer, and Gabriele Taentzer. ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proc. of International Symposium on Requirements Engineering*, pages 92–99. IEEE Computer Society, 1999. ISBN 0-7695-0188-5.

[41] Joseph A. Goguen. A categorical manifesto. In *Mathematical Structures in Computer Science*, pages 49–67, 1991.

[42] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[43] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[44] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 321–335. Springer, 2006.

[45] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. of International Conference on Graph Transformation*, pages 161–176. Springer, 2002.

[46] Frank Hermann, Susann Gottmann, Nico Nachtigall, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, and Thomas Engel. On an Automated Translation of Satellite Procedures using Triple Graph Grammars. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 50–51. Springer, 2013.

[47] Ákos Horváth, Gergely Varró, and Dániel Varró. Generic Search Plans for Matching Advanced Graph Patterns. *Electronic Communications of the EASST*, 6, 2007.

[48] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pages 201–228. Springer, 2008.

[49] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: A QVT-like Transformation Language. In *Proc. on Symposium on Object-Oriented Programming Systems, Languages, and Applications, Companion*, pages 719–720. ACM, 2006.

[50] Stefan Jurack. *Composite Modeling based on Distributed Graph Transformation and the Eclipse Modeling Framework*. PhD thesis, Philipps-Universität Marburg, 2012.

[51] Stefan Jurack and Gabriele Taentzer. Towards Composite Model Transformations Using Distributed Graph Transformation Concepts. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 226–240. Springer, 2009.

[52] Stefan Jurack and Gabriele Taentzer. A Component Concept for Typed Graphs with Inheritance and Containment Structures. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proc. of International Conference on Graph Transformation*. Springer, 2010.

[53] Stefan Jurack and Gabriele Taentzer. Transformation of Typed Composite Graphs with Inheritance and Containment Structures. *Fundamenta Informaticae*, 118(1-2):97–134, 2012.

[54] Christian Kästner. *Virtual separation of concerns*. PhD thesis, University of Magdeburg, 2010.

[55] Christian Kästner and Sven Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40, 2008.

[56] Amogh Kavimandan, Aniruddha Gokhale, Gabor Karsai, and Jeff Gray. Managing the Quality of Software Product Line Architectures through Reusable Model Transformations. In *Proc. of QoSA/ISARCS*, pages 13–22. ACM, 2011.

[57] Timo Kehrer, Christopher Pietsch, Udo Kelter, Daniel Strüber, and Steffen Vaupel. An Adaptable Tool Environment for High-level Differencing of Textual Models. In *Proc. of OCL and Textual Modeling Workshop*, pages 62–72. CEUR-WS.org, 2015.

[58] Pierre Kelsen, Qin Ma, and Christian Glodt. Models Within Models: Taming Model Complexity Using the Sub-Model Lattice. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 171–185. Springer, 2011.

[59] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[60] Dimitrios Kolovos, Louis Rose, Nicholas Matragkas, Richard Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A Research Roadmap towards Achieving Scalability in Model Driven Engineering. In *Proc. of Workshop on Scalability in Model Driven Engineering*, pages 1:1–10. ACM, 2013.

[61] Andreas Kraus, Alexander Knapp, and Nora Koch. Model-Driven Generation of Web Applications in UWE. In *Proc. of Workshop on Model-Driven Web Engineering*. Springer, 2007.

[62] Christian Krause, Matthias Tichy, and Holger Giese. Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 325–339. Springer, 2014.

[63] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reuse in Model-to-Model Transformation Languages: Are We There Yet? *Journal of Software and Systems Modeling*, pages 1–36, 2013.

[64] Thomas K. Landauer, Peter W. Foltz, and Darrell Laham. An Introduction to Latent Semantic Analysis. *Discourse Processes*, (25):259–284, 1998.

[65] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 166–182. Springer, 2014.

[66] John W Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[67] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.

[68] Qin Ma, Pierre Kelsen, and Christian Glodt. A Generic Model Decomposition Technique and Its Application to the Eclipse Modeling Framework. *Journal of Software and Systems Modeling*, pages 1–32, 2013.

[69] Sonja Maier and Mark Minas. A Pattern-based Approach for Initial Diagram Layout. *Electronic Communications of the EASST*, 58, 2013.

[70] Mihaly Makkai and Marek Zawadowski. Duality for simple ω-categories and disks. *Theory and Applications of Categories*, 8(7):114–243, 2001.

[71] Martin Mann, Heinz Ekker, and Christoph Flamm. The graph grammar library-a generic framework for chemical graph rewrite systems. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 52–53. Springer, 2013.

[72] Merriam-Webster, n.d. Web. 19 Aug. 2015. "Monolithic.". `http://www.merriam-webster.com/dictionary/monolithic`.

[73] Tamás Mészáros, Gergely Mezei, Tihamér Levendovszky, and Márk Asztalos. Manual and automated performance optimization of model transformation systems. *International Journal on Software Tools for Technology Transfer*, 12(3-4):231–243, 2010.

[74] Alix Mougenot, Xavier Blanc, and Marie-Pierre Gervais. D-Praxis: A Peer-to-Peer Collaborative Model Editing Framework. In *Proc. of International Conference on Distributed Applications and Interoperable Systems*, pages 16–29, Lisbonne, Portugal, 2009.

[75] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and Merging of Variant Feature Specifications. *IEEE Transactions on Software Engineering*, 38(6):1355–1375, 2012.

[76] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.

[77] Paul Nguyen and Robert Chun. Model Driven Development with Interactive Use Cases and UML Models. In *Software Engineering Research and Practice*, pages 534–540, 2006.

[78] OMG. UML Resource Page of the Object Management Group. `http://www.uml.org/`.

[79] OMG. The Essential MOF (EMOF) Model. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01.pdf`, 2006. Sec. 12.

[80] David L. Parnas. Information distribution aspects of design methodology. Technical report, DTIC Document, 1971.

[81] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[82] Nam H Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. Complete and Accurate Clone Detection in Graph-Based Models. In *Proc. of International Conference on Software Engineering*, pages 276–286. IEEE, 2009.

[83] L.S. Pitsoulis and M.G.C. Resende. *Handbook of Applied Optimization*. Oxford Univ. Press, 2002.

[84] Yasaman Talaei Rad and Ramtin Jabbari. Use of Global Consistency Checking for Exploring and Refining Relationships between Distributed Models: A Case Study. Master's thesis, Blekinge Institute of Technology, School of Computing, January 2012.

[85] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge Univ. Press, 2011.

[86] István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 342–356. Springer, 2009.

[87] Andreas Rentschler. *Model Transformation Languages with Modular Information Hiding*. PhD thesis, Karlsruher Institut für Technologie, 2015.

[88] Andreas Rentschler, Dominik Werle, Qais Noorshams, Lucia Happe, and Ralf Reussner. Remodularizing Legacy Model Transformations with Automatic Clustering Techniques. In *Proc. of Workshop on the Analysis of Model Transformations*, pages 4–13. CEUR-WS.org, 2014.

[89] Elie Richa, Etienne Borde, and Laurent Pautet. Translating ATL Model Transformations to Algebraic Graph Transformations. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 183–198. Springer, 2015.

[90] Martin P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *Proc. of European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 11–20, 2005.

[91] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: a Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[92] Julia Rubin and Marsha Chechik. Combining Related Products into Product Lines. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 285–300. Springer, 2012.

[93] Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In *Domain Engineering: Product Lines, Conceptual Models, and Languages*. Springer, 2013.

[94] Julia Rubin and Marsha Chechik. Quality of Merge-Refactorings for Product Lines. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 83–98. Springer, 2013.

[95] Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In *Proc. of Workshop on Theory and Application of Graph Transformations*, page 238. Springer Science & Business Media, 1998.

[96] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Automatic Variation-Point Identification in Function-Block-Based Models. In *Proc. of International Conference on Generative Programming and Component Engineering*, pages 23–32. ACM, 2010.

[97] Rick Salay, John Mylopoulos, and Steve M. Easterbrook. Managing models through macromodeling. In *Proc. of International Conference on Automated Software Engineering*, pages 447–450. IEEE, 2008.

[98] Douglas C Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):0025–31, 2006.

[99] Sergio Segura, José A Galindo, David Benavides, José A Parejo, and Antonio Ruiz-Cortés. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. of Workshop on Variability Modelling of Software-intensive Systems*, pages 63–71, 2012.

[100] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[101] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[102] Mary Shaw. The coming-of-age of software architecture research. In *Proc. of International Conference on Software Engineering*, pages 656–663. IEEE Computer Society, 2001.

[103] Marten Sijtema. Introducing Variability Rules in ATL for Managing Variability in MDE-based Product Lines. *Proc. of Workshop on Model Transformation with ATL*, pages 39–49, 2010.

[104] Richard Soley. Model Driven Architecture. *Object Management Group*, 2000.

[105] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2006.

[106] Dave Steinberg, Frank Budinsky, Marcelo Patenostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison Wesley, 2008.

[107] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[108] Harald Störrle. Large scale modeling efforts: a survey on challenges and best practices. In *Proc. of IASTED International Multi-Conference: Software Engineering*, pages 382–389. ACTA Press, 2007.

[109] Harald Störrle. Structuring Very Large Domain Models: Experiences from Industrial MDSD Projects. In *Proc. of European Conference on Software Architecture: Companion Volume*, pages 49–54. ACM, 2010.

[110] Harald Störrle. On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 518–534. Springer, 2014.

[111] Niels Streekmann. *Clustering-Based Support for Software Architecture Restructuring*. Springer, 2011.

[112] Daniel Strüber, Michael Lukaszczyk, and Gabriele Taentzer. Tool Support for Model Splitting using Information Retrieval and Model Crawling Techniques. In *Proc. of Workshop on Scalability in Model Driven Engineering*, pages 44–47. CEUR-WS.org, 2014.

[113] Daniel Strüber, Jennifer Plöger, and Vlad Acretoaie. Clone Detection for Graph-Based Model Transformation Languages. 2016. submitted.

[114] Daniel Strüber, Felix Rieger, and Gabriele Taentzer. MUTANT: Model-Driven Unit Testing using ASCII-art as Notational Text. In *Proc. of Workshop on Flexible Model Driven Engineering*, pages 2–11. CEUR-WS.org.

[115] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. RuleMerger: Automatic Construction of Variability-Based Rules. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, 2016. accepted.

[116] Daniel Strüber, Julia Rubin, Marsha Chechik, and Gabriele Taentzer. A Variability-Based Approach to Reusable and Efficient Model Transformations. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 283–298. Springer, 2015.

[117] Daniel Strüber, Julia Rubin, Gabriele Taentzer, and Marsha Chechik. Splitting Models Using Information Retrieval and Model Crawling Techniques. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 47–62. Springer, 2014.

[118] Daniel Strüber and Stefan Schulz. A Tool Environment for Managing Families of Model Transformation Rules. 2016. submitted.

[119] Daniel Strüber, Matthias Selter, and Gabriele Taentzer. Tool Support for Clustering Large Meta-Models. In *Proc. of Workshop on the Scalability of Model-Driven Engineering*, pages 7.1–4. ACM Digital Library, 2013.

[120] Daniel Strüber, Gabriele Taentzer, Stefan Jurack, and Tim Schäfer. Towards a Distributed Modeling Process Based on Composite Models. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 6–20. Springer, 2013.

[121] Eugene Syriani and Jeff Gray. Challenges for Addressing Quality Factors in Model Transformation. In *Proc. of International Conference on Software Testing, Verification and Validation*, pages 929–937. IEEE, 2012.

[122] Gabriele Taentzer, Thorsten Arendt, Claudia Ermel, and Reiko Heckel. Towards Refactoring of Rule-Based, In-Place Model Transformation Systems. In *Proc. of Workshop on the Analysis of Model Transformations*, pages 41–46. ACM, 2012.

[123] Matthias Tichy, Christian Krause, and Grischa Liebel. Detecting Performance Bad Smells for Henshin Model Transformations. In *Proc. of Workshop on the Analysis of Model Transformations*, pages 82–86. CEUR-WS.org, 2013.

[124] Salvador Trujillo, Ander Zubizarreta, Josune De Sosa, and Xabier Mendialdua. On the Refinement of Model-to-Text Transformations. In *Proc. of Jornadas de Ingeniería del Software y Bases de Datos*, pages 123–133, 2009.

[125] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In *Proc. of International Conference on Theory and Practice of Model Transformations*, pages 125–140. Springer, 2013.

[126] Gergely Varró, Katalin Friedl, and Dániel Varró. Adaptive Graph Pattern Matching for Model Transformations using Model-Sensitive Search Plans. *Electronic Notes in Theoretical Computer Science*, 152:191–205, 2006.

[127] Gergely Varró, Andy Schürr, and Daniel Varró. Benchmarking for Graph Transformation. In *Proc. of International Symposium on Visual Languages and Human-Centric Computing*, pages 79–88. IEEE, 2005.

[128] Steffen Vaupel, Daniel Strüber, Felix Rieger, and Gabriele Taentzer. Agile bottom-up development of domain-specific IDEs for model-driven development. In *Proc. of Workshop on Flexible Model Driven Engineering*, pages 12–21. CEUR-WS.org.

[129] Konrad Voigt. *Structural Graph-based Metamodel Matching*. PhD thesis, Univ. of Dresden, 2011.

[130] Gerd Wagner, Adrian Giurca, and Sergey Lukichev. A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML and SWRL. *Proc. of Workshop on Reasoning on the Web*, 2006.

[131] Ingo Weisemöller and Andy Schürr. Formal Definition of MOF 2.0 Metamodel Components and Composition. In *Proc. of International Conference on Model-Driven Engineering Languages and Systems*, pages 386–400. Springer, 2008.

[132] Ulrich Wolffgang. Multi-platform Model-driven Software Development of Web Applications. In *Proc. of International Joint Conference on Software Technologies*, pages 162–171, 2011.

[133] Rui Xu, Donald Wunsch, et al. Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[134] Xifeng Yan and Jiawei Han. gspan: Graph-Based Substructure Pattern Mining. In *Proc. of International Conference on Data Mining*, pages 721–724. IEEE, 2002.

[135] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. of the VLDB Endowment*, 3(1-2):340–351, 2010.

[136] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: Towards a Static Noninteractive Approach to Feature Location. *ACM Transactions on Software Engineering and Methodology*, 15:195–226, 2006.

[137] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines. In *Proc. of Symposium on Applied Computing*, pages 1064–1071. ACM, 2014.

[138] Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S Kolovos, and Richard F Paige. Type inference in flexible model-driven engineering. In *Proc. of European Conference on Modelling Foundations and Applications*, pages 75–91. Springer, 2015.

[139] Athanasios Zolotas, Nicholas Matragkas, Dimitrios S Kolovos, and Richard F Paige. Flexible modelling for requirements engineering. In *Proc. of Workshop on Flexible Model Driven Engineering*, page 32. CEUR-WS.org.

Typeset February 23, 2016