JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

# Automatic Model-based Management of Design Constraints

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Andreas Demuth

Angefertigt am:
Institut für Systems Engineering und Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed, MSc (Betreuung)
Assoz. Univ.-Prof. Mag. Dr. Wieland Schwinger, MSc

Linz, September, 2013

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 13. September, 2013

Andreas Demuth

# Abstract

Model-Driven Engineering (MDE) is an established paradigm in software engineering. It promotes the use of models as primary development artifacts, thereby raising the level of abstraction and allowing stakeholders to focus on those parts of a system that seem most relevant to them without being overwhelmed by its high overall complexity. However, using models as abstractions of real-world systems does impose the necessity of using defined syntax and semantic. In order to achieve positive effects through the application of MDE, it is crucial that models do comply to those notations and are semantically valid. Unfortunately, ensuring that only valid models are built is a non-trivial task. Although various approaches have been developed to address this issue, those approaches are often limited to specific scenarios or restrict the model designer or the modeling process significantly. In this thesis, we investigate an approach that proposes the incremental generation and management of model syntactic and semantic constraints based on existing design models. We show how our approach can provide assistance to the designer for reaching a valid model without restraining the designer or the development process being used. The approach is applied to the domains of model-transformation, metamodeling, and software product lines to demonstrates both its feasibility and applicability. Case studies with industrial-size models suggest that the approach scales and that it is suitable for providing instant feedback about a model's validity.

# Kurzfassung

Model-Driven Engineering (MDE) ist ein etabliertes Paradigma in der Softwareentwicklung, welches die Verwendung von Modellen als primäre Entwicklungsartifakte fördert. Die Verwendung von Modellen erhöht dabei die Abstraktionsebene und erlaubt Beteiligten, sich auf die für sie relevanten Teile eines Systems zu konzentrieren, ohne von der Komplexität des Gesamtsystems überfordert zu werden. Dies erfordert auch die Verwendung von definierten Notationen und Semantiken. Um MDE tatsächlich gewinnbringend einsetzen zu können, müssen Modelle den Notationen entsprechen und semantisch korrekt sein. Dies sicherzustellen ist jedoch eine hoch komplexe Aufgabe. Es gibt zwar zahlreiche Ansätze um dieses Problem zu lösen, jedoch sind diese Ansätze oft auf bestimmte Szenarien limitiert oder schränken den Modellierer oder auch den Modellierungsprozess signifikant ein. In dieser Arbeit erforschen wir einen neuen Ansatz, welcher inkrementelles Erzeugen und Managen von Modell-Restriktionen auf Basis existierender Modelle propagiert. Wir zeigen, dass der Ansatz dabei helfen kann, gültige Modelle zu erzeugen, ohne dabei die Freiheit des Modellierers oder des Modellierungsprozesses einzuschränken. Um die Machbarkeit und die breite Anwendbarkeit des Ansatzes zu zeigen, wenden wir ihn in den Bereichen Modell-Transformation, Metamodellierung und Software-Produktlinien an. Fallstudien mit praxisnahen Modellen zeigen, dass der Ansatz skaliert und Modellierern augenblicklich Rückmeldung über die Validität eines Modells gibt.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

The research was partially funded by the Austrian Science Fund (FWF) through the projects P21321-N15 and P23115-N23.

# 1 Introduction

Over the last decades, the concept of *Model-Driven Engineering (MDE)* [1] has emerged to become a principle commonly used in software engineering industry and also a very active field of research [2]. MDE encourages stakeholders to focus on abstract representations (i.e., models) of real-word systems and the individual entities they consist of instead of dealing with complex and very specific details of those entitites [3]. The developed models are then used as the primary source for constructing the desired systems (or at least parts thereof) automatically, eliminating various repetitive and time consuming tasks for developers. Therefore, applying MDE concepts can increase the efficiency of the development process [4, 5].

In fact, it has been shown that all major phases that occur in typical software processes are supported by MDE [6] and that MDE can lead to faster outputs that are of higher quality [4]. During requirements engineering, for instance, requirements models may be used to efficiently capture the functional and non-functional requirements stakeholders may want the desired product to provide. During the initial development phase, software architects build architecture models that define the overall structure of the software system and the patterns of communication within that system. In later phases, designers create design models that describe in much more detail the different elements from which the system can be composed of, the states of operations the system may operate in, and the interactions that occur within the system. Even those detailed models are still abstractions that are typically independent of programming languages and do not contain implementation details. For the testing stage, various approaches for performing tests on model representations of systems have been proposed. Finally, the deployment of a software system can also be modeled and then executed automatically.

## 1.1 Consistency Issues

As mentioned above, models are abstraction of real-world systems. In development projects, the used models most of the time are abstractions of not yet existing real-world systems. Thus, those models cannot be derived automatically (e.g., through reverse engineering [7]), but they have to be defined in a manual process, using people's knowledge and ideas. Furthermore, working with models and employing them as a means for more efficient communication [5] also requires the active use of abstraction concepts that allow for a simplified and straightforward representation of complex real-world scenarios – it requires a well-defined notation and also a common understanding of how that notation should be interpreted [8] – called a *metamodel* or *language* [9, 10]. Note that the interpretation of certain model elements (i.e., the model semantics) may depend on both the model's metamodel and the specific domain of the development project. Thus, the meaning of a model element often does not only depend on which kind of model element it is (e.g., its shape) and how such an element should be interpreted according to the metamodel, but it also depends on the domain in which the development project takes place – it is quite common in development projects to use defined languages for modeling but apply different rules for their interpretation. Depending on the tools employed for building models – those can range from pen and paper to highly sophisticated software tools or even modeling environments that make use of augmented reality concepts (e.g., [11]) – modelers may find themselves in situations where they do not encounter any restrictions regarding the model they build. For example, a modeler is free to use any notation imaginable when using pen and paper. When using a sophisticated modeling tool, on the other hand, he or she may be forced to use a small set of available modeling elements that comply to a defined notation.

In both cases, it is of the highest importance that the created models do follow defined notations. If they do not, it becomes impossible to share models for the sake of efficient communication as there is no common understanding and each individual may interpret certain – undefined – shapes differently. However, correct use of notations is not the only issue when creating models manually. Since models are abstractions of complex real-world systems, it is necessary to model different aspects of those systems in different views. This allows individual stakeholders to focus on specific parts of the system they are interested in. For instance, customers often are interested in internal data flows, while implementers of core business logic should not be bothered with user-interface design. Still, there is a connection. Both the interface designer and the programmer working on business logic will work with the same method that should be called in case of a certain event. If a designer now, for any reason, changes the name of that method, both the interface designer and the programmer will have to adapt their respective parts (or views) of the model they are working with. Otherwise, the model would still be correct with respect to the use of notation, but there would be inconsistencies between separate parts of the model (i.e., the model would be semantically invalid). Although this might not be an issue when only using models as a means of communication, such inconsistencies may prevent a model from being used for deriving (parts of) the modeled system's implementation. Clearly, the division of a single system into multiple views in MDE may increases complexity instead of actually reducing it [3].

To ensure that a model complies to a defined notation and is semantically valid, *model constraints* are required to precisely define conditions that must hold in a given model to be compliant. Note that some model constraints may be enforced on all models (e.g., syntactical constraints that check the correct use of notation) or only on some models that belong to a certain domain (e.g., domain-specific semantic constraints). Moreover, depending on the used modeling technology, some constraints may be enforced implicitly. For instance, a modeling tool might allow the user to draw only defined shapes. Thus, no model created with the tool will have ill-formed shapes. This is typically the case in modeling tools as they implement a fixed metamodel and only allow for instantiation of defined metamodel classes. Other constraints may be enforced explicitly. For example, based on the implemented metamodel's semantics, a popup notification may appear when a modeler tries to assign identical identifiers to different model elements and the modeler is forced to select a different value. Either way, the presence of a constraint – in any form – is mandatory. However, even sophisticated modeling tools (e.g., [12, 13]) often only enforce a small set of generic constraints that are not sufficient to ensure that constructed models are actually valid in a specific modeling context (i.e., a specific project) [14]. Note that the modeling context does not only involve the domain of the modeled system, but also the environment in which the modeling takes place (e.g., the company and its development process). This specific context requires additional syntactical and semantical constraints that are tailored to it. Indeed, this cannot be foreseen by tool authors and thus they cannot provide necessary constraints. Therefore, such additional, project-specific constraints usually have to be written manually. This makes them prone to errors and requires much effort to maintain when evolution takes place and models or also the modeling context are subject to change.

## 1.2 Thesis Overview

In this thesis, we investigate a novel approach that generates and manages constraints automatically based on existing design models. In particular, the approach ensures that at all times – even after the evolution of involved models – the desired set of model constraints is used for consistency checking. The rules defining which constraints are required and when they have to be generated are project-specific and can be tailored to different domains freely. Based on this maintenance of design constraints, inconsistencies can be detected reliably and further user-guidance for repairing them and building a consistent model can be provided.

To validate this approach, several questions must be answered. First, we need to demonstrate that automatic generation and management of constraints using information from existing models is feasible. That is, we have to investigate whether common models do provide enough information for generating constraints that ensure syntactical and semantical correctness of a model. Second, it must be shown that both the generation and the adaptation of constraints can be done incrementally and efficiently during modeling without interrupting users of modeling tools. Since constraints only provide information once they have been validated, this also includes the efficient chaining of constraint management and constraint validation. Third, we have to demonstrate that constraint generation and the automation of constraint management does address important issues in MDE. Finally, in order to validate the general applicability of the approach, it must also be investigated whether it can be implemented in different ways that meet the specific needs of different domains.

### *1.2.1 Research Questions*

Based on those goals, we define the four specific research questions:

- **RQ1:** Is it feasible to generate and manage relevant constraints automatically?

- **RQ2:** Can constraint management be done incrementally and efficiently?
- **RQ3:** Is constraint management capable of addressing important MDE issues?
- **RQ4:** Is the approach generally applicable and can it be tailored to different domains?

### *1.2.2 Organization*

The remainder of this thesis is organized as follows. In Chapter 2 (based on [15] and [16]), we discuss the core concepts required for generating constraints from models using a typical model-driven engineering example. Additionally, we show how ambiguity and missing information may cause issues when generating models automatically or using consistency preserving approaches to handle model evolution. It is demonstrated how our approach enables a different, constraint-driven and user-centric way to reach not only consistent but actually intended models when it is not possible to obtain those models automatically.

In Chapter 3 (based on [17]), the principles developed in Chapter 2 are applied to the domain of metamodeling. We illustrate how constraints are commonly based on metamodel information and discuss why the co-evolution of metamodels and constraints is necessary when applying MDE-concepts in evolving domains. We will also show that our approach supports different ways of constraint generation: while model transformations are used in Chapter 2, automatically instantiated constraint templates are used in Chapter 3.

In Chapter 4, we apply our approach of constraint generation to the domain of Software Product Lines (SPLs). We present a formal theory for incremental consistency checking of individual products in SPLs, demonstrating further that the approach can be applied to different domains and implemented in different ways. A case study shows that generating constraints for individual software products is more efficient than existing product checking approaches.

Each chapter includes a motivation section in which the problem domain is discussed, an example typical for the respective domain that illustrates the problem to be addressed, an extensive description of the proposed solution in general as well as a detailed discussion of how the solution is applied to the representative example, and a validation of the presented concepts that includes a realistic case study and performance analysis. Moreover, each chapter concludes with a detailed discussion of related work in the respective field.

Chapter 5 then revisits the research questions defined above and discusses the results thoroughly. Moreover, the chapter looks into possible threats to validity. Finally, it gives a short overview of planned future work and briefly summarizes the findings of this thesis.

# 2 Constraint-Driven Modeling

With the broadening use of *Model-Driven Engineering (MDE)* [1] for complex software systems, the generation of models from existing artifacts through *model transformation* [18] is a vital necessity. Various classifications and taxonomies have been published to compare the state-of-the-art (e.g., [19, 20, 21]). Rich transformation languages are available, such as the *ATLAS Transformation Language (ATL)* [22], *Query/View/Transformation (QVT)* [23] or *Triple Graph Grammars (TGG)* [24], which define *transformation rules* that are executed by a *transformation engine.* The transformation engine generates a target model from a source model, typically through a series of transformation rules. To date, various sophisticated transformation techniques exist that produce excellent results as long as the generated models are static (i.e., they are not adapted manually) and there are no uncertainties involved (i.e., there is exactly one possible target model) [25, 26]. However, model transformation has yet to overcome several key challenges [20, 27, 28]:

- How to allow transformation without overwriting changes made by the designer?

- How to transform in the face of uncertainty where multiple target models satisfy a given source model?
- How to support consistency when using bi-directional transformations?

Traditional model transformation either generates a target model out of a source model if none existed or it overwrites the existing target model. The latter is only desired if overwriting the target model does not lead to information loss – a problem when the designer edited the target model prior to transformation. In an idealized model transformation scenario, the target model is generated but never modified after transformation (later re-transformation does not lead to loss) or the source model will not be modified after transformation (avoiding later re-transformation). There are some situations where models are stable enough for this idealized scenario to apply but in context of iterative/incremental development, designers tend to change models continuously [25]. In such cases, a complete re-transformation may seriously affect the designer's normal workflow [29, 30] if they do overwrite changes made manually beforehand. Hence, incrementality is required to allow partial model transformation in order to limit such conflicts [30]. Although model-synchronization as well as incremental and in-place transformation approaches have been proposed (e.g., [31, 32, 33, 34, 35, 36]), those approaches typically rely on defined strategies for handling conflicts and re-establishing consistency. Moreover, there are situations in which these approaches cannot re-establish consistency [36, 37]. Thus, it cannot be guaranteed that they produce the results that are actually intended.

The problem of re-transformation is also related to another problem: that of uncertainty. Manual changes to a target model are necessary if the information present in the source model is insufficient to generate the target model. There are two possible reasons for this: 1) the source model is incomplete, and/or 2) the semantic differences between the source and target models make it impossible to generate the

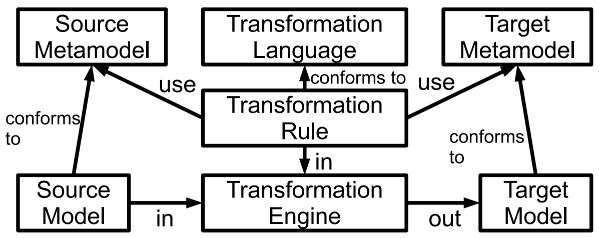


FIGURE 2.1: Traditional model transformation approach

target model in its entirety. Either reason implies uncertainty and current state-of-the-art is either not used in these cases or the transformation implements a heuristic that automatically decides. This severely reduces the usefulness of transformation in the presence of incomplete information or uncertainties: either because model transformation is not applicable (leading to less automation) or because transformation generates a potentially unintended target model (requiring manual addition and changes to the target model after transformation). The issue is similar with *bidirectional transformations* [28, 38], which are often used to synchronize models or to keep them consistent, when both involved models are edited concurrently. As we will show later, these issues might limit the usefulness of model transformation in certain situations.

In this chapter, which is based on [15] and [16], we introduce *Constraint-driven Modeling (CDM)*, a generic approach that guides the construction of of desired target models – either from scratch or from an existing but incomplete or undesired target model – relying on model constraints and detected inconsistencies. In particular, this chapter will discuss the key requirements and possibilities of incremental constraint generation and management. Moreover, this chapter will illustrate how the approach can address the issues of lost updates and missing information that may arise with the use of model transformations during design model development.

CDM does not generate a target model out of a source model, as it is done by traditional model transformation approaches (see Fig. 2.1). Instead, CDM gener-

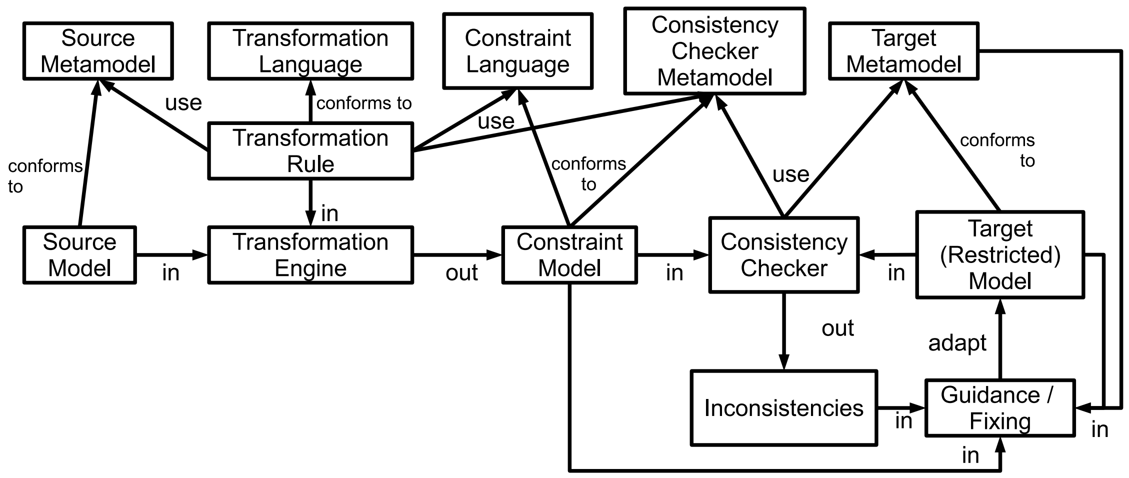


Figure 2.2: CDM approach overview

ates *constraints* from the source model that express conditions that must hold in the target models, as shown in Fig. 2.2. Those constraints are not part of the target models but are passed to a dedicated consistency checker that validates them on the target model. CDM thus does not overwrite or adapt the target mode and, as will be shown, uses constraints to address the issues of lost updates because of re-transformation, incomplete information because of uncertainties, and race conditions during model-sychronization and when using bidirectional transformations. The generated constraints, written in a *constraint language* (e.g., the *Object Constraint Language (OCL)* [39] or *First-Order Logic* [40]), are then continuously validated by a *constraint checker* on the target model. The constraint checker provides the designer with feedback on constraint violations for *guidance*. Constraint violations in turn help the designer to stepwise adapt the existing target model to a target model that satisfies all constraints. Indeed, CDM may also be chained with existing state-of-the-art technologies for repairing constraint violations (e.g., [41, 42]) – with or without designer interaction. These existing technologies provide guidance on what to repair and, at times, even how to repair in order to achieve a correct target model. Indeed, if the constraints are only satisfiable in a single target model then these repairs should be able to compute this model. Here, the advantage is that these repair mechanisms do consider the state of the target model as part of the repair – including any manual changes made before or after the transformation. Moreover, the generated constraints may serve as input for a reasoning engine to generating (parts of) a target model.

Thus, CDM should be seen as a complement to traditional model transformation and model generation approaches; or it could also be seen as an alternative in concert with repair technologies.

We evaluated our approach and showed its feasibility by implementing a prototype tool that generates constraints, updated and enforces them incrementally, and informs the designer about existing constraint violations (i.e. inconsistencies). It should be noted that CDM does not prescribe a particular constraint checking or repair mechanism. To demonstrate the possibility of efficiently chaining constraint generation and validation, we incorporated one such mechanism, called the Model/-Analyzer [43], a scalable and incremental constraint checking mechanism which has been validated on numerous industrial models [44]. However, any other technology for constraint checking and repairing may be used instead. The use of the Model/-Analyzer is merely to demonstrate that there is at least one such technology available already. Since the Model/Analyzer has a proven scalability record in terms of handling changes of the checked model [44], the evaluation will focus on the correctness and scalability of the generation of target model constraints from a source model and their initial validation. We will show in a case study that performing incremental constraint transformation for UML design models does not impose significant losses in performance when used live during modeling. Moreover, we will demonstrate that the subsequent constraint validation times are similar to those observed in [44]. Our findings show that the approach scales and also provides instant feedback when involved (source) models are edited.

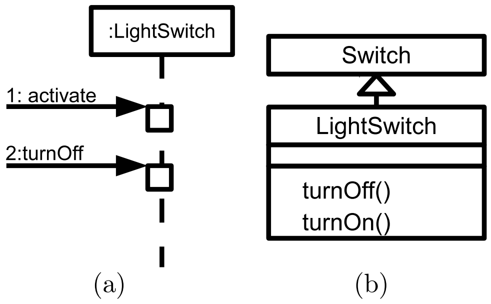


FIGURE 2.3: Two UML models (a) and (b)

## 2.1 Motivating Example – Transformation Ambiguity

To illustrate our work, we first present a scenario that is challenging for common model transformation approaches. Let us consider the sequence and class diagrams – defined with the *Unified Modeling Language (UML)* [45] – shown in Fig. 2.3(a) and Fig. 2.3(b) respectively. In Fig. 2.3(a), the unnamed instance of class `LightSwitch` receives a message named `activate`. A common interpretation of UML sequence diagrams is that this message requires that the instance of `LightSwitch` provides a method named `activate` also [46, 47, 42, 48, 41]. At first glance, a simple transformation appears to solve the problem by automatically adding the method `activate` to class `LightSwitch` in Fig. 2.3(b) whenever a message is added to a sequence diagram whose name does not match any method in the class. An example for such a transformation rule written in ATL is shown in Listing 2.1. The rule is executed for every `Message` instance (defined in the `from`-block) for which it generates a `Method` instance (defined in the `to`-block). It assigns a name and an owner derived from the `Message` object for which it is executed to the attributes `name` and `owner` of the generated `Method`.

However, there is an issue with this approach: Should the method `activate` be added to `LightSwitch` or would it make more sense for the system to add it to the superclass `Switch`? Obviously, this question is hard to answer automatically.

```
from
  s : SequenceDiagram!Message
to
  t : ClassDiagram!Method (
    name <- s.name,
    owner <- getClass(s.receiver.className)
  )
```

Listing 2.1: Sample transformation to generate methods in class diagrams

One possibility would be to make an assumption. For instance, always add the method to the target class, `LightSwitch` in our example. This option can lead to the generation of potentially unintended models where methods are not declared in the desired place or where methods are unnecessarily overridden. Another option is the use of heuristics to find the most suitable solution for the problem. However, heuristics typically employ metrics to evaluate possible solutions. Thus, heuristics may chose solutions that are optimal with respect to defined criteria but that may not be seen as optimal by designers. Note that these design decisions are typically made when defining the modeling domain and its semantics, not at the time the actual system under development is modeled. Therefore, those decisions cannot take into account the characteristics of the model which they actually affect.

## 2.2 Approach

Common transformation languages usually describe the steps that have to be performed to generate new models from existing ones. However, the previous section illustrated that it can be difficult or even impossible to write transformation rules that automate complex decisions.

### *2.2.1 Core Principles*

In contrast to standard model transformations, we propose to generate constraints on a model (to guide designers or enable automatic approaches that establish consistency) rather than generating the model itself whenever precise transformation results cannot be derived. For example, the added message `activate` on the source model imposes – according to the common semantics discussed above – a constraint that a same named method should be available to class `LightSwitch` rather than saying it should be owned by it. If the method is already there then the constraint is instantly satisfied. If the method does not exist then the constraint's violation will identify this lack and further actions are required to deal with this problem – actions that must either come from a human or be derivable from the constraint violation (inconsistency) through reasoning (e.g., [49, 42]). Note that this delays the design decision from a point in time at which nothing was known about the specific system to a point at which designers already have knowledge about the specific system that is modeled.

When traditional model transformation approaches are used (as depicted in Fig. 2.1), the transformation process can be regarded as:

$$A \xrightarrow{T_m} B_g \tag{2.1}$$

where $A$ is called the source model that conforms to a source metamodel and consists of an arbitrary number of model elements. $T_m$ is a set of transformation rules (called the *transformation model*), that is used to transform $A$ to the generated (denoted by the subscript $_g$) model $B_g$, which conforms to the target metamodel. Transformation rules are based on both source and target metamodels and are executed by a transformation engine.

We expanded this notation and define our approach as:

$$A \xrightarrow{T_c} C \rightsquigarrow B_r \tag{2.2}$$

where the variable $A$ denotes the source model and $T_c$ is a set of model transformation rules. However, as the solid arrow from $A$ to $C$ and the changed subscript $_c$ of $T$ indicate, the transformation model no longer generates a model (i.e., $B_g$), but instead it contains different transformation rules that are applied to $A$ in order to generate a set of constraints using information available in $A$, the constraint model $C$ which conforms to a constraint metamodel (i.e., a constraint language), as depicted in Fig. 2.2.[1][2] This constraint model consists of a set of constraints that are enforced by an incremental consistency checker on the model $B_r$, as indicated by the curvy arrow from $C$ to $B_r$. The model $B_r$ is no longer the generated model but is now called the restricted model, as indicated by the subscript $r$, that is either consistent or inconsistent with the constraint model $C$, and therefore a valid or invalid solution of the modeling problem. Note that in CDM the (generated) constraint model is separated from the target model (i.e., the generated constraints are not part of the target model).

Note that an initial version of $B_r$ may be generated through a traditional transformation (analogous to $B_g$), through automatic metamodel instantiation that is guided by the generated constraints, or even built manually by a designer. However, once generated, this proposed approach can detect inconsistencies if both A and $B_r$ are evolved concurrently. Thus, our approach should not be seen as replacing traditional transformation approaches but instead complementing them in case of uncertainties and, as will be demonstrated below, also co-evolution, complex rule-scheduling issues, or even model merging. Next, we present how it is applied.

### *2.2.2 Application: Uncertainties*

Let us come back to our running example from Section 2.1 where we illustrated that choosing the right class for a required method cannot be fully automated. The tradi-

[1] This is different from existing approaches (e.g., [50, 51]) that derive constraints through interpretation or translation of transformation rules which generate the target model. In CDM, transformation rules are executed by standard transformation engines to generate constraints – there is no interpretation or translation of rules involved.

[2] CDM supports transformation rules that use source model information only (i.e., they do not read information from the constraint model).

```
rule t1
  from
    s : SequenceDiagram!Instance
  to
    t : ConstraintModel!Constraint (
      context <- "Package",
      inv <- "self.classes->exists(c|c.name='" + s.className + "')"
    )
rule t2
  from
    s : SequenceDiagram!Message
  to
    t : ConstraintModel!Constraint (
      context <- "Class",
      inv <- "self.name='" + s.receiver.className + "' implies self.providedMethods->
          exists(m|m.name='" + s.name + "')"
    )
```

Listing 2.2: Transformation rules to generate class (t1) and method (t2) constraints

tional approach shown in Fig. 2.4(a) automatically generates one of several possible models and we could at most use heuristics for deciding on which transformation to use (which never guarantees intended results). However, while the knowledge contained in Fig. 2.3(a) is insufficient to generate an update of the class diagram that will always lead to the intended solution, it is sufficient to generate a correct constraint on that diagram. Such constraints can be generated incrementally by transformation rules whose execution is triggered by the addition/removal of class instances or messages in sequence diagrams that can be efficiently validated by state-of-the-art consistency checkers.

To automate constraint generation in our example, we define two transformation rules that are triggered by class instances or messages in sequence diagrams and that

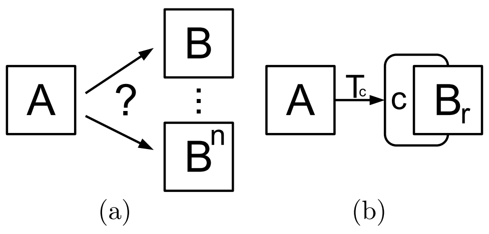


FIGURE 2.4: From ambiguous model transformation (a) to constraint transformation (b)

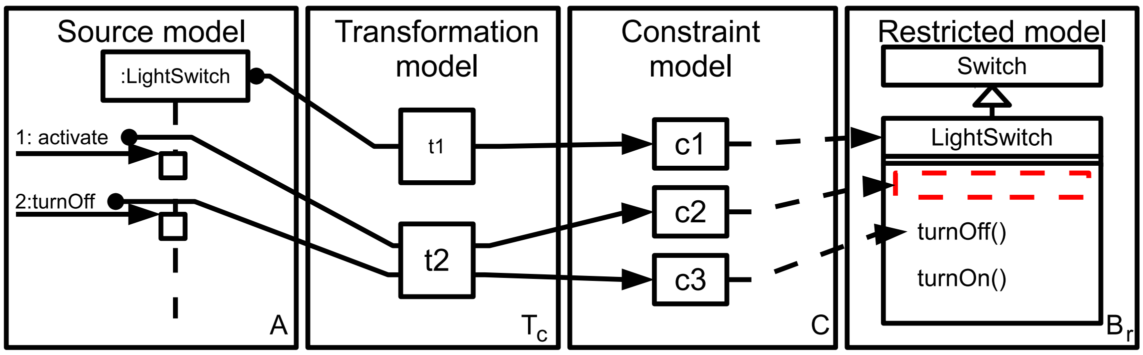


FIGURE 2.5: Application of CDM to models from Fig. 2.3(a) and Fig. 2.3(b)

use information from the sequence diagram to generate very specific and expressive constraints. These rules are shown in Listing 2.2. The rule $t1$ takes an instance specification (e.g., a lifeline) and generates a constraints that requires the model's base package to provide a class with a matching name. Rule $t2$ is triggered by a message and generates a constraint that requires the message's receiver class to provide a method with a name equal to the message name. Note that, even though we use ATL-like syntax for this example, our approach can be used with any transformation language. After applying these rules to the motivating example from Section 2.1 as illustrated in Fig. 2.5 and according to Eq. (2.2), $C$ consists of the following OCL constraints:

**c1** `context Package inv:`
```
    self.classes->exists(c|c.name='LightSwitch')
```

**c2** `context Class inv:`
```
    self.name='LightSwitch' implies
    self.providedMethods->exists(m|m.name='activate')
```

**c3** `context Class inv:`
```
    self.name='LightSwitch' implies
    self.providedMethods->exists(m|m.name='turnOff')
```

In Fig. 2.5 we can see that the method required by the constraint $c2$ is not present in $B_r$, as indicated by the empty, dashed rectangle in the `LightSwitch` class, meaning that this particular model will be marked inconsistent. Note that we use OCL as the constraint language in our example because it is a well known and accepted language for writing constraints for design models and we have existing tool support for incrementally validating OCL constraints and repairing them. Nonetheless, in principle any constraint language and consistency checker may be used (e.g., [52]).

Fig. 2.4(b) illustrates the basic concept of the constraint-driven modeling approach. It is noteworthy that the approach does not modify the restricted model. It simply defines constraints that restrict it. The generated restriction – depicted as partial frame with rounded corners around the restricted model $B_r$ – may be light in that there may be various options on how to change the restricted model to satisfy the constraints - hence supporting uncertainty. In such as case, the designer – or also a fully automated approach that uses heuristics – has the freedom to decide which of the options to select (e.g., add `activate` to `LightSwitch` or `Switch`) with the knowledge that the approach prevents options that are invalid. In the most extreme case, the restrictions may be severe enough to allow for one option only. In this case, the only remaining option could be chosen automatically, like in traditional transformation, to modify the target model $B_r$. Note that in contrast to heuristics-based model transformation, an approach that derives possible model adaptations automatically after the generation of constraints may not only use a defined strategy and the target model itself as input for finding solutions, but it may also consider the generated constraints. Thus, more information is available for reasoning.

Fig. 2.6 shows the visual notations we use throughout this chapter to illustrate

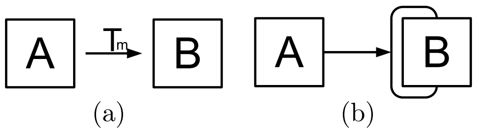


FIGURE 2.6: Traditional (a) and constraining (b) approach

the differences between common model transformation and our approach. Fig. 2.6(a) is equivalent to Eq. (2.1). Fig. 2.6(b) shows the notations for our approach. Note that the constraint model $C$ and the curved arrow to the restricted model in Eq. (2.2) are replaced by a partial, restricting frame around $B$; $T_c$ is not depicted.

### *2.2.3 Incremental Constraint Model Management*

Let us take a closer look at the transformation that generates the constraint model $C$. As shown in Eq. (2.2) and Fig. 2.5, applying the transformation rules of the transformation model to the source model generates the constraint model.

The transformation approach we use supports incrementality through the automatic generation of trace information during the execution of transformation rules to allow updates of both the source model and the transformation model without performing a complete re-transformation. When a transformation rule is executed, our approach captures all source model elements that are accessed during the execution and in particular which source model element matched the rule execution context and thus triggered the execution. This means that updates of the source model or the transformation model do only lead to a partial, operation-based update of the constraint model (i.e., the existing constraint model is adapted instead of generated from scratch). Indeed, those partial constraint model updates may change the consistency status of a restricted model.

*Source model update.*

When the source model $A$ is updated to $A'$, we can write this as

$$A \xrightarrow{\Delta A} A' \tag{2.3}$$

where $\Delta A$ is a sequence of modifications (i.e., $\langle \text{model element}, \text{action} \in \{add, remove, update\} \rangle$) done to $A$ (e.g., add a new model element or update an existing element). Based on $\Delta A$ and the transformation model $T_c$, the set $\Delta C$ can be generated, as shown in Eq. (2.4).

$$\Delta A \xrightarrow{T_c} \Delta C \tag{2.4}$$

Specifically, matching transformation rules from the transformation model are executed to produce newly required constraints for added source model elements and constraints that were generated from removed source model elements must be removed from the constraint model. The constraints to be removed are determined by using the trace information generated during rule execution. For source model element updates, trace information is used to find the constraints that have been generated using now outdated source model information. For those constraints, the transformation is re-executed and the previously generated constraints have to be removed from the constraint model. $\Delta C$ includes pairs of constraints and actions (i.e., $\langle \text{constraint}, \text{action} \in \{add, remove\}\rangle$) that define whether the constraint should be added or removed from the existing constraint model $C$. By applying $\Delta C$ on $C$, the updated constraint model $C'$ is generated:

$$C \xrightarrow{\Delta C} C' \tag{2.5}$$

Specifically, for a pair $\langle \text{constraint}, add\rangle$, the constraint is added to the constraint model and for a pair $\langle \text{constraint}, remove\rangle$ the constraint is removed from the constraint model. Let us consider the evolution of the models shown in Fig. 2.3(a) and Fig. 2.3(b) to the versions shown in Fig. 2.7(a) and Fig. 2.7(b) where the name of the message #2 was updated to `deactivate`, the message #3 was introduced, and the name of the method `turnOff` was changed to `switchOff`. For the changes in the source model, the corresponding $\Delta A$ is $\langle\langle Message2, update\rangle, \langle Message3, add\rangle\rangle$.

To build $\Delta C$, the transformation engine executes those transformation rules that use elements in $\Delta A$ (i.e., message #2 and message #3) to generate the corresponding

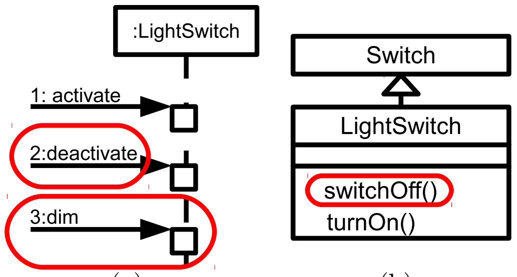


FIGURE 2.7: Evolved versions of Fig. 2.3(a) and Fig. 2.3(b)

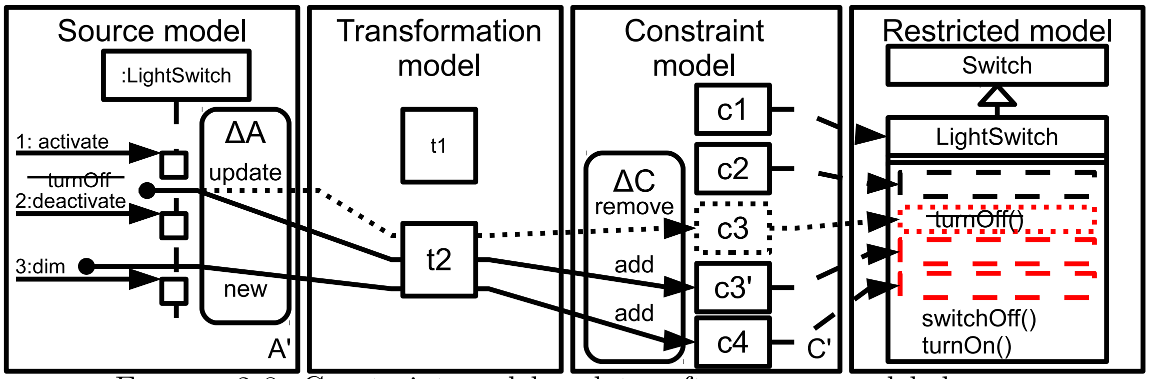


FIGURE 2.8: Constraint model updates after source model changes

constraints, as defined in Eq. (2.4) and shown in Fig. 2.8. For $\langle Message2, update \rangle$, the constraint $c3'$ is generated and the information $\langle c3', add \rangle$ is added to $\Delta C$.

**c3'** `context Class inv:`

```
    self.name='LightSwitch' implies
    self.providedMethods->exists(m|m.name='deactivate')
```

Since the constraint $c3$ was already generated from the same element as $c3'$, message #2, $\langle c3, remove \rangle$ is also added to $\Delta C$ in order to remove the now outdated constraint $c3$. Note again that the previous generation of $c3$ through the execution of $t2$ with message #2 can be found as that rule execution generated corresponding trace information. For $\langle Message3, add \rangle$, the transformation rule $t2$ is executed to generate a new constraint $c4$ and $\langle c4, add \rangle$ is added to $\Delta C$.

**c4** `context Class inv:`

```
self.name='LightSwitch' implies
self.providedMethods->exists(m:Method|m.name='dim')
```

Note that in this example, both elements in $\Delta A$ are used as context of a transformation. However, if an element in $\Delta A$ was accessed in any way during the execution of a transformation rule, the transformation also has to be re-executed (e.g., $c3$ would have been updated to $c3'$ also if not the message name but the receiver's name would had changed). At this point, $\Delta C$ is $\{\langle c3, \mathit{remove}\rangle, \langle c3', \mathit{add}\rangle, \langle c4, \mathit{add}\rangle\}$.[3] When these changes are applied to $C = \{c1, c2, c3\}$ as defined in Eq. (2.5) and shown in Fig. 2.8, the resulting updated constraint model is $C' = \{c1, c2, c3', c4\}$. We used dotted lines for removed elements, that is $c3$ and the corresponding inconsistency in `LightSwitch`. As Fig. 2.8 indicates, the constraints $c3'$ and $c4$ are violated by the restricted model since the class `LightSwitch` does not provide the required methods `deactivate` and `dim`. Note that, as discussed above and shown in Fig. 2.2, the constraint model is generated incrementally and separated from the restricted model. If constraints are added to the constraint model manually by designers, those constraints remain unchanged by our approach due to the nature of our incremental constraint model updates and the fact that for manually added constraints there is no trace information available. However, manual updates of generated constraints are currently not supported and may be overriden during constraint updates.

*Transformation model update.*

If the transformation model $T_c$ is updated to $T_c^{'}$, the changes $\Delta T_c$, which have the form $\langle \text{transformation rule}, \text{action} \in \{\mathit{add}, \mathit{remove}\}\rangle$, are derived – based on existing

[3] If the re-execution of a transformation rule updates an existing constraint ($c_x$) rather than re-generating it, then both the outdated version (i.e., $c_x$) and the updated version (i.e., $c_x^{'}$) identify different version of the same object (i.e., the version $c_x$ is lost as a consequence of the update). In this case, the constraint is removed and added back to the constraint model, causing the validation of the constraint with the updated information (i.e., the necessary re-validation). Such updates may also be handled automatically by some consistency checkers.

trace information – and used to generate $\Delta C$ from $A$, as shown in Eq. (2.6).

$$A \xrightarrow{\Delta T_c} \Delta C \tag{2.6}$$

$\Delta C$ can then be used as shown in Eq. (2.5) to update the constraint model to $C'$.

Based on the model versions after the discussed source model update, let us consider further changes – this time to the transformation model – as depicted in Fig. 2.9: the removal of the transformation rule $t1$ and the addition of the new transformation rule $t3$, which is triggered by elements of the UML type `Instance`. $\Delta T$ in this case is $\langle\langle t1, \mathit{remove}\rangle, \langle t3, \mathit{add}\rangle\rangle$. For the removal of $t1$, the constraints that have been generated for this rule are removed from the constraint model and no transformation rules are executed. Since traces reveal that $c1$ is the only constraint generated by $t1$, $\langle c1, \mathit{remove}\rangle$ is added to $\Delta C$. For the addition of $t3$, the transformation executes just this transformation rule and produces the new constraint $c5$ and adds $\langle c5, \mathit{add}\rangle$ to $\Delta C$, which finally is $\{\langle c1, \mathit{remove}\rangle, \langle c5, \mathit{add}\rangle\}$.

**c5** `context Package inv:`
```
self.ownedElements->exists(c:Class|
c.name='LightSwitch')
```

By executing $\Delta C$, the existing constraint model $C = \{c1, c2, c3', c4\}$ becomes $C' = \{c2, c3', c4, c5\}$.

Ultimately, changes of the source model $A$ affect the constraints that are enforced by the consistency checker:

$$C' \rightsquigarrow B_r \tag{2.7}$$

Next, we describe how such constraint model changes can affect the consistency status of the restricted model $B_r$.

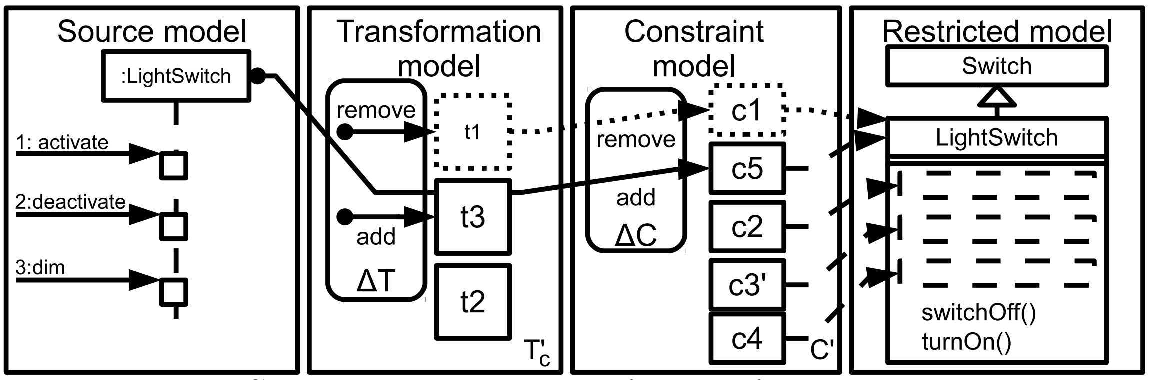


FIGURE 2.9: Constraint model updates after transformation model changes

### *2.2.4 Constraint Validation and Solution Space*

We define the *solution domain* of a modeling problem to include all possible instances of a metamodel (there are likely infinite). The *solution space* includes all valid models of the solution domain. Constraints define certain criteria that valid models must meet. Validating a constraint on a specific model determines whether the model meets those criteria – if it does not, it is not part of the solution space. Therefore, applying a constraint decreases the size of the solution space. For example, Fig. 2.10(a) shows that the solution space in our running example is reduced to the specific models $B1$, $B2$, $B3$, and $B5$ (drawn as black filled circles) when the constraints $c1$ - $c4$ are applied after the source model changes performed in Section 2.2.3. In our illustrations we show the remaining solution space towards the center of the circle, models outside the solution space are drawn as an unfilled circle. Note that this simplified view is only used to make the illustrations readable. Moreover, we assume that there is only a limited number of possible models for the illustration – in practice the number of valid models may remain infinite.

We define the validation of a constraint $c$ for a specific model $m$ as $val : (m, c) \rightarrow \{false, true\}$ where $false$ is returned if $m$ violates $c$, $true$ otherwise. For a restricted model $B_r$ and a constraint model $C$, the result of a total validation (i.e., a validation of all available constraints, written as $val_T$) would then be equal to:

$$val_T(B_r, C) = \bigwedge_{1 \leqslant i \leqslant |C|} val(B_r, c_i) \tag{2.8}$$

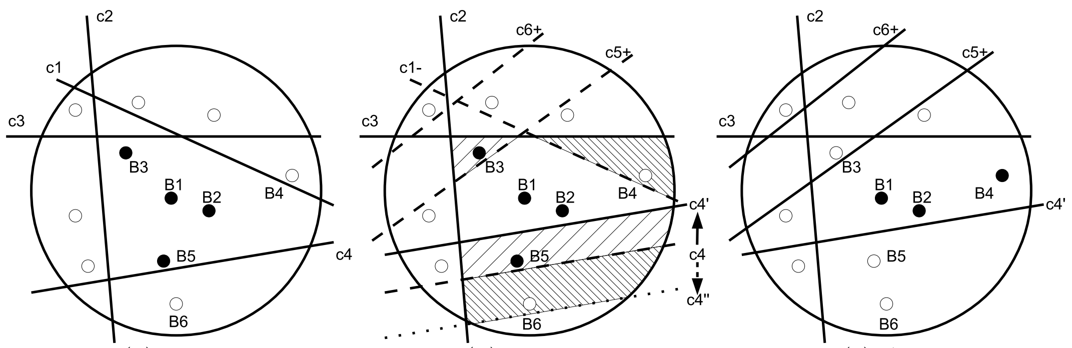
If at least one constraint validation $val(B_r, c_i)$ returns $false$, the overall status of $B_r$ is also $false$ and therefore the model is outside the solution space. It is easy to see that **the order of constraint validation does not affect the final result**. However, the execution order determines when the overall inconsistency of a model $B_r$ is discovered during the validation and the order in which inconsistencies are corrected can of course be important when deriving stepwise adaptations.

Constraints are composed of expressions that belong to exactly one constraint (i.e., information is not shared between constraints) and that are evaluated on the restricted model only. Therefore, *structural dependencies* between constraints do not exist and are not considered here. The addition of a new constraint thus does not affect the validity of existing constraints as they are not connected structurally. This leads us to the conclusion that **constraints are structurally independent of each other**.[4] Furthermore, the used transformation rules do only access the source model to construct constraints and add the constraint to the constraint model without accessing other constraint model elements, thus the **transformation rules for generating constraints are independent** and dependencies between them that require a certain order of execution cannot occur. These observations have interesting benefits to model transformation discussed next.

When the constraint model is changed, so need to be the restrictions imposed by it. There are three possible constraint model changes, which we will show next: i) addition, ii) removal, and iii) update of constraints.[5]

[4] Note that structural independence is a general property of constraints and is not limited to specific constraint languages – in contrast to, for example, OCL's property of being side-effect-free.

[5] In practice, a constraint update may be done by a consecutive removal and addition of constraints.



(a) Before (b) Changes (c) After

FIGURE 2.10: Effects of constraint changes on solution space

*Constraint addition.*

When new constraints are added to the consistency checker, the solution space either stays unmodified or is narrowed, as illustrated in Fig. 2.10. The addition of $c5$ in Section 2.2.3, which requires an element of type `Class` and with the name `LightSwitch` must be owned by a `Package`, does reduce the remaining solution space (indicated by the thinly shaded area around $B3$ in Fig. 2.10(b)) so that the previously valid model $B3$ is removed from the solution space because there the class `LightSwitch` did not provide such a method, as shown in Fig. 2.10(c). The manual addition of a new constraint $c6$, which requires a UML `Package` to own at least one element does not change the remaining solution space because there are other constraints (e.g., $c5$) that impose stronger restrictions than $c6$.

**c6** `context Package inv:  self.ownedElements->size()>0`

As Eq. (2.9) shows, we can easily split the complete evaluation of the new constraint model $C'$ into the validation of the old constraint model $C$ and only those parts of $C'$ that were actually added (i.e., $C' \backslash C$).

$$\begin{aligned} \bigwedge_{h=1}^{|C'|} val(B_x, c'_h) = & \bigwedge_{i=1}^{|C|} val(B_x, c_i) \wedge \\ & \bigwedge_{j=1}^{|C' \backslash C|} val(B_x, (C' \backslash C)_j) \end{aligned} \tag{2.9}$$

Assuming that the validation result for $C$ is still available, we can see that only the last part of Eq. (2.9) has to be evaluated to determine whether a specific model $B_x$ is part of the updated solution space (i.e., a valid model). Obviously, this validation is not even relevant if the old validation result was already *true*.

As we have already discussed, the order of constraints is not relevant for the final result of the consistency checker. Therefore, the order in which constraints are added to the constraint model is not relevant for the final consistency status of a model either.

*Constraint removal.*

The removal of constraints can increase the size of the solution space. Fig. 2.10 shows the removal of constraint $c1$, as discussed in Section 2.2.3. This change means that the model $B4$ becomes part of the solution space (as shown in Fig. 2.10(c)), which was increased by the densely shaded area around $B4$ in Fig. 2.10(b). Intuitively, it is not possible that the removal of a constraint narrows the solution space. After constraints are removed, the restricted model is validated with the remaining constraints. As we have seen, the order of the constraints has no effect on the final validation result.

In contrast to the addition of constraints, the basic consistency checking mechanism we used in Eq. (2.8) cannot be split into parts to reduce calculation effort since we do not know the result of the remaining constraints. However, if we capture the constraints that were inconsistent during the validation of $C$, we can define a function $ic(B_x, C) = \{c | c \in C \wedge val(B_x, c) = false\}$ that returns exactly those constraints.

After removing constraints from $C$ to generate $C'$, $C \backslash C'$ is the set of constraints that are removed from the constraint model. If $ic(B_x, C) \subseteq C \backslash C'$ holds, the model $B_x$ is a valid solution after the constraint model update since all constraints that previously caused an inconsistency are no longer part of the constraint model. Whether the function $val_T$ or the set comparison is used to determine the new consistency status of a model, either way the order of removal does not affect the final result.

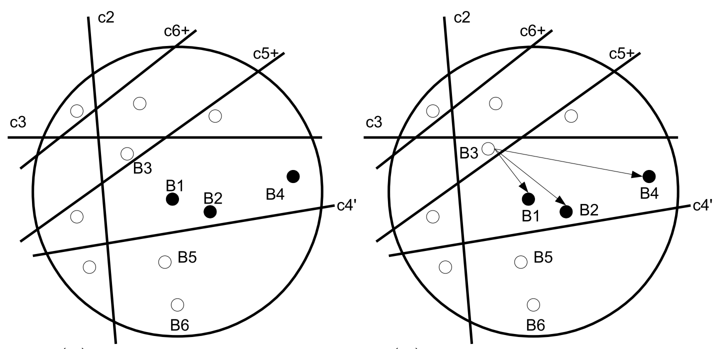
*Constraint update.*

We have defined that a constraint update consists of the removal of old and the addition of new constraints. Therefore, such an update can either increase or decrease the size of the solution space, depending on the specific update. The constraint $c4$ is based on a message in a sequence diagram and it requires the `LightSwitch` class to provide a `Method` with the name `dim`. If the constraint is updated so that it requires an instance of a more specific class, for example an instance of `AsyncMethod` instead of `Method`, $c4$ is updated to $c4'$, the solution space size is decreased (indicated by the thinly shaded area around $B5$ in Fig. 2.10(b)) and $B5$ becomes inconsistent because the class only provides a `Method` object. However, Fig. 2.10(b) also illustrates a possible update to $c4''$ (indicated by the dotted line) where the required object is allowed to be an instance of a more general class (e.g., `NamedElement`). This update would increase the solution space by the densely shaded area and make $B6$ consistent.

After performing the addition of $c5$ and $c6$, the removal of $c1$, and the update of $c4$ to $c4'$ the updated solution space consists of $B1$, $B2$, and $B4$ as shown in Fig. 2.10(c).

## 2.3 Providing Guidance and Executing Fixes Automatically

When an inconsistency is detected, the minimum amount of guidance provided to the designer is a notification about the inconsistency's occurrence and its location



(a) Without guidance (b) With guidance for $B3$

FIGURE 2.11: Adding guidance to transform inconsistent models to consistent ones

(i.e., which model element violates which constraint). Based on data captured during constraint validation, consistency checkers can determine which model elements are actually causing the inconsistency [42]. Hence, the designer can be informed about the locations of error-causing elements.

Constraint-driven modeling may appear inferior to traditional transformation in that it does not generate or update model elements in the restricted model. However, there is currently considerable progress in suggesting repairs to individual inconsistencies in design models (e.g., [49, 53, 42, 54, 55, 56, 57]). Based on a specific constraint (or a set of constraints) and the inconsistent parts, it is thus possible to derive modifications – like specialized transformations – that lead to a consistent model. If such modifications can be derived, they are proposed to the designer as a list of options, or a suitable solution may be chosen automatically by applying heuristics. In Fig. 2.11, the general idea of fixing is illustrated for the inconsistent model version $B3$. The solution space for the modeling problem is depicted in Fig. 2.11(a). In Fig. 2.11(b), the derived fixes for the inconsistency (i.e., the possible transformations that may be executed to change the current model version to one within the allowed solution space) are depicted with solid arrows from $B3$ to $B4$, $B5$, and $B6$. If the restrictions are unambiguous, only a single option remains and it may be applied automatically (much like transformation). For example, the action `<add`

`method "dim" to class "LightSwitch">` is an option for removing the inconsistency caused by the absence of the method `dim` in the `LightSwitch` class and the constraint $c4$. Thus, using constraints does not only expose inconsistencies but it also enables user guidance to help understanding and solving them [58]. However, models may contain a large number of inconsistencies that makes manually fixing them one-by-one – even with guidance – a challenge [14]. Hence, the use of fully automated approaches may be preferable in such cases. Thus, deciding which fixing strategy to use (i.e., guided manual fixing or automated fixing) strongly depends on the specific situation (i.e., the present inconsistencies). Let us now discuss how both strategies support our approach and for which typical scenarios they are most suitable.

### *2.3.1 Guided Fixing*

In our running example we used transformations for generating the constraints from specific model elements such as messages. We believe that incorporating source model data makes a constraint much more specific and expressive when presented to the designer than a manually written, generic constraint that relies on metamodel data and functions (e.g., the constraints use actual method names).

We discussed above that structural dependencies between constraints do not occur and that their validation is independent. However, *logical dependencies* between constraints in terms of required model characteristics and corresponding model elements may occur (e.g., $c1$ requires a class `LightSwitch` and $c2 - c3$ require specific methods in this class). Without a `LightSwitch` class, the model cannot be consistent. However, the constraints $c2 - c3$ are consistent if there is no such class at all or if the specific class provides the methods. This means that choosing an option that removes the inconsistency of $c1$ (i.e., the addition of a new and empty class named `LightSwitch`) leads to the constraints $c2 - c3$ being inconsistent until they are addressed by adding the required methods to the new class. Creating additional inconsistencies can therefore be necessary to achieve overall model consistency. Recent research has shown that such problem can be solved, for example using search-based transformation in combination with constraints [59]. Other recent studies suggest that such dependencies can be leveraged to decrease the number of possible actions to fix inconsistencies dramatically [60].

However, even without considering logical dependencies between constraints it has been shown that the average number of possible fixes per inconsistency for common UML well-formedness constraints in industrial models stays below 10 [42]. Thus, a newly introduced inconsistency can be easily handled through guided fixing. Moreover, the time for calculating possible fixes typically remains under 0.1 seconds with common approaches such as [61] or [42]. Therefore, chaining the incremental constraint generation and management of CDM, which, as we will show in Section 2.5, updates the constraint model and provides consistency feedback within milliseconds, with existing semi-automatic fixing approaches produces a modeling environment in which designers are informed about both inconsistencies and possible options for resolving them during modeling. Note that a variety of such approaches is available for use with CDM. For example, there are search-based approaches that find possible fixes by exploring the problem's solution space (e.g., [14] or [61]). Moreover, there are also incremental techniques that compute possible fixes directly based on specific inconsistencies (e.g., [42]).

Guidance is however not limited to inconsistencies. For each constraint, its source as well as the locations where it is validated are available and can be presented to the designer by a modeling tool. When the source model is edited during development, the constraints that are affected by those changes can also be highlighted. When a designer, for example, adds a new message to a sequence diagram with a name that already has a matching method in a class diagram, a modeling tool may highlight the constraint and show him or her the existing method. The designer can then easily decide whether this existing method should be used (i.e., the message means the existing method) or if a naming conflict was introduced (i.e., a new method was planned).

### *2.3.2 Automated Fixing*

Although guided fixing is suitable for removing small numbers of inconsistencies, there may be situations in which larger numbers of inconsistencies occur that are not manageable through fixing inconsistencies one-by-one. As we have briefly mentioned above, constraints may come from diverse sources. For example, they may be derived from a metamodel and may be used for validating the structural integrity of a specific metamodel instance (i.e., model) in a flexible modeling tool (e.g., [62, 63, 64]). In such a case, a generic constraint derived from the metamodel could, for example, check that all modeled classes provide exactly one name. Another constraint may check that no class has more than one parent class. Note that such generic constraints have to be enforced for every single class present in the restricted model. Indeed, if a new concept is then introduced in the metamodel (e.g., a new single-valued attribute for classes called "description"), this would result in the generation of a new generic constraints that required all classes to provide such a field. For every single class in the restricted model, an individual inconsistency is detected. Indeed, the number of inconsistencies that occurs due to such a change is often not manageable manually.

There has been significant progression in the field of automated model healing and fixing of inconsistencies. For instance, Anastasakis et al. [55] presented an approach for the transformation of UML models and OCL constraints to *Alloy* [65] that allows for sophisticated reasoning over UML models. After using Alloy for analyzing the model and extending it automatically to establish consistency, the extended model can be transformed from Alloy back to UML using the approach presented by Shah et al. [56]. Kuhlmann and Gogolla [57] use a similar approach for transforming UML models and OCL constraints to relational logic reasoning, based on *SAT* [66], with *Kodkod* [67] and transforming the resulting model back to UML.

A different approach for fixing inconsistencies was presented by Xiong et al. [41]. They developed a language called *Beanbag* that allows the definition of constraints and fixing behavior at the same time. When a constraint becomes inconsistent, its associated fixing behavior is used to derive a set of model changes that are executed to restore consistency. The Beanbag language is quite similar to standard OCL but provides several additional language constructs that allow constraint authors to also specifiy how issues may be solved. Our approach can be used to generate such Beanbag programs instead of pure OCL constraints. Note that with our approach, the fixing-related parts may also be generated using specific source model data, potentially allowing for more precise or more efficient calculation of fixes.

In principle, it is thus possible to chain CDM and the discussed approaches to automatically keep target models consistent at all times.

### *2.3.3 User-centric Approach*

As we have discussed, CDM may be extended by both guided manual fixing and automated fixing in principle. However, usually one approach is more favorable than the other depending on the various aspects (e.g., the number of inconsistencies, kinds of inconsistent constraints). The decision which approach is more suitable is non-trivial and may also depend on the designer's abilities. Therefore, we believe that it should be the designer to decide whether an inconsistency (or multiple inconsistencies) should be fixed manually or automatically.

As discussed in Section 2.3.1, changes in the source model will often lead to a small number of new inconsistencies that can be fixed manually because of typically small numbers of available options. However, as discussed in Section 2.3.2, there may also be scenarios in which multiple inconsistencies – which are, however, based on a single kind of constraint – are introduced by a single source model modification. In that scenario, the use of fully automated approaches will often be the best solution. Note that using an automatic approach based on sophisticated reasoning is always an option for getting a valid model quickly.

## 2.4 Additional Benefits of Constraint-driven Modeling

Above, we described how CDM addresses uncertainty in transformations, an issue on which we also focused in the case study we used for evaluation (see below). Next we discuss how the approach can be applied in additional scenarios to address issues related to model-synchronization and bidirectionality.

### *2.4.1 Rule-scheduling and race conditions.*

Now let us consider an example where two transformation rules $t_{m1}$ and $t_{m2}$ are working with the same generated model and the order of rule execution is important. For example, the sequence diagram in Fig. 2.3(a) contains an instance of the class `LightSwitch`. Therefore, let us assume that transformation rule $t_{m1}$ generates a corresponding class if no such class exists in the diagram in Fig. 2.3(b). As we have discussed in Section 2.1, the sequence diagram requires the class `LightSwitch` to provide a method `activate`. Let transformation rule $t_{m2}$ generate this method in `LightSwitch`.[6] When the transformations are performed, it is crucial that $t_{m1}$ is executed before $t_{m2}$ to ensure that the class `LightSwitch` exists before the method `activate` is added. This issue is illustrated in Fig. 2.12(a) where the bottom transformation encounters an error after the execution of $t_{m2}$. If the rule $t_{m1}$ is still executed, the resulting model $B$ will contain an empty `LightSwitch` class because

[6] We ignore the fact that such a transformation will not always lead to satisfying results – as discussed above – for this example.

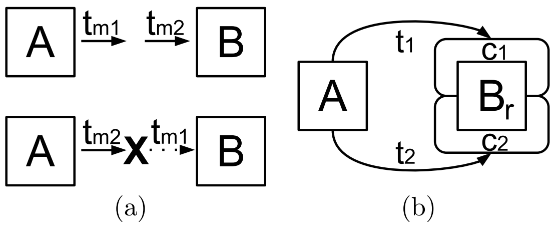


FIGURE 2.12: From dependent (a) to independent (b) transformation rules

only $t_{m1}$ was executed successfully. If the execution of rules is stopped after the error, no model is generated at all. Defining the order of rule execution manually is tedious and a constant source of error. Moreover, support for defining an execution order is not a standard feature of all transformation languages or systems [19].

The constraining approach, shown in Fig. 2.12(b), is free of scheduling issues because constraints cannot structurally depend on other constraints and the order of transformation is not relevant for the transformation results, as discussed in Section 2.2.4. Hence, the rules $t_1$ and $t_2$ we have previously defined can be applied in any order and produce the distinct constraints $c1$ and $c2$ in Fig. 2.12(b). If a model does not provide the required information for constraint validation (e.g., the class that should be checked is not present), the validation fails and an inconsistency is detected.

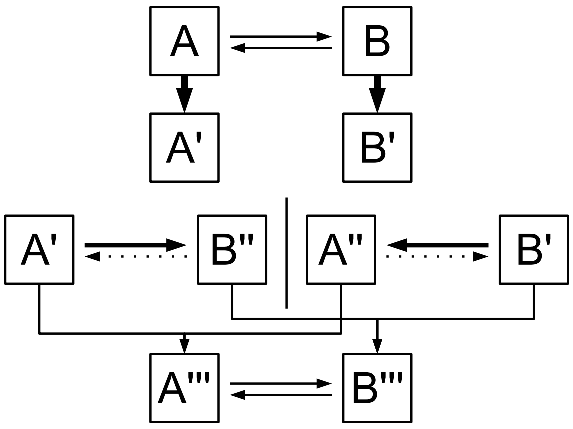
### *2.4.2 Bidirectionality and model merging.*

When models should be synchronized automatically, transformations are often used to propagate changes from one model to the other and perform the corresponding changes. Let us assume that we have established transformation rules that keeps message names and method names synchronized and that a link between messages and corresponding methods exists. In Fig. 2.7(a), the name of the highlighted message has been changed from `turnOff` (see Fig. 2.3(a)) to `deactivate`. Concurrently, the corresponding method in the class diagram was changed from `turnOff()` (see

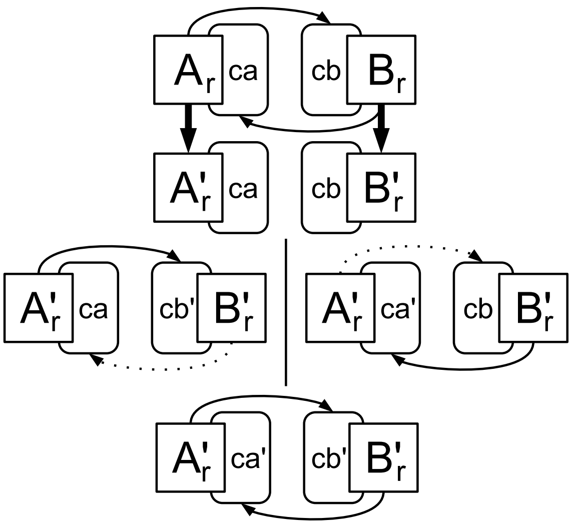
Fig. 2.3(b)) to `switchOff()`, as highlighted in Fig. 2.7(b). Since both synchronized model elements were changed (indicated by the bold arrows), there is no way to determine in which direction the required synchronization should be performed. Performing a synchronization in this situation will lead to the loss of the changes in the generated model (i.e., either $B''$ overrides changes in $B'$ or $A''$ overrides changes in $A'$ that cannot be used for a transformation in the opposite direction afterwards). A possible solution would be the concurrent execution of the transformations followed by a merge of the updated models ($A'$ and $B'$) and the resulting generated models ($A''$ and $B''$), as illustrated in Fig. 2.13(a), that generated $A'''$ and $B'''$. However, this requires a complex merging strategy and is likely to produce models that still require manual adaptation.

The solution of the constraint transformation approach is shown in Fig. 2.13(b). We can see that our approach still has to decide which change to process first. However, because only constraint models are updated, the restricted models $A'_r$ and $B'_r$ are not changed and can therefore still be processed to perform constraint updates in the opposite direction, leading both constraint models *ca* and *cb* being updated. With our approach, no immediate merging (either automated or manual) is required when restricted models are edited and following source model changes lead to constraint updates.

After the constraint model updating took place in the example, there are two new constraints: i) message number 2 in Fig. 2.7(a) should be named `switchOff` (from Fig. 2.7(b)) and ii) the name of the method `switchOff` in Fig. 2.7(b) should be changed to `deactivate` (from Fig. 2.7(a)). The designer can then decide which of the elements should be renamed.



(a)



(b)

FIGURE 2.13: From bidirectionality (a) to unidirectional (b) constraint transformation

## 2.5 Case Study – UML Model Constraints

To assess the feasibility and the applicability of CDM, we conducted a case study with two goals: i) demonstrate the applicability of CDM to typical MDE scenarios, and ii) assess the efficiency and performance of our constraint-management approach. For the former goal, the case study relies on large-scale industrial-grade models in the

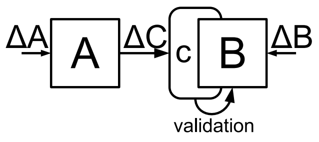


FIGURE 2.14: Possible changes in CDM

domains of object-oriented software modeling – CDM was used to generate various kinds of domain-specific constraints. For the latter, we performed various source model changes to trigger changes in the set of enforced constraints (e.g., constraint addition). In Fig. 2.14, the different sources of change (Δ) in our approach are depicted. Note again that the application of CDM consists of two core phases: i) constraint management, and ii) constraint validation. The first phase is triggered by changes of the source model (ΔA) and includes an update of the applied constraints (ΔC). The second phase – validation – is triggered by changes of either the constraints (ΔC) or the restricted model (ΔB).

In principle, we need to demonstrate performance for both phases – including the validation triggered by changes of the restricted model (ΔB). However, the performance assessment for model changes (ΔB) can be omitted for two reasons. First, this work uses an existing consistency checker that was already thoroughly evaluated on 34 large-scale industrial models of up to 162,237 model elements and complex constraints in [44, 48]. It was shown that most changes to a restricted model are processed in less than one millisecond and that the validation is faster than typical batch-validation performed by conventional consistency checkers. Second, this thesis does not prescribe any consistency checker in particular. In principle, any technology may be used. The one used in this chapter is merely a proof that current state-of-the-art is sufficiently progressed to support the second phase. In our performance tests we captured the time needed for handling a source model change and generate, remove, or update constraints (i.e., processing of ΔA and ΔC) as well as the time for constraint validation triggered by the resulting changes of constraints (ΔC). In our analysis we provide values for both constraint management only and constraint management with constraint validation. The observed times for constraints constraint were compared with the previously observed validation times for manually written constraints to assess how using generated constraints affects the validation performance of existing consistency checking technologies.

### *2.5.1 Prototype Implementation*

To perform the case study, we implemented a prototype tool. The core component of the tool is a custom implementation of an incremental constraint transformation engine that supports arbitrary EMF-based source models and generates OCL constraints. The transformation engine builds the trace information required for incremental constraint model updates automatically. For efficient constraint validation, we employ the *Model/Analyzer* [43] consistency checking framework which we slightly adapted to support EMF-based models in general instead of only UML-models. Note that the constraint validation may be done by any consistency checker that supports EMF models and OCL constraints. However, we chose the Model/Analyzer because it was evaluated with design models that were used in the case study. Thus, observed validation times may be compared to previously observed data. We also implemented operation-based change trackers that are used to observe models and inform the transformation engine or the consistency checker, respectively, about changes, as we have discussed in Section 2.2. The transformation engine is implemented to support arbitrary numbers of source and restricted models concurrently.

### *2.5.2 Scenarios and Design Models*

For this case study, we apply CDM to typical object-oriented software models in and generate constraints from different diagrams, as we have already illustrated in Section 2.1 and Section 2.2. In particular, we use individual parts (i.e., diagrams) of large-scale UML models from which we generate constraints that restricts other parts of the same model.

We employed 22 of the industrial models that were previously used in [44, 48]. Note that those 22 models were selected because they include different types of diagrams from which constraints that restrict other diagrams can be generated (i.e., we did not use design models that include diagrams of only one type).

The models include class diagrams, sequence diagrams, state charts, use-case diagrams, and various other UML diagrams. The model sizes varied between 337 and 27,751 top-level model elements. That is, instances of primitive data types, for example, were not counted.

### *2.5.3 Transformation Rules*

We used the transformation rule $t2$ shown in Listing 2.2 to generate constraints from `Message` instances in sequence diagrams, as discussed in Section 2.2.

### *2.5.4 Performance Evaluation*

For assessing change processing performance, we used three different tests:

**Test I** Replacing ambiguous transformations – as discussed in Section 2.2

**Test II** Replacing merges – different sources restrict a single model

**Test III** Restricting multiple models – one source restricts different models

Test I simulated the replacement of traditional transformations in the scenario involving uncertainty we described in Section 2.2. Note that UML models usually consist of one single model that includes all data and that different diagrams only represent different views on that data. Therefore, we use the same model as source and restricted model in Test I (this being another benefit of our approach). The test was executed for all available models that included elements matching the transformation rule context.

Test II assessed the performance of our approach in scenarios where multiple source models are used to generate various constraints that are restricting the same model (i.e., merges of generated models would be required with traditional approaches). This test shows the behavior of the approach when complexity is increased and more models become involved. As in Test I, we use one model as source and restricted model at the same time. Other models are used as additional source models from which additional constraints on the restricted model are generated. The same models as in Test I were used as source and restricted models, the selection of additional source models was done randomly. The test was done with five and ten concurrent source models.

Please note that in Test I and Test II our approach was applied to replace traditional model transformations and model merges. However, CDM follows a different principle than the replaced approaches as it generally emphasizes the importance of user interaction. The validation therefore focuses on assessing whether the performance of CDM is acceptable for usage in modeling tools without interrupting tool users. Thus, we do not discuss in detail the replaced approaches and it is not necessary to compare the performance of CDM to the performance of the approaches it replaced.

Test III simulated scenarios where a single model is the source for transformations that produce constraints for different, concurrently active restricted models. Again, the same models as in Test I were used as source models. The restricted models were selected randomly, groups of five and ten concurrent restricted models were used. This test was performed to assess the scalability and the efficiency of our approach when the numbers of generated constraints and required constraint validations as well as the required infrastructure's complexity increase.

*Source Model Changes.*

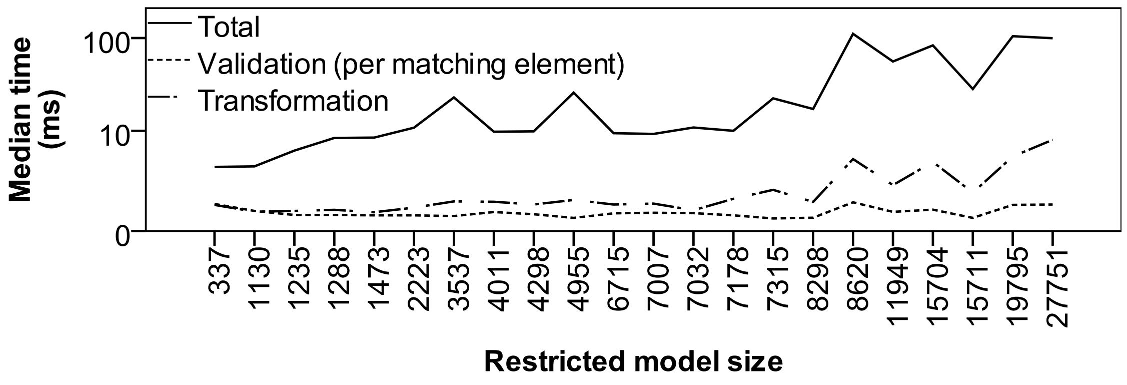
For all tests, randomly selected single model elements were removed from a source model and then added back to the source model, which forced an incremental constraint model update. That is, the update of exactly one constraint for Test I and Test II and the update of at least one constraint for Test III. For Test II, the source model on which a change was performed was also chosen randomly for every performed change. Note that we only performed changes that updated constraints which were actually validated on at least one restricted model element. This ensures that all changes impact source models used during transformation. Note also that complex model changes or refactorings, such as the removal of a whole package, can be expressed as a sequence of atomic changes. The processing time for complex changes can therefore be estimated using the number of required atomic changes and the observed times required per such atomic change.

All the tests were run on an Intel Core i5-650 machine with 8GB of memory running Windows 7 Professional. Each model change was executed 100 times and we used the median values for our analysis.

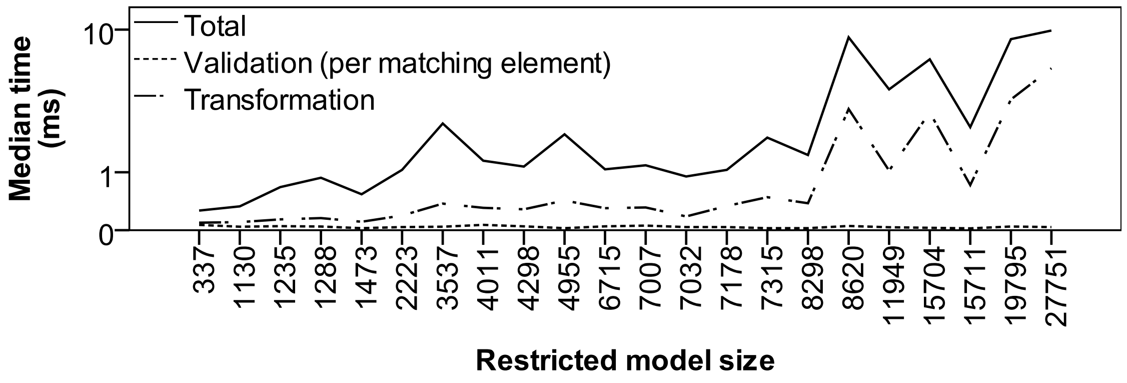
*Results.*

We now discuss the observations made for the individual tests.

*Test I:* In Fig. 2.15(a) the median times for adding a source model element are depicted. For the transformation time (i.e., the time required for constraint generation), the observed processing times are steady at about 1 ms and slightly increase for large models of over 27,000 model elements to about 10 ms. For the total processing time including both transformation and constraint validation, we observed a correlation with the model size. This is based on the fact that we generated generic constraints that were validated for all instances of a specific type. Therefore, the total time required for validation increases with the number of elements matching the



(a) Addition processing times



(b) Removal processing times

FIGURE 2.15: Results for Test I

constraint context. However, the dashed line in Fig. 2.15(a) shows that the validation time per constraint context matching element was steady at under 1 ms. These observations are consistent with those made in [44] and indicate that the constraints generated with CDM allow for the same quick validation as manually written constraints. Note that the total processing times even for large models reach a maximum of about 100 ms.

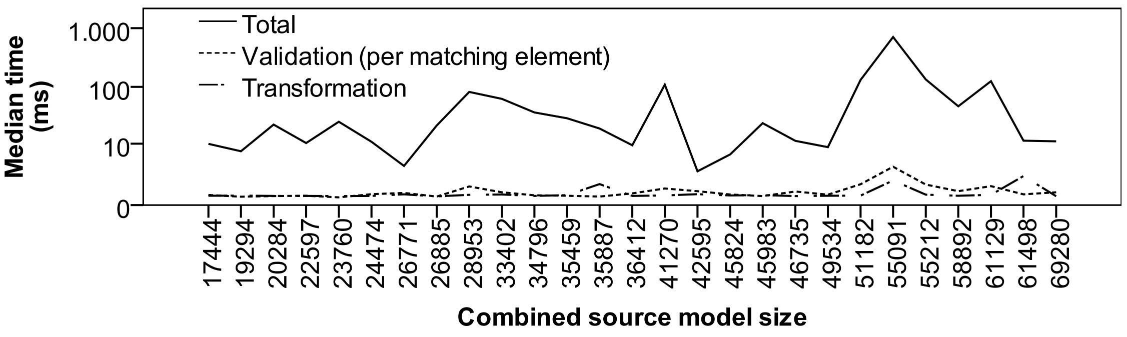
The median times for removing a source model element are depicted in Fig. 2.15(b). Note that the median total processing time has a maximum of about 10 ms even for the largest models. The time required for constraint validation is negligible because there is no validation required when removing constraints. We observed a correlation between the number of constraints generated from the source model and the time required for finding and removing the constraints based on the removed source model element in the test. This is caused by the implementation of the trace information in the transformation engine. Specifically, we believe that this is based on the use of hash tables and an increasing number of hash collisions occuring with increasing numbers of traced source model elements. The spikes in the graph for models between 8,000 and 20,000 model elements support this assumption and indicate that the model characteristics have a stronger effect than the model size.

*Test II:* In Fig. 2.16(a), the processing times for the addition of an element to one out of five concurrent source models are shown. Note that there is no correlation between the total source model size and the median processing times.

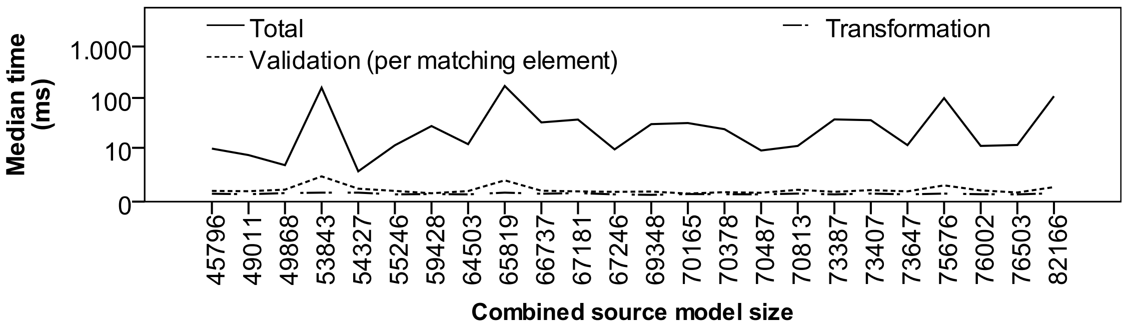
In Fig. 2.16(b) the processing times for ten concurrent source models are depicted. Again, there is no correlation between the total source model size and the processing time. Additionally, note that the processing times for both five and ten source models are typically within 10 ms and 100 ms – similar to the results of Test I in Fig. 2.15(a) – indicating that the additional infrastructure required for handling concurrent source models does not lead to reduced efficiency.

In Fig. 2.16(c) the median required processing time for the addition of a source model element as a function of the number of restricted model elements matching the affected constraint's context (i.e., the number of constraint validations triggered by the change) are depicted. As for Test I, both the validation time per constraint matching element and the transformation time are steady at about 1 ms and reach a maximum for large models of 5 ms. The total processing time of course increases with the number of required validations. For the removal of a source model element, both the processing times for validations per matching element and for the transformation remain at under 1 ms and the maximum total processing time stays under 100 ms (figures are omitted).

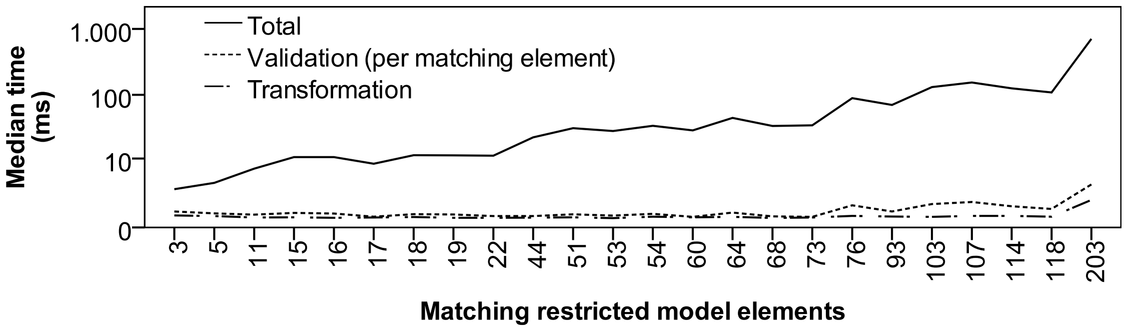
*Test III:* The observed processing times for the addition of an element to the source



(a) Addition processing times (5 source models)



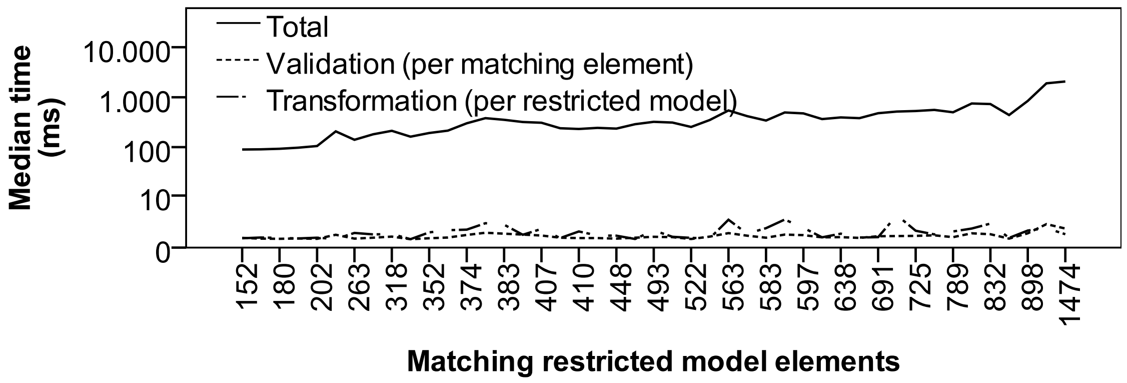
(b) Addition processing times (10 source models)



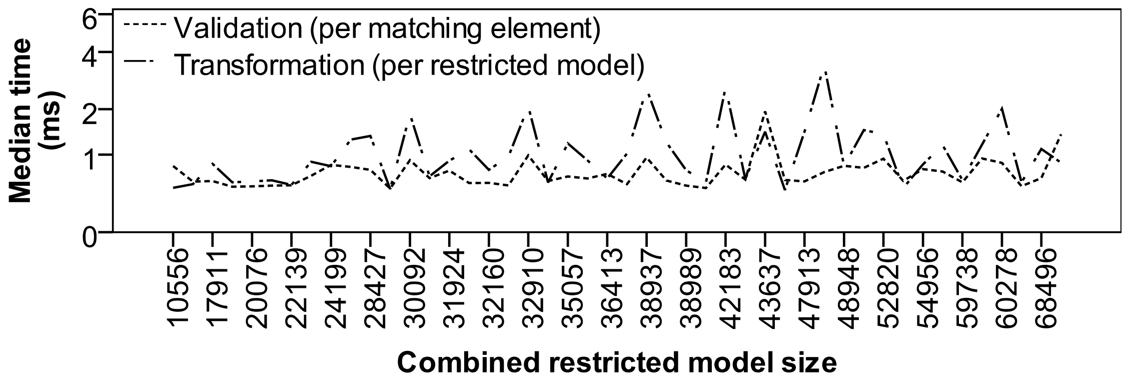
(c) Addition processing times for different effects on restricted model

FIGURE 2.16: Results for Test II

model are depicted in Fig. 2.17(a). The results for tests with both five and ten concurrent restricted models are combined and the times are drawn as a function of the number of constraint validations that become necessary after the change. Note that the times for both transformation per restricted model and constraint validation per matching element are nearly static at about 1 ms. As in Test II, the total processing time is determined primarily by the number of required constraint



(a) Addition processing times for different change effects



(b) Addition processing times for different restricted model sizes

FIGURE 2.17: Results for Test III

validations.

In Fig. 2.17(b), the processing times for constraint validation and transformation are depicted as a function of the combined restricted model sizes.

For the removal of source model elements, the results observed for Test III are similar to the results observed in Test I and Test II. That is, processing of source model element removal is significantly faster that element addition as there is no need for generating and validating constraints (figures are omitted).

## 2.6 Discussion

In this section we discuss the results of the presented case studies: the scalability, the correctness, and possible threats to validity.

### *2.6.1 Key Challenges Addressed*

Let us briefly revisit the three key challenges we identified at the beginning of this chapter and discuss how they are addressed in CDM by providing information to designers instead of automating highly complex and situation-specific decisions.

As we have shown, CDM does not override model changes done by designers but it updates constraints. In doing so, CDM ensures that decisions made by designers are never lost.

In case of uncertainties, we have illustrated how CDM avoids premature design decisions by generating and updating constraints to provide valuable information for finding the most suitable target model instead of generating a correct but probably not ideal target model.

By using CDM, bidirectional transformations can be replaced with unidirectional transformations. This reduces scheduling issues, race-conditions, and merging issues when handling concurrent model changes because the restricted models are not changed automatically and thus also the order of constraint updates is not relevant.

### *2.6.2 Case Study Results*

We demonstrated the feasibility of CDM by implementing a prototype tool. The applicability was demonstrated by applying the approach to UML modeling projects. The case study evaluated a range of small to large systems.

The observed transformation times remained nearly constant even when model sizes increased significantly, which indicates that the core aspect of our approach can be performed efficiently. The observed overall processing times (including constraint validation) show that even for large models our approach works efficiently. The results demonstrate in a proof-of-concept manner that there exists at least one consistency technology that supports CDM in a scalable manner.[7]

[7] Note that the validation is not meant to demonstrate superior performance of the employed

The case study was conducted with the same models that were previously used for evaluating the employed consistency checker. The observed constraint validation times indicate that times required for validating generated constraints are comparable to those required for validating manually written constraints. Thus, generated constraints do not necessarily impose a higher validation effort.

### *2.6.3 Correctness Aspects*

Let us now discuss various aspects that may affect the correctness of our approach. For traditional model transformations, errors in both the source models and the applied rules lead to errors in the generated model. Such errors obviously affect CDM also because through the likely involvement of humans during source model creation and transformation rule writing, it is not possible to guarantee correct constraints being generated and enforced. The correctness of the enforced constraints and the provided user guidance is thus a factor of the correctness of: i) the source model, ii) the transformation rules, iii) the transformation engine, and iv) the consistency checker.

Even if invalid source models or transformation rules lead to incorrect constraints, our approach has substantial benefits over generating a model directly: incorrect results do not affect the restricted model directly.

Designers may inspect constraints that seem incorrect and may decide to ignore them, meaning that incorrect or contradictory constraints do not prevent designers from constructing the desired model. By tracing back the origin of generated constraints (which is possible in model transformation and supported by CDM) designers can also use faulty constraints to detect and report or fix errors in the source model or the transformation rules [42].

consistency checker but only to demonstrate feasibility of CDM.

### *2.6.4 Threats to Validity*

Although it seems intuitive that decisions made by domain experts in situation with very specific problems and with guidance are more trustworthy than automated decisions based on generalized knowledge or heuristics, we have yet to show that the quality of the resulting models is higher or that our approach leads to *intended* target models quicker than traditional transformation. Additionally, we have not investigated to which degree guidance and suggested options reduce the time needed for design decisions or finding inconsistencies. However, those questions are not specific for CDM but they apply to all approaches that focus on semi-automatic fixing of inconsistencies.

In Section 2.2 we illustrated the application of CDM and assumed that transformation rules are executed incrementally. Even though there is significant progress in terms of incremental execution of transformations for common languages such as ATL or QVT, there are transformation languages for which such support is not available. However, we have discussed in Section 2.2.4 that individual constraints are structurally independent. This simplifies the implementation of incremental rule execution dramatically as it allows tools to execute a transformation rule with a given source model element in a sandbox-like environment (i.e., execute the transformation rule with a temporary and empty target model) and then add the resulting constraint to the set of applied constraints. Note that the order of rule execution does not matter. Moreover, if transformation rules are executed individually then this also simplifies the backtracking later (i.e., if it is believed that a constraint is incorrect).

## 2.7 Related Work

Model Transformation is a very active field of research and several topics related to our work have been discussed.

**Comparison to Existing Constraint Generating Approaches.** Let us now discuss what separates the CDM approach from existing approaches that focus on model transformations and rely on constraints.

The major difference between CDM and approaches such as, for example, the ones presented by Büttner et al. [50] or Cabot et al. [51] is that the goals of these approaches are different from those of CDM. While CDM helps dealing with uncertainties and avoiding the generation of unintended target models, other approaches typically generate constraints in order to do verification or validation of transformation rules through sophisticated reasoning over those transformation rules and target models (e.g., derive an optimal order of execution for a set of transformation rules). They derive target model conditions from the transformation rules directly through a translation. Thus, they are typically not capable of dealing with domain-specific semantics and uncertainties like those we have discussed in Section 2.1. For the transformation in Listing 2.1, for example, those approaches may generate a constraint that requires a `Method` with a specific name and a specific owner to be present in the target model after the transformation was executed with a given `Message`. This constraint ensures that the transformation rule behaves as expected. The constraint may then be used in combination with constraints derived from other transformation rules, for example to find logical contradictions within the transformation rule, to find contradictions between rules, or to determine the required order of execution. Note that when those constraints are used as input for reasoning engines or fixing approaches, the constraints are already based on an assumption that was made during rule authoring to overcome an uncertainty (e.g., the desired location of the method) and that was incorporated in the transformation rule. With CDM, on the other hand, a constraint that a method must be *provided* by the class would be generated (i.e., a different transformation rule would be used), expressing the actually desired condition without making assumptions. Thus, existing approaches that generate constraints from rules are designed for different problems (e.g., ensuring syntactical correctness of target models or checking validity of transformation rules). However, those approaches are of course a valid choice for verifying and validating unambiguous transformations and may be used for verifying the transformations used for CDM (e.g., to find out whether the applied transformations lead to contradictory constraints).

Another approach related to CDM is proposed by Vallecillo et al. [68]. They define *Tracts*, contracts between source and target models which include constraints. Using those tracts, model transformation rules can be derived or existing transformation rules can be tested. While this approach is similar to CDM in that it focuses on the intentions of the designer and allows for domain specific knowledge, it does not address the issue of designers being forced to make premature decisions when writing transformation rules. For example, it would identify the transformation rule presented in Listing 2.1 as valid with respect to the target model constraints – the solution it produces will always be valid as the required method is present in the target model. However, the transformation rule still relies on assumptions and it may produce unintended target models.

Another benefit of our approach is that it is generic. It can be used with arbitrary models, transformation languages, constraint languages, and consistency checkers. Existing approaches interpret rules and generate constraints for a given transformation language and are also strongly tailored to a constraint language, a consistency checker, or a reasoning engine (e.g., [50, 51]).

Generally, CDM can be seen as a bridging technology between modeling and advanced approaches for model fixing that rely on the existence of constraints. What makes CDM stand out from other approaches is that it allows designers to express desired model conditions in a reusable way (i.e., as transformation rules) rather than inferring those conditions from transformation rules in which assumptions have already been made.

**Model transformation and OCL constraints.** Büttner et al. [69] discussed various endogenous transformations of OCL constraints that are necessary when either the constraint or the model it is based on change. We apply transformations to automatically generate such constraints from a source model and, instead of transforming the existing constraints, use incremental re-transformation to reflect model changes. Simplifications of OCL constraints through transformations was also discussed by Giese and Larsson [70]. They defined transformation rules that could be used with our approach to simplify the constraints that have been automatically created and possibly make them faster to check and also easier to read. In [71], Bajwa and Lee propose an approach to automatically generate OCL constraints by transforming business rules specified in Semantics of Business Vocabulary and Business Rules (SBVR) [72]. Our approach of course also supports business models as constraint source. However, we have shown that the generation of constraints is a valid option to reduce ambiguities and provide guidance in different domains.

**Bidirectional transformation.** Giese and Wagner [31] as well as Xiong et al. [73] performed extensive research on bidirectional transformations and synchronization in general. Interestingly, Xiong et al. define an "undo" operation after model updates (i.e., reverting performed changes) to be an unwanted option for fixing introduced inconsistencies because it would ignore the latest decisions made by the designer [41]. Indeed, for automated fixing without human intervention making such an assumption is necessary as otherwise a simple "undo" of the latest change would typically be the best solution that removes all inconsistencies without any side effects (i.e., introducing new inconsistencies through fixing existing ones). However, a designer's decisions can be wrong and therefore we believe that presenting the "undo"-option in addition to other possible fixes is necessary to address this possibility. Regarding language support for bidirectional transformations, Sasano et al. [74] developed a system to perform bidirectional transformations with ATL, and Stevens [26] focused on bidirectionality for QVT. In general, we tackle the complexity of bidirectional transformations by using unidirectional transformations that generate constraints – without tailoring our approach to a specific transformation language or specialized transformation engines, and without deriving constraints from target model generating transformation rules. Updating one of the involved models then changes its own consistency status and potentially lead to updates of constraints on the other model. The involved models are synchronized if they are both free of inconsistencies. As we have shown, the risk of overriding changes because of the change processing order is eliminated as we only update constraints automatically and the designer ultimately decides how the models should be adapted. Cicchetti et al. [75] developed the bidirectional transformation language *JTL* that supports non-bijective transformations and change propagation. JTL generates constraints by translating user-defined transformation rule and it uses *answer set programming (ASP)* to find models that match those constraints. The translation of transformation rules to ASP constraints can be interpreted as a transformation of those rules. Thus JTL could be seen as an application of CDM chained with automatic model finding where the user-defined transformation rules are used as source model to generate constraints through the execution of other transformation rules (i.e., rules that capture the translation semantics).

**Incrementality and execution speed of transformations.** Jouault and Tisi [32] proposed an approach to make ATL transformations incremental. They achieve incrementality by using scopes built during OCL expression execution to determine which rules have to be re-executed after source model changes. We make use of automatically created scopes in the same way to determine which constraints have to be re-created in our prototype and also for finding constraints that have to be re-validated by the consistency checker [43]. In [76], Tisi et al. propose the lazy execution of transformations, which eliminates the need for an initial transformation of the entire source model to speed up the process for large source models, which is also the performance bottleneck of our prototype.

**Design space effects of constraints.** Saxena and Karsai [77] published a MDE-based approach for design space exploration in which constraints are used to describe invariants of valid models. Our approach can be used to generate constraints for design space exploration algorithms. The source of these constraints may be the metamodel to which the generated model must conform or also an already existing partial model that is either provided by the designer as input for the exploration algorithm or generated by the algorithm itself. Our approach also reduces the solution space and provides guidance to transform an invalid solution to one that is within the remaining solution space. Horváth and Varró [78] presented an approach for design space exploration using a CSP-solver and *dynamic constraints* that may change over time. Combined with CDM to generate and manage these constraints automatically, their approach is suitable for finding out whether there actually are restricted models for which all constraints are satisfied. Moreover, their approach supports *flexible constraints*. That is, it can providing solutions even if contradictory constraints that cannot be satisfied at the same time are used.

**Finding domain design errors.** Queralt and Teniente [79] presented an approach for finding conceptual errors in UML schemas and OCL constraints (e.g., a schema from which some classes may never be instantiated without causing inconsistencies). They transform UML class diagrams and OCL constraints to logic formulas such that standard reasoning engines can be employed. The reasoning engine may then be used to find a sample instantiation of the schema which is consistent. In contrast to the automatic inconsistency fixing approaches discussed in Section 2.3.2, the generated solution is not a UML class diagram, but an object diagram which conforms to the class diagram (i.e., they use the schema defined with UML as the metamodel for the reasoning). As with the approaches discussed in Section 2.3.2, CDM can be used for generating and updating the constraints which are used for reasoning (i.e., it is an enabling technology).

## 2.8 Summary

In this chapter, we presented an incremental and generic approach that uses model transformation to automatically generate and update constraints.

Based on constraints that are structurally independent of each other, we showed that constraint validation does not require a fixed order of execution. We illustrated how traditional transformation approaches produce consistent, albeit unintended models that have to be fixed manually. By using constraints, designers are notified about existing inconsistencies and the importance of fixing them. We illustrated how generated constraints enable user guidance (i.e., by using them to find possible fixes) and encourage the use of domain-knowledge to solve specific modeling problems. Even though our approach reacts to source model changes, updates affected constraints, and validates those constraints immediately, it does not necessarily enforce any of the available fixes for inconsistencies automatically. This sacrifice of automation allows us to tolerate inconsistencies – at least temporarily. However, we also described how CDM can be used with automated inconsistency fixing approaches in certain situations where there are too many inconsistencies to fix them manually. And we discussed how model transformation issues like ambiguity, rule-scheduling, model merging, and bidirectionality can be addressed. In conclusion, we believe this work contributes a novel complement to existing state-of-the-art on model transformation.

We validated the approach by developing a prototype implementation and used it to conduct case studies in three different domains, demonstrating both the feasibility and the broad applicability of CDM. Performance tests showed that our approach is scalable and provides guidance for designers.

In the next chapter, we will investigate further issues, specificially in metamodeling, that can be addressed by using the principles of CDM.

# 3

# Co-evolution of Metamodels and Model Constraints

In *Model-Driven Development (MDD)* [1], metamodels play a key role as they reflect real-world domains and define the language of design models as well as the constraints these models must satisfy. Over the past years, a trend has emerged that calls for design tools with adaptable metamodels – to customize the tool to a particular discipline, domain, or even application under development. *Domain-specific Languages (DSLs)* [10, 80] provide a set of modeling concepts tailored to fit a specific real-world domain. Indeed, metamodels and DSLs must evolve continuously; for example, to reflect changes of the domain or to meet new business needs. Refactorings that improve a metamodel's structure and usability are also common [81]. Nowadays, a range of "flexible" design tools with adaptable metamodels are available to support such scenarios (e.g., [62, 63, 82]).

*Co-evolution* of models denotes the process of concurrently evolving metamodels and their design models – a process that is non trivial since inconsistent co-evolution may cause models and metamodels to drift apart. Several incremental approaches have been proposed to support this process (e.g., [83]).

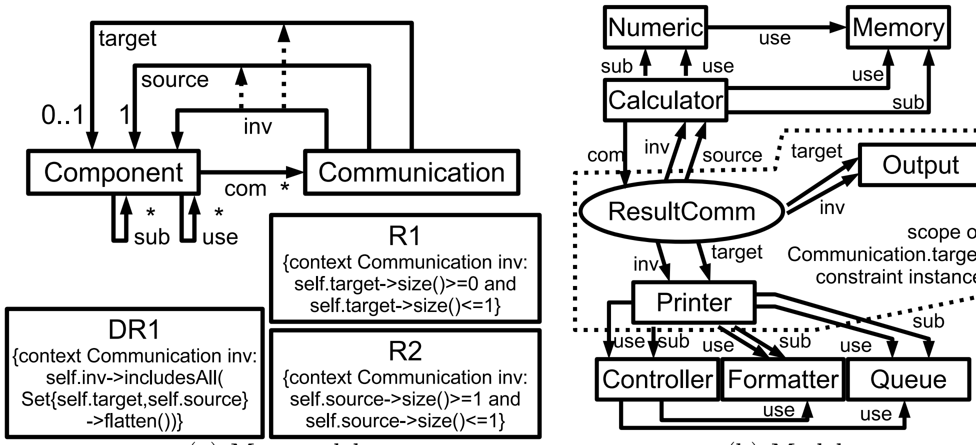
However, metamodels also impose constraints onto design models. When the metamodels evolve, so must the constraints – a scenario that has been largely overlooked so far. For example, the *Unified Modeling Language (UML)* [45] is supported by hundreds of well-formedness rules and the community augmented these with even more consistency rules. Moreover, *UML Profiles*, which may also include consistency rules, are commonly used to extend the UML and adapt it to specific domains [84]. Modifying the UML metamodel thus impacts these constraints. Previously semantically and syntactically correct constraints may become incorrect after structural or semantic metamodel changes; or new constraints may appear. It is crucial to extend the notion of co-evolution to include the continuous maintenance of constraints such that only correct constraints are enforced on design models. Of course, it is also crucial to have available a consistency checker that is not only able to react to design model changes but also to metamodel/constraint changes. Generating and adapting constraints incrementally as well as checking them incrementally are thus pre-requisites to ensure that designers are always given instant and reliable feedback on the validity of their modeling work.

State-of-the-art consistency checkers are commonly employed to validate constraints and determine whether a model is consistent with respect to its metamodel. Most consistency checkers rely on a fixed set of constraints for performing the validation [29, 43]. It is common to write these constraints manually, typically in a standardized language such as the *Object Constraint Language (OCL)* [39]. Often, constraints are also "hard-coded" into modeling tools. Although the automatic co-evolution of metamodels and design models has become an active field of research, the issue of co-evolving constraints is not well addressed. Incremental consistency checkers typically do not support the live updating of constraints and little support for updating outdated constraints is available.

This chapter, which is based on [85] and [17], describes an approach that handles the evolution of metamodels and performs the co-evolution of constraints. The approach in principle supports arbitrary metamodels and models. It uses constraint templates and a template engine to automatically and incrementally manage constraints based on atomic metamodel evolution steps (i.e., addition, deletion, and modification of metamodel elements). This chapter also includes the in-depth illustration and discussion of the approach and a prototype implementation (the *Cross-Layer Modeler (XLM)* [64]). Moreover, we evaluated our approach by using the XLM for automating the generation of constraints that ensure the structural integrity of UML models and by performing sample evolutions of the UML metamodel. Tests were performed on 21 large industrial UML models of up to 36,205 model elements. While the UML is not the primary motivation for our approach (it changes occasionally only), it is like any modeling language in that it must adhere to a metamodel and imposes constraints. UML metamodel changes thus impose the same kind of challenges. The fact that the UML language is far from trivial and we have available large-scale, industrial models thus make it a very suitable environment to test the scalability of our approach and the XLM tool. The results show that our approach is correct and works efficiently even as model sizes increase.

## 3.1 Motivating Example – Component-based System

We use an excerpt of a simple metamodel, shown in Fig. 3.1(a), to illustrate our work. The metamodel consists of two elements: `Component` and `Communication`. Every `Component` can include an arbitrary number of `sub`-components and can directly `use` an undefined number of other components. A `Communication` expresses a data exchange from a `source` to at most one `target` component. Components can have an arbitrary number of open communications (`com`).



(a) Metamodel (b) Model

FIGURE 3.1: Metamodel and model of component-based system with constraints

### *3.1.1 Metametamodel*

For building this metamodel, we used a simple metametamodel consisting of the elements: `Class`, `Reference`, `DerivedReference`. References between classes are drawn as arrows with an assigned name and a defined cardinality. Multiple references can be combined to a single *derived reference* which we draw without cardinality values and with dashed arrows to the references from which it is composed. For example, a derived reference is used to retrieve the components that are involved in a communication (`inv`).

### *3.1.2 Constraints*

For MDD to be effective, it is crucial to work with valid models that conform to their metamodels. That is, that such models adhere to the constraints specified in the following sources:

**I: Metamodel syntax.** First, we use intuitive constraints that check the cardinality of references. For each reference, we create a constraint (e.g., $R1$ or $R2$ in Fig. 3.1(a)) that ensures that every instance of the owning element is connected to the specified number of elements in a model (e.g., every instance of `Communication` must be connected to exactly one `Component` instance through a connection named `source`). We use the term *connected* in models to avoid ambiguity with *references* in the metamodel. Connections are depicted as named arrows in model diagrams. Constraints for references with unrestricted cardinalities (e.g., `com`) are not shown in Fig. 3.1(a) for readability reasons. Note that common modeling tools (e.g., [12, 62]) that use the *Eclipse Modeling Framework (EMF)* [86] for example either do not derive such constraints or have them "hard-coded", meaning that changes cannot lead to constraint updates which effectively disables automated co-evolution.

**II: Metamodel semantics.** Next, we create a constraint for the derived reference (e.g., $DR1$ in Fig. 3.1(a)) to ensure that instances of the owning element are connected to all the elements that are reached through the aggregated references (e.g., for every instance of `Communication`, all elements that are connected to it via `source` and `target` must also be connected via `inv`). Note that our constraints make use of OCL collection iterations even though they are invoked on single objects. The issues arising because of the distinction between single and multi-object values in OCL have been discussed and identified in literature as a problem especially during evolution [69]. For the sake of generality, we use a consistency checker with an OCL interpreter that allows collection operations being used with single objects by performing the necessary conversions automatically.

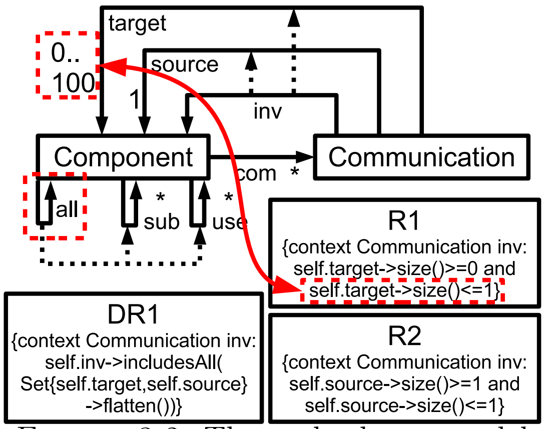
**III: Domain knowledge.** While the first two kinds of constraints could be generated automatically, constraints of the third type cannot be derived from the metamodel automatically with traditional approaches. An example would be a constraint that restricts direct usage of components based on component hierarchies.

As depicted in Fig. 3.1(b), the metamodel from Fig. 3.1(a) is used to create a small model of a calculator system. The `Calculator` component has two sub-components that are used directly: `Memory` and `Numeric`. The `Numeric` component also uses the component `Memory`. A `Printer` has three sub-components: `Formatter`, `Queue`, and `Controller`. It uses the `Queue` to store print jobs and informs the `Controller`, which retrieves data from the `Queue` and runs the `Formatter` before printing. Finally, there is an `Output` component to display information to the user. The `Calculator` uses a `Communication` element called `ResultComm` to send its results to the `Printer` and the `Output` components.

As indicated by the encircled area in Fig. 3.1(b), the two `target` connections of `ResultComm` are causing an inconsistency because only one `target` is allowed according to the metamodel. Note that any consistency checking approach could detect inconsistencies in the model according to the constraints we defined above.

### *3.1.3 Incremental Consistency Checking*

As the model size increases, so does the effort to check its consistency. Checking consistency in an entire model can easily become a time consuming task. Incremental consistency checking addresses this limitation by looking only at a subset of an entire model, namely the elements that change as a model evolves [87]. This set of elements can be either directly observed or calculated from differences between model versions [88, 43]. The existing approach automatically defines *constraint instances* that validate whether specific model elements violate a given constraint [87]. The change impact *scope* of a constraint instance is the set of model elements that are used for calculating the constraint instance's validation result which are also computed automatically. For example, Fig. 3.1(b) shows a *constraint instance* of the `Communication` metaclass constraint $R1$ that requires communications to have exactly one `target`. The scope of this constraint instance consists of the two elements that are reached through the `target` reference to `Printer` and `Output`. Changes falling within scope of a constraint instance, like removing a `target`, would lead to a re-validation of the constraint instance. The Model/Analyzer automatically creates, re-evaluates, and destroys constraint instances according to changes in the model in



target
0..
100
source
1
inv
Component
com *
Communication
all
*
sub
*
use
R1
{context Communication inv:
self.target->size()>=0 and
self.target->size()<=1}
DR1
{context Communication inv:
self.inv->includesAll(
Set{self.target,self.source}
->flatten())}
R2
{context Communication inv:
self.source->size()>=1 and
self.source->size()<=1}

FIGURE 3.2: The evolved metamodel

Fig. 3.1(b). However, if the metamodel were to change, consistency checkers would continue to validate the now-potentially-outdated design rules.

### *3.1.4 Co-Evolution Examples*

Let us consider what happens when a metamodel changes. For instance, if the number of maximum targets of a `Communication` rises from `1` to `100` because new technologies allow multicasting of messages between components. Additionally, a new derived reference `all` is introduced to combine the `sub` and `use` references of a `Component`. These two changes are encircled with dashed lines in Fig. 3.2. These changes have the following consequences:

- Constraint $R1$ becomes incorrect. The upper bound checked by $R1$ (`1`), is no longer equal to the actual upper bound value of the reference (`100`).
- An additional constraint is needed for the new derived reference `all`.

In the first case, $R1$ must be adapted by replacing the upper limit value `1` with literal `100`. Without this adaptation, the corresponding constraint instance, circled in Fig. 3.1(b), would still incorrectly try to enforce an upper bound of `1`. In the second case, the inconsistency that neither `Calculator` nor `Printer` have the required

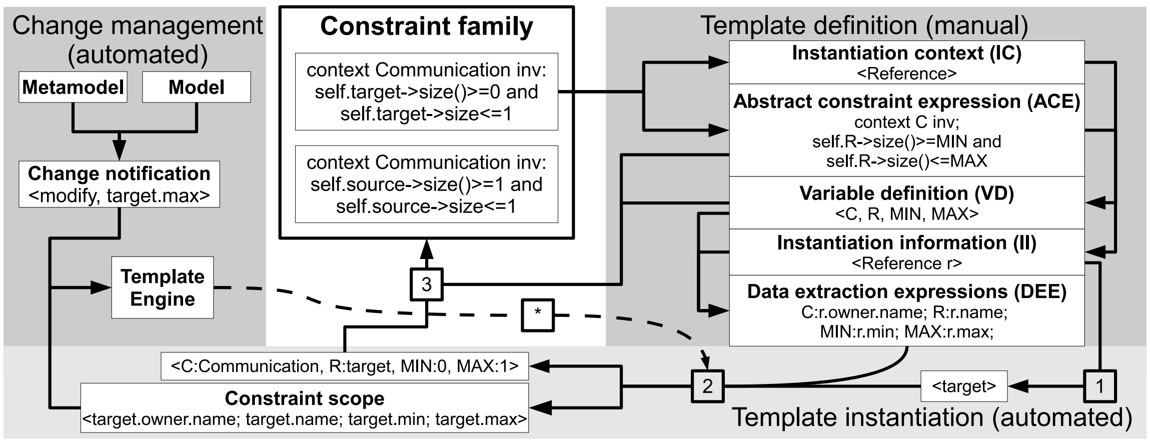

Change management (automated)
Metamodel
Model
Change notification <modify, target.max>
Template Engine
Constraint family
context Communication inv: self.target->size()>=0 and self.target->size<=1
context Communication inv: self.source->size()>=1 and self.source->size<=1
Template definition (manual)
Instantiation context (IC) <Reference>
Abstract constraint expression (ACE) context C inv; self.R->size()>=MIN and self.R->size()<=MAX
Variable definition (VD) <C, R, MIN, MAX>
Instantiation information (II) <Reference r>
Data extraction expressions (DEE) C:r.owner.name; R:r.name; MIN:r.min; MAX:r.max;
3
*
2
1
<target>
<C:Communication, R:target, MIN:0, MAX:1>
Constraint scope <target.owner.name; target.name; target.min; target.max>
Template instantiation (automated)

FIGURE 3.3: Example of steps performed during template definition, instantiation, and change management

connection `all` in our model is missed. To address this problem, a constraint that checks the derived reference `all` needs to be added.

A common way of dealing with co-evolution is to manually re-write the constraints after performing a metamodel modification. Although this approach can work in our example because of its small size and simple constraints, manually identifying and adapting affected constraints in more complex models is both time consuming and error-prone.

## 3.2 Constraint Templates and Template Engine

We propose the use of constraint templates to automate the co-evolution of models and their constraints. These templates are based on the metamodel and constraints we want to evaluate. Basically, templates contain the static aspects that constraints have in common (e.g., fragments of an OCL constraint string) and define the points of variability. As models evolve, the templates are filled with specific data – to reflect the model evolution – and instantiated to automatically generate or update the constraints.

Next, we illustrate how constraint templates can be derived and how they are

Table 3.1: Template structure

| |
|---|
| Instantiation context (IC) |
| Abstract constraint expression (ACE) |
| Variable definition (VD) |
| Instantiation information (II) |
| Data extraction expressions (DEE) |

managed by a template engine to automate constraint generation and updating.

### *3.2.1 Template Definition*

Templates are written manually by metamodel authors who are also in charge of maintaining and evolving metamodels. Before discussing the authoring process in detail, we discuss the structure of a template, as shown in Table 3.1, and the information it requires. The *instantiation context (IC)* defines for which elements, or combinations thereof, a template should be instantiated. The IC is thus an element of the metametamodel. Although we use a simple metametamodel in this paper for illustration purposes, the approach is generic and supports arbitrary metametamodels. The *abstract constraint expression (ACE)* is used to define the *family of constraints* generated from the template. A constraint family consists of constraints that share some static aspects (e.g., the structure) and have some variable parts that differ for each constraint. Thus, the ACE captures the static parts of the constraint family and also identifies the locations of variability which are also defined explicitly in the *variable definition (VD)*. The VD declares which parts of the ACE are interpreted as variables. To bind specific values to these variables, data has to be read from specific elements that are available when the template is instantiated. These elements are specified in the *instantiation information (II)*. How the values for the variables are extracted from the elements is declared in *data extraction expressions (DEE)*. We use OCL statements for the DEEs in this paper as it is a well known language for reading information from (meta)model elements. However, other languages and techniques may be used in specific implementations of our approach. Let us now show how we can write a template $T1$ for the constraint family of $R1$ and $R2$.

*Template for cardinalities.*

The top-right section "Template definition" in Fig. 3.3 illustrates the steps we perform next. The remainder of the figure depicts template instantiation and change management processes we discuss later. Template $T1$, shown in Table 3.2, creates a constraint for every instance of the metametamodel element`Reference` (see Section 3.1.1) in a metamodel, for example when the reference `target` is added to the class `Communication` during the initial modeling of our sample metamodel. Therefore, we define the IC of our template to be `<Reference>`. This means that we provide an instance of metametamodel element `Reference` to the template in order to create a new constraint. Note that templates are reusable for other metamodels that conform to the same metametamodel (e.g., other metamodels that are instances of the sample metamodel described in Section 3.1.1). We define the ACE by using the desired expression of one sample constraint of the constraint family (e.g., an OCL statement) and replacing all concrete values that are specific for a single instance with variables. In our example, we take the expression from the constraint $R1$ for the reference `Communication.target` in Fig. 3.1(a):

```
context Communication inv:
self.target->size()>=0 and
  self.target->size()<=1
```

And replace the two values 0 and 1 with `MIN` and `MAX` for the minimum and maximum number of connected elements, the context `Communication` with `C` for the checked class, and the two occurrences of `target` with `R` for the used reference. The result is the abstract constraint expression:

Table 3.2: Definition of template T1

| | |
|---|---|
| IC: | `<Reference>` |
| ACE: | `context C inv:`<br>`self.R->size()>=MIN and`<br>`self.R->size()<=MAX` |
| VD: | `<C, R, MIN, MAX>` |
| II: | `<Reference r>` |
| DEE: | `<C:r.owner.name,`<br>`R:r.name,`<br>`MIN:r.min,`<br>`MAX:r.max>` |

```
context C inv:
self.R->size()>=MIN and
self.R->size()<=MAX
```

as defined in Table 3.2 with the variable parts (VD) being `<C, R, MIN, MAX>`. As shown in Fig. 3.3, the instantiation information of $T1$ is `<Reference r>`.

Desired constraints are built by reading the `min`, `max`, and `name` values of the passed reference $r$ as well as the `name` of the class that owns the reference `owner.name`. The data extraction expressions can then be written as `r.min`, `r.max`, `r.name` and `r.owner.name`. In the DEEs, the variable to which the read data should be assigned is written before each DEE followed by a colon. Note that because of the single element instantiation context (i.e., we instantiate the template for every instance of that type), only one element is available as instantiation information, making both the II itself and the use of a prefix (i.e., "r") for the DEEs redundant. However, if more complex patterns were used in the IC, the II would contain more than one element from which DEEs read data. For example, we could have used the pattern `<Class,Reference>` as IC for $T1$ to generate a constraint for each reference that is actually added to a class. Then, distinguishing the class and the reference in the II and using prefixes in DEEs becomes necessary. We have now completed the template definition for $T1$.

Table 3.3: Definition of template T2

| | |
|---|---|
| IC: | `<DerivedReference>` |
| ACE: | `context C inv:`<br>`self.DR-> includesAll(`<br>`REFS->collect(x\|self.{x}))` |
| VD: | `<C, DR, REFS>` |
| II: | `<DerivedReference dr>` |
| DEE: | `<C:dr.owner.name,`<br>`DR:dr.name,`<br>`REFS:dr.refs->collect(name)>` |

*Template for derived references.*

We use the same process to write template $T2$, as shown in Table 3.3, based on the constraint $DR1$ as an example for the constraint family that checks derived references.

As a simplification, we replaced the set of references (`Set{self.source, self.target}->flatten()`) from $DR1$ in Fig. 3.1(a) with a construct (`collect(x|self.{x})`) that allows us to aggregate the results of different references – based on a set of reference names – dynamically. When the template is instantiated for the derived reference `Communication.inv`, the resulting constraint is:

```
context Communication inv:
self.inv->includesAll(
Set{“target”, “source”}->collect(x|self.{x}))
```

The expression `Set{“target”, “source”}->collect(x| self.{x})` then collects all the elements returned by the expressions `self.target` and `self.source`.

Now that the templates $T1$ and $T2$ are written, let us discuss how templates are instantiated automatically to generate constraints.

### *3.2.2 Template Instantiation*

To enable a template, it is passed to the template engine that observes a specific model and handles template instantiation and updating. We will now discuss how the template $T1$ for checking reference cardinalities is instantiated when it is applied to the metamodel in Fig. 3.1(a).

For each occurence of the IC `<Reference>`, the template is instantiated once. In Fig. 3.1(a) there are five references and thus $T1$ is instantiated five times. However, we focus on a detailed discussion of the instantiation process for the reference `Communication.target`, as illustrated in the bottom box "Template instantiation" in Fig. 3.3. The process starts with the instantiation information (1). In this case, it containts the reference `target`. The data extraction expressions are applied to the element to retrieve the names (i.e., `Communication` and `target`) and the cardinality values (i.e., `0` and `1`). This is shown in Fig. 3.3(2). In order to allow later updates of the generated constraints, the *constraint scope* is built automatically during the execution of the DEEs in step (2). This scope constains all elements that are accessed by the DEEs. The scope for the constraint $R1$ is therefore `<target.owner.name, target.name, target.min, target.max>`. The variables in the ACE are then replaced with these values to generate the constraint (3).

After applying our templates $T1$ and $T2$ to the initial version of our example metamodel from Fig. 3.1(a), template $T1$ was instantiated once for every reference (i.e., five times in total), template $T2$ was instantiated once to generate the constraint for the only derived reference `inv` in the metamodel.

At this point we have shown how templates are written and how they are instantiated. We have seen that a template captures the static and the variable parts of a family of constraints. Typically, a single constraint template is written for every constraint family in the system. Combining templates is only necessary in the rare cases where different constraint families should be merged into one. If such a merge is required, template authors can build the corresponding template by writing a template for the merged constraint families. Next, we will illustrate how automatic constraint updates are performed.

### *3.2.3 Change Management*

In Section 3.1 we discussed the effects of two metamodel evolutions on the correctness of constraints. We will now present how such metamodel evolutions are handled automatically by the template engine.

*Metamodel evolution.*

After every atomic modification of the metamodel (i.e., the addition, deletion, or modification of a metamodel element), the template engine is notified about the performed operation, as shown in the top-left box "Change management" in Fig. 3.3. The change notification is operation-based and includes information about the changed metamodel elements which the engine uses to determine the actions that are required to adapt the set of current constraints to the new version of the metamodel.

After the addition of metamodel elements, the engine looks for templates that can be instantiated (i.e., the types of the added model elements match the instantiation context). When metamodel elements are deleted, constraints that are based on these elements (i.e., their scope contains a removed element) are also removed. A metamodel element modification triggers the update process and the template engine uses the modified model element and the constraint scopes to calculate the set of affected constraints that need updating.

As an example, consider the metamodel version shown in Fig. 3.2. We first replaced the upper bound value `1` of the constraint $R1$ with the value `100`. The change notification that is passed to the engine indicates that the metamodel element `target.max` was modified. Since the scope of the constraint $R1$ contains the modified element, as discussed above, the engine detects that this constraint is affected by the modification. Because there are no other constraints that include the modified model element in their scope, $R1$ is identified as the only constraint that needs to be updated.

The update is performed by executing the data extraction expressions that added the modified metamodel element to the constraint's scope, as depicted by step (*) in Fig. 3.3, and replacing the outdated values in the constraint expression with the newly retrieved ones. In our example, `target.max` now returns the value `100`. Replacing the old value results in the new constraint expression

```
context Communication inv:
self.target->size()>=0 and
 self.target->size()<=100
```

And the constraint co-evolution was successfully completed. Note that currently we delete the existing constraint and re-instantiate the template to generate an updated constraint. The update of single values or logical fragments in the existing constraints will be addressed in future work.

The second metamodel modification we have to consider is the addition of the new derived reference `all` to `Component`. When the template engine is informed that a derived reference has been added, it automatically discovers that this element matches the instantiation context of template $T2$. Therefore, template instantiation is triggered and the instantiation information `<all>` is used by the data retrieval expressions to retrieve the values that are then used to replace the variables in $T2$ in order to produce the required constraint.

Finally, let us consider what would happen if we remove the derived reference `Communication.inv` in another evolution step. In that case, the template engine would identify $DR1$ as the only constraint that includes the removed element in its scope. Therefore, it would remove the no longer needed constraint $DR1$ from the metamodel automatically.

*Model evolution.*

As we have discussed in Section 3.1.3, changes of a model typically lead to a re-validation of affected constraint instances. With our approach, such changes can affect the scopes of generated constraint instances. For example, imagine the addition of a new component as a `target` of `ResultComm` in Fig. 3.1(b). Indeed, this may affect the consistency status of a constraint instance of $R1$. However, since such changes are handled entirely by the employed consistency checker, we omit a detailed discussion here and refer to [43].

## 3.3 Case Study – UML Metamodel Evolution

We evaluated the applicability and the performance of our approach with a case study that was done using a prototype implementation.

### *3.3.1 Prototype Implementation*

For the evaluation, we developed the *Cross-Layer Modeler (XLM)* [64]. This tool allows working with models and their metamodels at the same time. We used some of the concepts of the *Level-agnostic Modeling Language (LML)* [89], a modeling language explicitly designed for multilevel modeling, to implement a model that is capable of dealing with an arbitrary number of metalevels. The XLM leveraged from our previous work on the Model/Analyzer [43, 87] which supports efficient and scalable incremental consistency checking of arbitrary design constraints.

We extended the Model/Analyzer by adding an incremental template engine and the corresponding infrastructure to support the incremental creation, deletion and modification of constraints (based on meta model changes) which the Model/Analyzer then incrementally validates against model changes. Ten sample templates from different domains are available at the tool website `http://www.jku.at/sea/content/e139529/e126342/e157573`.

### *3.3.2 Metamodel and Models*

As our case study, we used templates and the Cross-Layer Modeler tool to automate constraint generation and updates for the UML. On the first glance, the UML meta model may not appear to be an ideal choice because it is rarely used for co-evolution. Nonetheless, for measuring the performance of our approach the UML is as suited as any meta model. The UML metamodel is clearly a sophisticated and complex metamodel for defining various types of diagrams related to object-oriented software engineering artifacts. And it is also very large (i.e., the UML metamodel was given as Ecore model which contained 6583 model elements). If our approach scales for UML model/meta-model evolution then this should be convincing. Additionally, numerous industrial software models are available [44]. For this case study we used the UML metamodel and industrial UML models with quite diverse characteristics from [44].

### *3.3.3 Performance Evaluation*

We used templates to automatically create constraints that check the structural integrity of UML model elements (e.g., modeled classes). Structural integrity is given if a model element provides the structural features as defined in the UML metamodel. Our constraints are based on the *ECore* metamodel and check the number of assigned elements as well as the assigned elements' types for every reference and attribute in the UML (e.g., every instance of `NamedElement` must have exactly one `String` object assigned as its `name`). The according templates are shown in Table 3.5 and Table **??**.

Table 3.4: Template for UML references

| | |
|---|---|
| IC: | `<EReference>` |
| ACE: | `<context NAME inv:`<br>`self.REFERENCE->size()>=MIN and`<br>`self.REFERENCE->size<=MAX and`<br>`self.REFERENCE->forAll(e|e instanceof TYPE)>` |
| VD: | `<NAME, REFERENCE, MIN, MAX, TYPE»` |
| II: | `<EReference r>` |
| DEE: | `<NAME:r.owner.name,`<br>`REFERENCE:r.name,`<br>`MIN:r.lowerBound,`<br>`MAX:r.upperBound,`<br>`TYPE:r.eType»` |

Table 3.5: Template for UML attributes

| | |
|---|---|
| IC: | `<EAttribute>` |
| ACE: | `<context NAME inv:`<br>`self.ATTRIBUTE->size()>=MIN and`<br>`self.ATTRIBUTE->size<=MAX and`<br>`self.ATTRIBUTE->forAll(e|e instanceof TYPE)>` |
| VD: | `<NAME, ATTRIBUTE, MIN, MAX, TYPE»` |
| II: | `<EAttribute a>` |
| DEE: | `<NAME:a.owner.name,`<br>`ATTRIBUTE:a.name,`<br>`MIN:a.lowerBound,`<br>`MAX:a.upperBound,`<br>`TYPE:a.eType»` |

Every test was performed 100 times on an Intel Core i5-650 machine with 8GB of memory running Windows 7 Professional. The median and average values were used for analysis. We classify the changes in our study in three categories.

*Category I. Metamodel evolution.*

Different metamodel modifications and common refactorings have been discussed in literature [37, 90, 83, 91, 92, 93]. During most common metamodel evolutions, references or attributes are added, removed, or are modified (e.g., the cardinality of an attribute is changed or an attribute is moved to another class). Therefore, we performed these kinds of evolutions with the UML metamodel. From this point on

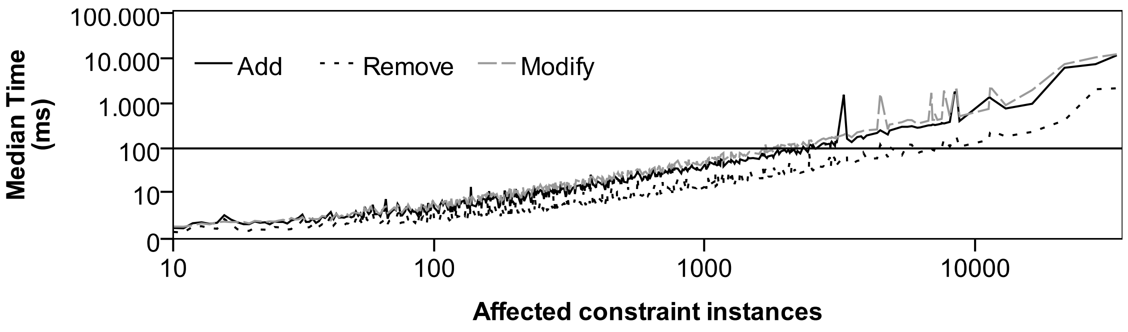


FIGURE 3.4: Metamodel evolutions

we will use the term *property* for references and attributes alike.

*Scenario 1. Add new property.* In the first scenario, a new property was added to every single element of the UML metamodel, which required the generation of a new constraint (as we discussed in Section 3.1.4 where we added a new derived reference to our sample metamodel). We investigated the total time required for performing the metamodel change, the required co-evolutions and the validation of the model with the new constraint. Note that for our statistics we only considered those changes that created constraints that could actually be validated with at least one model element (e.g., we ignored the addition of a new reference to `UseCase` if the model did not include any use cases). Fig. 3.4 shows the required processing times for changes that affected different numbers of constraint instances. 99% of all modifications took less than 166 ms to finish and only 0.15% of all performed changes took more than 500 ms. On average, changes took 12.5 ms and the generated constraint was validated with 201 constraint instances in the model. For the addition of elements in this test we observed a Pearson correlation coefficient of 0.845 between the required time and the number of required validations.

In Fig. 3.5, the median total processing times for different model sizes are depicted. Additionally, the times required for constraint management only (e.g., transformation of constraints without validation) and constraint validation per affected element (i.e., per element on which a constraint had to be re-validated) are shown.

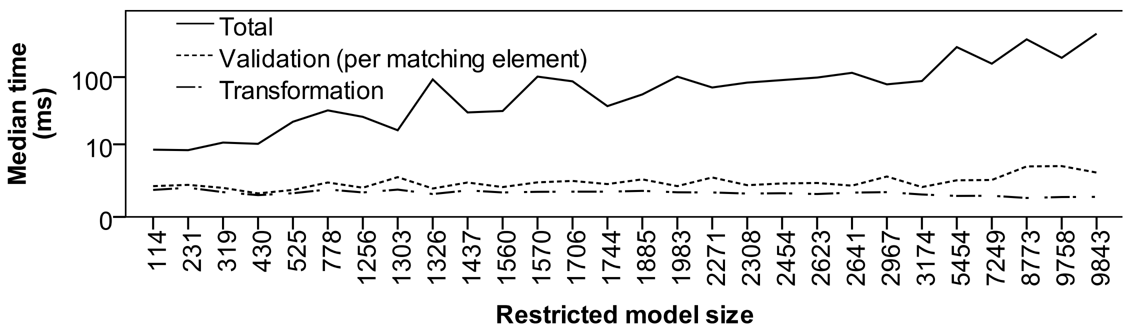


FIGURE 3.5: Metamodel element addition processing times

Not surprisingly, the time required for constraint management only is steady for all restricted models, regardless of their size. The correlation between $T$ and the model size $S$, $P(T, S)$ was 0.099, which also indicates that the processing time strongly depends on the validation effort needed for the new constraint and that it is independent from the model size.

For constraint validation, on the other hand, we observed that the total required time increases slightly with the restricted model size. Note that this is based on the fact that the generated constraints have to be validated not only one time, but once for every restricted model element that matches the constraint's context (e.g., all `Class` instances) and larger restricted models typically have higher numbers of those matching elements. Thus, the processing time for a specific metamodel change does not depend on the model size, but primarily on the model characteristics. Note that the spikes in the solid lines showing the total processing times in Fig. 3.5 are consistent with the average numbers of constraint-matching elements in the respective restricted models shown in Fig. 3.6. As the dashed line in Fig. 3.5 also indicates, the time required for constraint validation per validated element is nearly unaffected by the model size.

The median total validation time for a restricted model of just under 10,000 model elements was still below 500 ms for a source model change that required the constraint

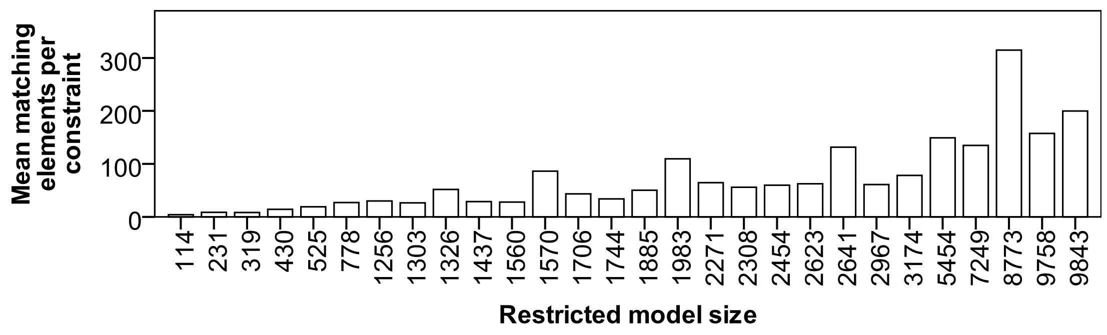


FIGURE 3.6: Average number of matching restricted model elements per constraint

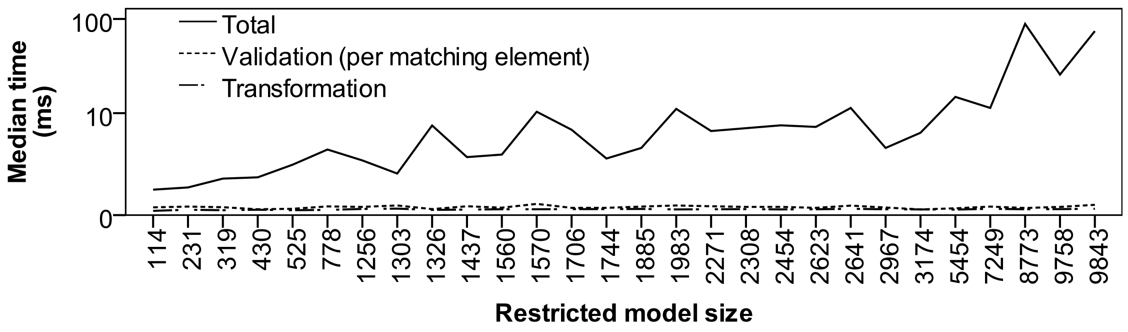


FIGURE 3.7: Metamodel element removal processing times

to be validated for an average of 200 different elements. During the transformation, data from 107 different metamodel elements was processed on average per generated constraint (i.e., implying that the transformations were far from trivial).

*Scenario 2. Remove existing property.* In the second scenario, each test run started with the unmodified UML metamodel and exactly one property was removed, meaning that exactly one constraint became obsolete and was removed from the consistency checker. Again, only changes of metamodel elements that were actually used in the model were captured. 99% of all modification took less than 38.5 ms. Only 0.1% of the modifications took longer than 250 ms. On average, element removal took 4.5 ms and 202 constraint instances were removed with the obsolete constraint. Fig. 3.4 shows that property removal is always faster than addition because there is no need for validating any constraint instances.

In Fig. 3.7, the total processing times for handling the removal of a metamodel element are depicted for different model sizes. Again, note that the time for constraint management (i.e., removing a constraint) is not affected by the model size. The time for validation per affected element (i.e., the removal of the consistency result for an element) is also stable. The total processing time, as discussed already in Scenario 1, does increase with the number of affected model elements.

*Scenario 3. Modify existing property.* For these tests, the cardinality as well as the name of every existing property in the UML were changed. 99% of the modification that caused an update of actually validated constraints were processed in less than 180 ms and 0.1% took more than 1,000 ms. For the modification of elements we observed a correlation coefficient of 0.734 between the required processing time and the number of validations.

*Category II. Model evolution.*

The incremental consistency checker that is used by the Cross-Layer Modeler, the Model/Analyzer, is highly scalable [44]. We previously evaluated the approach on 34 models with model sizes of up to 162,237 model elements and 24 types of consistency rules (constraints). Empirical evaluation showed that the consistency checking part requires only 1.4ms to re-evaluate the consistency of the model after a change for typical UML consistency and well-formedness constraints [48]. The data indicates that the additional change processing infrastructure does not impose a significant performance penalty.

*Category III. Template addition and removal.*

Even though adding, removing, or changing a template is a task performed less often than metamodel evolutions, we still investigated this aspect. Since the addition of a new template requires a full scan of the metamodel to create all possible constraints and a complete initial validation of the model we expected this task to be more

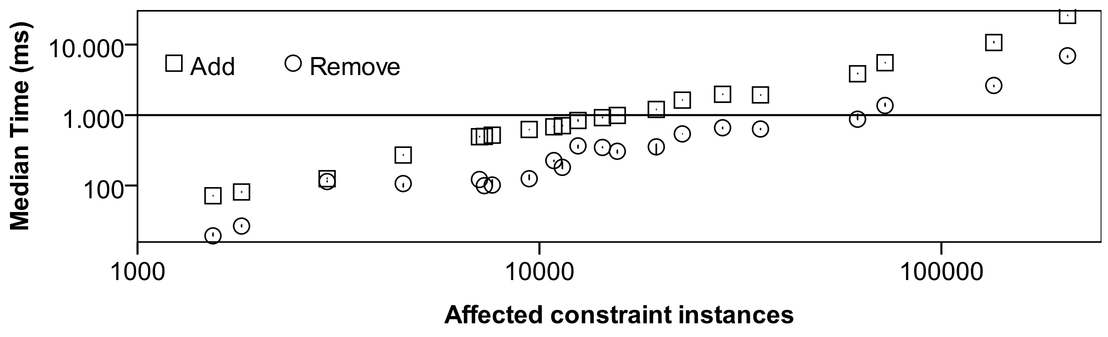


FIGURE 3.8: Template addition/removal

time consuming than processing changes incrementally. The processing times for the addition and removal of the templates we used in Category I to the UML metamodel that caused the generation or removal of different numbers of constraint instances are shown in Fig. 3.8. Adding a template took less than 5,700ms in 90% of our tests, in only 8% of the tests it took more than 10 s. On average, the addition of a template took 2,818ms and created constraints that were validated 31,936 times. Removing a template does not require validations of constraints, thus this task is performed in less than 1,600 ms in 90% of our tests. Only 5% of template removals take more than 3 s.

*Evaluation summary.*

The results of the representative metamodel evolutions clearly indicate that our approach is applicable to large and complex metamodels and that it is fast enough to deliver instant feedback about model consistency after metamodel changes. Processing changes that occur frequently during early development phases takes only milliseconds with our approach in most cases and even the worst case values are acceptable considering the fact that they were still below 16s and were reached in less than 1% of all changes. Although changing templates is slightly more expensive because of the inevitable processing of the entire model, the values are still acceptable for a rarely performed task.

### *3.3.4 Alternative Realization*

In this chapter, we have shown how the use of templates allows the generation of constraints from metamodels in an efficient way. Moreover, we have illustrated that creating a template only requires knowledge of the desired constraints and variable information.

To illustrate that the constraint management process discussed in this chapter is an application of the concepts introduced in Chapter 2, we also want to provide a set of traditional model transformation rules and helper functions that can be applied incrementally to generate constraints that check for structural correctness.

The UML metamodel is defined with the *Meta-Object Facility (MOF)* [94]. The elements available in UML are modeled as instances of the *Ecore* type `EClass`. Thus, it is possible to define a single transformation that uses `EClass` instances to generate constraints for the corresponding UML elements. The rule is shown in Listing 3.1 (lines 1–9). For a given `EClass`, which must not be abstract or an interface, it generates a constraint that is validated for all instances of the defined UML metamodel element (e.g., an `EClass` instance with the name `Message` is transformed to a constraint that checks all instances of `Message` in a UML model). Generating the invariants for the model elements is done in two separate helper functions called `generateAttributeInvariants` and `generateReferenceInvariants` for the `EAttribute` and `EReference` instances, respectively, belonging to the `EClass` either directly or through inheritance.

Function `generateAttributeInvariants` is shown in Listing 3.1 (lines 11–17). It takes as input an ordered set of attributes and recursively generates the invariant string by calling the helper `generateAttributeInvariant` with the first available attribute and itself with the remaining set of attributes.

For individual attributes, the helper `generateAttributeInvariant` only creates

```
rule EC
  from
    s : MetaModel!EClass (
      not s.isAbstract and not s.isInterface)
  to
    t : ConstraintModel!Constraint (
      context <- s.name;
      inv <- generateAttributeInvariants(s.eAllAttributes->asOrderedSet()) + " and "
          + generateReferenceInvariants(e.allReferences->asOrderedSet());
  )

helper def generateAttributeInvariants(s:OrderedSet(EAttribute)):String =
  if s->size()=0
  then
    "true"
  else
    let x=s->first() in generateAttributeInvariant(x) + " and " +
        generateAttributeInvariants(s-x)
  endif

helper def generateAttributeInvariant(s:EAttribute):String = cardinalityStatement(s.
    upperBound,s.lowerBound,s.name)

helper def cardinalityStatement(l:Integer,u:Integer,n:String):String =
  if (l=0 and u=-1) or (l=0 and u=1)
  then
    "true"
  else
    if l=1 and u=1
    then
      "self."+n+"<>null"
    else
      if l=0
      then
        "self."+n+"->size()<="+u
      else
        if u=-1
          "self."+n+"->size()>="+l
        else
          "self."+n+"->size()>="+l+"and self."+n+"->size()<="+u
        endif
      endif
    endif
  endif

helper def generateReferenceInvariants(s:OrderedSet(EReference)):String =
  if s->size()=0
  then
    "true"
  else
    let x=s->first() in generateReferenceInvariant(x) + " and " +
        generateReferenceInvariants(s-x)
  endif

helper def generateReferenceInvariant(s:EReference):String =
  cardinalityStatement(s.upperBound,s.lowerBound,s.name) + " and " +
      oppositeStatement(s,s.opposite)
```

Listing 3.1: Transformation rule and helper functions to generate constraints for UML metamodel types

a statement for checking cardinality by calling the helper `cardinalityStatement`, as shown in Listing 3.1 (line 19).

The helper `cardinalityStatement` generates invariants based on an associations lower and upper bounds, as shown in Listing 3.1 (lines 21–41). For unbounded associations (i.e., [0..1] for single-valued and [0..*] for multi-valued associations) the helper simply returns "true", for bounded associations the helper returns statements to check that an element provides the correct number of results for the given association.

Function `generateReferenceInvariants`, as shown in Listing 3.1 (lines 43–49), is a recursive helper similar to that for attributes, thus we omit a detailed discussion for space reasons.

To generate the invariants for an individual reference, the helper `generateReferenceInvariant`, shown in Listing 3.1 (lines 51–52), not only generates a cardinality statement, as it is done for attributes, but it also generates semantic invariants that check the correct implementation of opposite references. The latter is done through the helper `oppositeStatement`, which is shown in Listing 3.2.

Since OCL handles single-valued and multi-valued fields differently, the statement that checks the presence of opposite references must distinguish between different cardinalities for both the source and the opposite reference.

### *3.3.5 Threats to Validity*

Let us now discuss possible threats to the validity of the presented approach. First and foremost, the validation is based on a single metamodel only (i.e., UML). Therefore, the obtained results are only valid for the performed UML evolution and cannot be generalized. However, the UML was selected as the subject for the case study for two major reasons: i) the UML is a complex metamodel with hundreds of model

```
helper def oppositeStatement(x:EReference,y:EReference):String =
  if y=null
  then
    "true"
  else
    if x.upperBound=1
    then
      if y.upperBound=1
      then
        "self."+x.name+"."+y.name+"=self"
      else
        "self."+x.name+"."+y.name"+"->includes(self)"
      endif
    else
      if y.upperBound=1
      then
        "self."+x.name+"->forAll(z|z."+y.name+"=self)"
      else
        "self."+x.name+"->forAll(z|z."+y.name+"->includes(self))"
      endif
    endif
  endif
```

Listing 3.2: Helper function `oppositeStatement`

elements available and numerous interdependencies, and ii) UML models are among the most popular models in terms of industrial use. Thus, our case study is based on a well-established and rather complex metamodel that is used in industry. Moreover, the set of performed changes does include the atomic changes involved in complex metamodel refactorings. Although the observed results cannot be generalized, they provide strong evidence that our approach scales for typical domain-specific languages.

Another possible threat to validity is the correctness of constraint generation and updating. Our prototype implementation uses EMF-based models and metamodels, and employs the Model/Analyzer consistency checker. Moreover, it relies on a standard OCL interpreter to read information for template instantiation and builds the scopes using technologies from the well tested Model/Analyzer framework. The mechanisms used for triggering template instantiation and updating are also based on that framework. Therefore, we argue that our implementation works correctly given the existing technologies employed return correct results. However, we also performed systematic unit testing of the implementation to ensure correct behavior. Unit testing was also used to ensure the correct behavior of the implementation during the case study – we did not observe any cases where our implementation produced unexpected or unintended results (e.g., incorrect constraints).

## 3.4 Related Work

There has been an extensive research activity in models and their evolution. Here we focused on those closest to our work and grouped them in two themes.

**Metamodel and model (co-)evolution.** The efficient, and ideally automated, (tool-)support for metamodel evolution and the corresponding co-evolution of conforming models was identified by Mens et al. in 2005 as one of the major challenges in software evolution [27]. Since then, various approaches have been proposed to deal with this challenge. Wachsmuth addresses the issue of metamodel changes by describing them as transformational adaptations that are performed stepwise instead of big, manually performed ad hoc changes [93]. Changes to the metamodel become traceable and can be qualified according to semantics- or instance-preservation. He further proposes the use of transformation patterns that are instantiated with metamodel transformations to create co-transformations for models. Cicchetti et al. classify possible metamodel changes and decompose differences between model versions into sets of changes of the same modification-class [95]. They identify possible dependencies that can occur between different kinds of modifications and provide an approach to handle these dependencies and to automate model co-evolution.

Herrmannsdoerfer et al. also classified coupled metamodel changes and investigated how far different adaptations are automatable [96]. One aspect that these approaches have in common is that they are based on decomposing evolution steps into atomic modification for deriving co-adaptations. Our approach is also based on atomic modifications that are handled individually to perform necessary adaptations incrementally. However, we do not try to automate co-evolution of metamodels and models in the first place. Instead, the co-evolution of metamodels and constraints enables tool users to perform adaptations of a model with guidance based on specific constraints and their own domain knowledge.

In terms of constraint co-evolution, Büttner et al. discuss various metamodel modifications and how they affected constraints [69]. They describe how OCL expressions can be transformed to reflect metamodel evolution. We encountered some of the issues they identified during the evolution of our running example, for example the transition from single-object to collection values and vice versa because of multiplicity changes which is handled automatically in our prototype implementation.

**Flexible and multilevel modeling.** Atkinson and Kühne identified several issues in the field of multilevel (meta-)modeling, namely the so-called *shallow instantiation* of the UML [9] that forced us to use a graph-oriented model in XLM. They discussed different approaches to overcome these issues like the concept of *deep instantiation* where instances can be types at the same time; an approach we used in our tool. Ossher et al. lately presented the *BITKit* tool [63] that allows domain-agnostic modeling and on-the-fly assignment of visual notations to dynamically defined domain types. This approach is also implemented in our tool where the type of a model element can be changed at any time. Generally, our approach can be used with multilevel modeling tools, which typically do not support automatic consistency checking, and supports constraint templates that are based on arbitrary metalevels.

## 3.5 Summary

This chapter presented an approach that is based on the principles of CDM and uses constraint templates and an automated template engine to address the issue of co-evolving metamodels and constraints. We illustrated how constraint templates can be written and constraints are generated from them. Moreover, we discussed how automatic co-evolution of constraints is achieved and developed a prototype implementation. We performed a case study with UML as an example of a sophisticated metamodel and 21 industrial UML models that clearly showed that our approach is applicable for complex metamodels. The approach is scalable and processing times for co-evolution are primarily affected by the number of required validations after constraint generation or update.

# 4

# Incremental Safe Configuration for Software Product Lines

*Software Product Lines (SPLs)* [97] have become more common in recent years as they foster and support software reuse, enabling quicker development of products. Individual units of functionality are commonly bundled in *features* [98]. The functionality shared between all products as well as the possible points of variation (i.e., product variability) are defined in *feature models* that describe how features are related and how they can be combined. From feature models, products can easily be configured by selecting the features a product should provide. Based on such a selection, assets associated with the selected features are combined to construct products that can be delivered to customers. For software product lines in particular, those assets typically consist of source code, test scripts, or software models that are combined – transformed to code in the case of models – and compiled.

Although the SPL concepts seem to be straightforward to implement, several common challenges have been identified [99]. One of the biggest threats are errors in a feature model or the traceability between features and the assets that implement them. Such errors may lead to products that are correct with respect to the feature

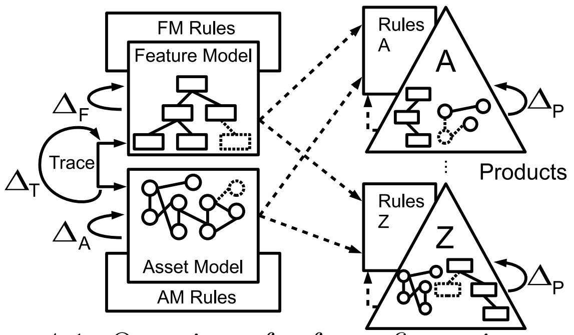


FIGURE 4.1: Overview of safe configuration approach

model but that cannot be compiled because of – just to name one example – missing files. Hence, checking the correctness of products is essential for successfully using SPLs [100].

In practice, a product line is likely to change as new functionality is added over time, changing both the available features and the set of available assets [101]. For example, source code or design model refactorings are common tasks that lead to changes in underlying assets. Additionally, customer requirements often cannot be fully foreseen by product line designers and domain analysts because of the large number of features, their interactions and constraints, or market changes. Thus, customers may demand products that are not valid with respect to the feature model or that include assets to enable requested functionality not described by the current feature model. The efficient handling of such scenarios is a challenge for typical SPL consistency checking techniques as they often check all possible products at once using traditional model checking approaches such as SAT-solvers (e.g., [102, 103, 98]).

This chapter introduces *Safe Configuration*, an incremental approach to ensure the consistency of both the product line definition (i.e., feature model and asset model) and configured products, as depicted in Fig. 4.1. Notice that, in contrast to other approaches for SPL consistency checking (e.g., [98]), safe configuration can handle unforeseen product configurations (i.e., it can check whether a product that does not conform to the product line definition is still working correctly), making it both versatile and broadly applicable. Furthermore it is generic and uses model-based representations of arbitrary assets, making it independent of the technologies used for product line implementation. Our approach does not only enforce consistency rules on feature and asset models (denoted as partially surrounding frames), but also enforces specific sets of consistency rules, which are based on feature and asset models, on individual products.[1] In contrast to other SPL checking approaches, as we discuss in Section 4.6 in detail, safe configuration checks individual products with consistency rules that are updated after feature or asset model changes automatically. This allows safe configuration to flag invalid products and indicate which rules are inconsistent – also after SPL evolution – making it easier for product line designers to determine the impact of a change and the effort required for fixing the problem. The dashed arrows indicate that the applied rules may stem from the features and assets actually selected or used in a product, or from the feature and asset models directly. Safe configuration supports evolution of product line definition as well as product configuration. More concretely it handles changes on the following parts of SPLs:

1. feature model ($\Delta_\mathrm{F}$) – e.g., addition of new features,
2. asset model ($\Delta_\mathrm{A}$) – e.g., source code refactorings,
3. individual product ($\Delta_\mathrm{P}$) – e.g., selection of a feature,
4. traces ($\Delta_\mathrm{T}$) between features and their implementing assets

Thus, our approach handles changes of all important artifacts in an SPL.

The novel contributions of safe configuration include: i) the checking of individual, actually configured products with product-specific sets of consistency rules that

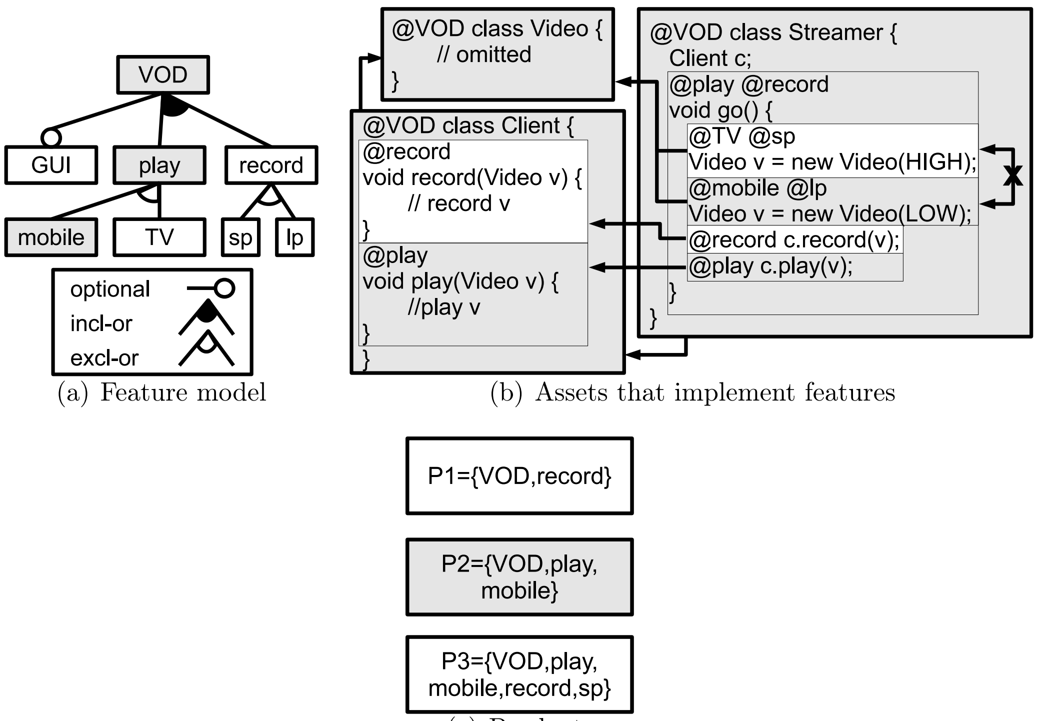
[1] We use the term *consistency rules* instead of *constraints* to indicate that those restrictions are applied to arbitrary representations rather than typical – for instance, EMF-based – models. Furthermore, the term constraint is already used in the SPL community, as discussed below.

combine constraints imposed by both feature model and asset model, and ii) the efficient handling of evolution in all parts of the product line (i.e., feature model, asset model, and each configured product).

We developed a prototype implementation that integrates a framework for defining product lines and products with an incremental consistency checker in order to demonstrate the feasibility of the approach as well as to assess its performance. Tests with a large scale product line of a crash management system [104] – a benchmark system for the *Model-Driven Development (MDD)* [1] community available in the *Repository for Model Driven Development (ReMoDD)* [105] – which consists of 66 features and 921 associated assets showed that both the evolution of the product line definition and changes of individual products are handled efficiently and lead to immediate consistency feedback to the user.

## 4.1 Motivating Example – Video-on-demand System

We use a small excerpt of a software product line of a video-on-demand system (VOD) [106] to illustrate the evolution scenarios safe configuration supports. The system can provide functionality for video playback as well as video recording. Moreover, it can be configured for execution on either a mobile device or a television. The corresponding feature model, using using the notation from [107] and [108], is depicted in Fig. 4.2(a). We used an `inclusive-or`-association for modeling the link between the root feature (`VOD`) and the features `play` and `record`, meaning that at least one of the two sub-features must be selected. An `exclusive-or`-association links the sub-features `mobile` and `TV` to the feature `play`, meaning that exacly one playback mode has to be selected. If video streams should be recorded, two recording modes are available: standard-play (`sp`) for recordings in high-definition and long-play (`lp`) for recordings in low-definition. Finally, there is an optional feature `GUI` that adds additional user interface functions.



(a) Feature model

```
@VOD class Video {
        // omitted
}
```

```
@VOD class Client {
@record
void record(Video v) {
        // record v
}
@play
void play(Video v) {
        //play v
}
}
```

```
@VOD class Streamer {
  Client c;
  @play @record
  void go() {
    @TV @sp
    Video v = new Video(HIGH);
    @mobile @lp
    Video v = new Video(LOW);
    @record c.record(v);
    @play c.play(v);
  }
}
```

(b) Assets that implement features

(c) Products

FIGURE 4.2: Video-on-demand product line with feature model, assets, and products. Product P2 marked gray

VOD is implemented in Java and consists of three classes for a streaming server, a video client, and a video itself. The sequence of actions for video playback is as follows: when a video is selected for playback, the streaming server is notified, the video is prepared by the streamer, and video playback is started at the requesting client as soon as the video is available for streaming. For recording, the streamer informs the client as soon as the video stream is ready for recording. Note that playback and recording may use videos of different quality, for example to record a low-definition version of the played video to save disk space. The assets are depicted in Fig. 4.2(b).

The variation points in the feature model (e.g., the playback mode) are of course also considered in the implementation. For example, there are two different statements present in the `Streamer` for preparing the video for playback – one loads a high-definition version for `TV` playback and one loads a low-definition version for the `mobile` device that typically has a lower bandwidth connection. As shown in Fig. 4.2(b), we used in annotations in our example for linking individual variables, methods, statements, or classes to features (e.g., `@play c.play(v)` in `Streamer.go()`). Indeed, there are dependencies between the different parts of source code. For example, the variable `Client c` in `Streamer` requires the class `Client`. Such dependencies are shown as arrows in Fig. 4.2(b). Obviously, only one of the two video preparing statements in `Streamer.go()` can be used at the same time (as indicated by the crossed arrow connecting the two statements) without getting compilation errors.

In Fig. 4.2(c), three configured products are depicted. Products are defined by the set of selected features and the assets that implement these features. The features and assets for `P2` are marked gray in 4.2(a) and 4.2(b), respectively. We omit listing the assets of the products explicitly for readability reasons. Checking the consistency of a product requires consistency rules inferred from: a) the product's selected features based on the feature model, b) the used assets, and c) the feature and asset models directly. For instance, in `P2` the following are examples of these inferred consistency rules: for a) the selected feature `play` requires either `mobile` or `TV` to be also selected; for b) the used class `Client` requires the class `Video` to be used; and for c) the feature model's root feature `VOD` must be selected (i.e., in feature models the root feature is always selected). We elaborate more on this issue in Section 4.3.

In contrast with other approaches (e.g., [98, 102]), safe configuration enforces consistency rules only on configured products (i.e., `P1`–`P3` in Fig. 4.2(c)) and does not consider products that are possible but not actually in production. Moreover, it does not enforce all possible rules on a product, but only those rules that are actually relevant. In our example, this means that the product `P2` is only checked with rules for the choice of `play` and `record` (i.e., that at least one of those features is selected), the alternative of `mobile` and `TV` (i.e., that exactly one is selected), and rules for the

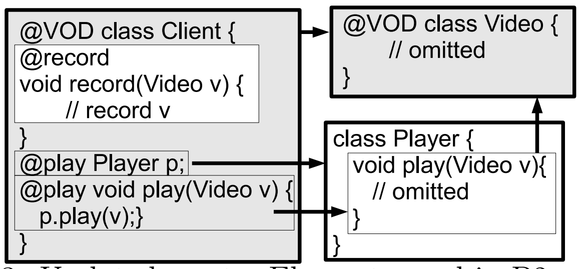


FIGURE 4.3: Updated assets. Elements used in P2 marked gray

used assets (e.g., that only one video preparation statement is used).

### *4.1.1 Product Line Evolution Examples*

Let us now present some sample product line evolutions that may occur over time.

*Refactoring of video playback ($\Delta_A$).*

We begin with a refactoring of the implementation of the VOD system – an evolution of the asset model ($\Delta_\mathrm{A}$). For example, we could change the class `Client` and its method `play(Video v)`, as shown in Fig. 4.3 and introduce a new class `Player` with the method `play()` in order to strenghten the separation of concerns in the implementation. The method `Client.play(Video v)` now additionally requires the method `Player.play()` and the newly introduced variable `Player p`, which of course requires the class `Player`. After these changes, the previously valid product `P2` becomes invalid because it does not include the new asset `Player`.

*Fixing the product line ($\Delta_T$ or $\Delta_P$).*

For the dangling reference problem introduced above, there are two intuitive fixes: a) we could add the class `Player` and its methods as implementation assets for the feature `play` (i.e., add annotations) – a change to the defined traces between features and assets ($\Delta_\mathrm{T}$), and b) we could manually add the missing assets for `Player` and `Player.play()` to the products `P2` and `P3` without changing the product line definition (i.e., without linking the asset to features through annotations) and without possible effects on other products – a change of a single product ($\Delta_\mathrm{P}$). Since safe configuration considers both the selected features and the used assets for determining whether a product is valid, both evolutions (fixes) can be handled: `P2` becomes valid again.

*Changing variability ($\Delta_F$).*

Let us consider an evolution of variability. For example, we could change the feature `GUI` to be no longer optional but mandatory for all video-on-demand systems – an evolution of the feature model ($\Delta_\mathrm{F}$). After this evolution, safe configuration would update the respective sets of applied rules for the existing products `P1`–`P3` and would detect that all of them become invalid as they do not have the feature `GUI` selected.

Before we explain in Sections 4.3 and 4.4 how our approach generates rules and incrementally checks their consistency, let us give a formal definition of product lines.

## 4.2 Product Line Terminology

We define a single model to hold all the information required for our product line (PL) definition that consists of three parts: a) feature model (FM), b) asset model (AM), and c) product model (PM).

$$\mathrm{PL} := \langle \mathrm{FM}, \mathrm{AM}, \mathrm{PM} \rangle \tag{4.1}$$

### *4.2.1 Feature Model*

The feature model contains a set of features (F) and is used for describing the different functionality a product can provide at a high level of abstraction [97]. Moreover, the model contains feature associations (FA) that express how features are related and how features should be selected during configuration. Though many different notations for defining feature models and feature associations have been proposed (e.g., [97, 109, 98, 110, 111, 112]), we follow the standard terminology in literature (e.g., [19, 110, 107]) and define the following kinds of feature associations: mandatory, optional, inclusive-or, and exclusive-or. Next, the model contains additional cross-tree feature constraints (FC) that express more general relations between features besides the feature model's tree-like hierarchical structure. Feature constraints either require or exclude other features. Finally, one of the existing features is selected as the models root feature.

$$\mathrm{FM} := \langle \mathrm{F}, \mathrm{FA}, \mathrm{FC}, \mathrm{root} \in \mathrm{F} \rangle \tag{4.2}$$

$$\mathrm{FA} := \{\langle k \in \mathrm{Kind}_{\mathrm{FA}}, \mathrm{s} \in \mathrm{F}, \mathrm{t} \in \mathcal{P}(\mathrm{F}) \rangle\} \tag{4.3}$$

$$\mathrm{Kind}_{\mathrm{FA}} := \{\mathrm{mand}, \mathrm{opt}, \mathrm{or}, \mathrm{xor}\} \tag{4.4}$$

$$\mathrm{FC} := \{\langle k \in \mathrm{Kind}_{\mathrm{FC}}, s \in \mathrm{F}, t \in \mathcal{P}(\mathrm{F}) \rangle\} \tag{4.5}$$

$$\mathrm{Kind}_{\mathrm{FC}} := \{\mathrm{requires}, \mathrm{excludes}\} \tag{4.6}$$

### *4.2.2 Asset Model*

The asset model of the product line includes all the assets (A) that can be used to implement the features defined in the feature model. These assets are typically source code files, configuration information or data schemas, image files used by user interfaces, etc. Moreover, abstractions of code can also be used, for example UML models, from which source code can be generated, compiled, and delivered. Note that assets in our asset model are generic and can represent arbitrary artifacts at any granularity level. For example, an asset can be defined for an entire file or also for a method defined at a specific position in a source code file. This interpretation of assets stems from common product line tools and techniques that allow for the composition of individual methods or even variables in source files or design models [113, 114].

In contrast to features, assets are not organized hierarchically (i.e., there is no root asset). However, as there may be dependencies between assets (e.g., method `a` calls method `b`) or also conflicts if certain assets are present in a system at the same time (e.g., alternative implementation of a method), thus there may be asset constraints (AC) that describe such relations (as depicted with solid and crossed arrows in Fig. 4.2(b)). In our model, we allow the following kinds of asset constraints: requires, excludes, inclusive-or, exclusive-or.

$$\mathrm{AM} := \langle \mathrm{A}, \mathrm{AC} \rangle \tag{4.7}$$

$$\mathrm{AC} := \{\langle k \in \mathrm{Kind}_{\mathrm{AC}}, s \in \mathrm{A}, t \in \mathcal{P}(\mathrm{A}) \rangle\} \tag{4.8}$$

$$\mathrm{Kind}_{\mathrm{AC}} := \{\mathrm{requires}, \mathrm{excludes}, \mathrm{or}, \mathrm{xor}\} \tag{4.9}$$

### *4.2.3 Product Model*

Finally, we have to define the product model. This model simply consists of individual configurations. Each configuration is identified by two sets, one holding the selected features and one holding the assets from which the configured product is composed.

$$\mathrm{PM} := \{\langle f \in \mathcal{P}(\mathrm{F}), a \in \mathcal{P}(\mathrm{A}) \rangle\} \tag{4.10}$$

### *4.2.4 Helper Functions*

In order to make data retrieval and the generation of consistency rules easier and more understandable, we define some additional helper functions.

*Feature model.*

For feature associations, we can query the corresponding source feature (Eq. (4.11)) or the set of target features (Eq. (4.12)). We overload the functions source and target to also accept feature constraints as input. Moreover, we want to easily access the associations (Eq. (4.13)) or constraints (Eq. (4.14)) for which a given feature is the source.

$$\text{source} : \mathrm{FA} \rightarrow \mathrm{F}, \langle k, s, t\rangle \mapsto s \tag{4.11}$$

$$\text{target} : \mathrm{FA} \rightarrow \mathcal{P}(\mathrm{F}), \langle k, s, t\rangle \mapsto t \tag{4.12}$$

$$\text{asso} : \mathrm{F} \rightarrow \mathcal{P}(\mathrm{FA}), f \mapsto \{x \in \mathrm{FA} | \text{source}(x) = f\} \tag{4.13}$$

$$\text{fcon} : \mathrm{F} \rightarrow \mathcal{P}(\mathrm{FC}), f \mapsto \{x \in \mathrm{FC} | \text{source}(x) = f\} \tag{4.14}$$

*Asset model.*

For the asset model we overload the helper functions `source` and `target` to retrieve source and target assets of asset constraints (see Eq. (4.15) and Eq. 4.16). Moreover, we define a function to retrieve the asset constraints for which an asset is the source (Eq. (4.17)). Additionally, we want to retrieve those assets that are implementing a specific feature (Eq. (4.18)).

$$\text{source} : \mathrm{AC} \rightarrow \mathrm{A}, \langle k, s, t\rangle \mapsto s \tag{4.15}$$

$$\text{target} : \mathrm{AC} \rightarrow \mathcal{P}(\mathrm{A}), \langle k, s, t\rangle \mapsto t \tag{4.16}$$

$$\text{acon} : \mathrm{A} \rightarrow \mathcal{P}(\mathrm{AC}), a \mapsto \{x \in \mathrm{AC} | \text{source}(x) = a\} \tag{4.17}$$

$$\text{impl} : \mathrm{F} \rightarrow \mathcal{P}(\mathrm{A}), f \mapsto \{x \in \mathrm{A} | x \text{ implements } f\} \tag{4.18}$$

In our running example, this traceability information used in Eq. (4.18) is defined by the used annotations. However, our approach supports various ways of establishing this information (e.g., a feature-asset matrix) and is not limited to annotations. Therefore, we omit an explicit definition of the implementation relation Eq. (4.18).

*Product model.*

For each product we want to access the selected features (Eq. (4.19)) and the used assets (Eq. (4.20)) directly.

$$\text{selected} : \mathrm{PM} \rightarrow \mathcal{P}(\mathrm{F}), \langle f, a\rangle \mapsto f \tag{4.19}$$

$$\text{uses} : \mathrm{PM} \rightarrow \mathcal{P}(\mathrm{A}), \langle f, a\rangle \mapsto a \tag{4.20}$$

Next, we show how this product line definition is used for deriving consistency rules.

## 4.3 Safe Configuration

As we will show next, the approach of safe configuration enables incremental and efficient consistency checking of evolving product line definitions and configurations. As we have briefly illustrated in our running example, checking the overall consistency of a product line involves checking the correctness of: a) the feature model, b) the asset model, and c) the products. As different parts of the product line evolve, we want the required consistency rules to be managed and validated incrementally in order to enable fast and efficient re-validation of both individual products and the entire product line. Before we consider the aspect of evolution, let us describe in detail the different kinds of rules that are required.

### *4.3.1 Feature Model Rules*

For the feature model we have to ensure certain syntactical and semantical properties that have been identified and discussed exhaustively in literature (e.g., [111, 110]), thus we omit a detailed discussion here. As an example of these rules, features must be organized hierarchically in a tree and all but the root feature must be linked to exactly one parent feature. We considered the following rules: i) features (except the root) must have one parent, ii) features must not require features they also exclude, iii) features must not require features that exclude them. The feature model in Fig. 4.2(a) is valid according to those standard rules.

### *4.3.2 Asset Model Rules*

For the asset model, a simple consistency rule to ensure that consistent asset selections are possible is shown in Eq. (4.21). The set of assets that are required by an asset (derived by $\text{getReq}_\text{A}$ and $\text{getReqDir}_\text{A}$; shown in Eq. (4.22) and Eq. (4.23), respectively) must not include assets that are excluded by one of the required assets (derived by $\mathrm{getExcl}_\mathrm{A}$; shown in Eq. (4.24)).

$$\begin{aligned} \mathrm{valid}_\mathrm{A} &: \mathrm{A} \to \{true, false\}, \\ a &\mapsto \mathrm{getReq}_\mathrm{A}(a) \cap \mathrm{getExcl}_\mathrm{A}(a) = \emptyset \end{aligned} \tag{4.21}$$

$$\begin{aligned} \mathrm{getReq}_\mathrm{A} &: \mathrm{A} \to \mathcal{P}(\mathrm{A}), \\ a &\mapsto \{x \in \mathrm{A} | x \in \mathrm{getReqDir}_\mathrm{A}(a) \vee \exists y \in \mathrm{getReqDir}_\mathrm{A}(a) : x \in \mathrm{getReq}_\mathrm{A}(y)\} \end{aligned} \tag{4.22}$$

$$\begin{aligned} \mathrm{getReqDir}_\mathrm{A} &: \mathrm{A} \to \mathcal{P}(\mathrm{A}), \\ a &\mapsto \{v \in \mathrm{A} | \exists \langle x, y, z \rangle \in \mathrm{AC} : x = \mathrm{requires} \wedge y = a \wedge v \in z\} \end{aligned} \tag{4.23}$$

$$\begin{aligned} \mathrm{getExcl}_\mathrm{A} &: \mathrm{A} \to \mathcal{P}(\mathrm{A}), \\ a &\mapsto \{v \in \mathrm{A} | \exists w \in \mathrm{getReq}_\mathrm{A}(a), \langle x, y, z \rangle \in AC : x = \mathrm{excludes} \wedge y = w \wedge v \in z\} \end{aligned} \tag{4.24}$$

As Eq. (4.25) shows, an asset model is valid if no asset violates the rule defined in Eq. (4.21)

$$\begin{aligned} \mathrm{valid}_\mathrm{AM} &: \mathrm{AM} \to \{true, false\}, \\ \langle a, ac \rangle &\mapsto \forall x \in a : \mathrm{valid}_\mathrm{A}(x) \end{aligned} \tag{4.25}$$

The asset model depicted in Fig. 4.2(b) is valid with respect to the defined consistency rule.

### *4.3.3 Product Model Rules*

The rules we have discussed so far check the consistency of feature and asset models, that is, they ensure a valid definition of variability and available assets. Now we discuss the rules required for checking individual products.

We define a rule on a product as a function that relates a product to a boolean value:

$$\mathrm{PM} \to \{true, false\}$$

To derive the set of rules needed for a product, both the selected features and the used assets have to be considered, as shown in Eq. (4.26).

$$\begin{aligned} \text{rules} : \text{PM} &\to \{\text{PM} \to \{true, false\}\}, \\ \langle f, a\rangle &\mapsto x | \forall y \in f : \text{genRules}(y) \subseteq x \wedge \\ &\quad \forall z \in a : \text{genRules}(z) \subseteq x \end{aligned} \tag{4.26}$$

For `P2` from from Fig. 4.2(c), this would generate rules for the features and assets marked gray in Fig. 4.2(a) and Fig. 4.2(b), respectively. The rules necessary for a specific feature include the rules for corresponding feature associations and rules for corresponding feature constraints, as derived in Eq. (4.27).

$$\begin{aligned} \text{genRules} : \text{F} &\to \{\text{PM} \to \{true, false\}\}, \\ f &\mapsto \text{genRules}(\text{asso}(f)) \cup \text{genRules}(\text{fcon}(f)) \end{aligned} \tag{4.27}$$

For the feature `play` in our running example, this means that one rule for the exclusive-or-association is needed. For the set of feature associations, the set of corresponding rules can easily be calculated, as shown in Eq. (4.28), by generating one rule for each association.

$$\begin{aligned} \text{genRules} : \mathcal{P}(\text{FA}) &\to \{\text{PM} \to \{true, false\}\}, \\ s &\mapsto \{x | \exists fa \in s : x = \text{genRule}(fa)\} \end{aligned} \tag{4.28}$$

In Eq. (4.29) we define how rules for feature association are derived. Note that the derived rule uses specific information of the association.

$$\begin{aligned} \text{genRule} : \text{FA} &\to (\text{PM} \to \{true, false\}), \\ \langle k, s, t\rangle &\mapsto \begin{cases} \langle f, a\rangle \mapsto t \subseteq f & \text{if } k = mand \\ \langle f, a\rangle \mapsto f \cap t \neq \varnothing & \text{if } k = or \\ \langle f, a\rangle \mapsto |f \cap t| = 1 & \text{if } k = xor \end{cases} \end{aligned} \tag{4.29}$$

For the inclusive-or-association from `VOD`, for instance, one rule is derived which checks that the intersection of selected features and the set $\{\texttt{play}, \texttt{record}\}$ is not empty (i.e., that at least one of the two features is selected).

Similar to feature associations, we also derive a single rule for each feature constraint, as shown in Eq. (4.30).

$$\begin{aligned} \text{genRules} &: \mathcal{P}(\text{FC}) \to \{\text{PM} \to \{true, false\}\}, \\ s &\mapsto \{x | \exists fc \in s : x = \text{genRule}(fc)\} \end{aligned} \tag{4.30}$$

For feature constraints, the function defined in Eq. (4.31) derives the corresponding rules.

$$\begin{aligned} \text{genRule} &: \text{FC} \to (\text{PM} \to \{true, false\}), \\ \langle k, s, t \rangle &\mapsto \begin{cases} \langle f, a \rangle \mapsto t \subseteq f & \text{if } k = requires \\ \langle f, a \rangle \mapsto f \cap t = \varnothing & \text{if } k = excludes \end{cases} \end{aligned} \tag{4.31}$$

As with features and feature constraints, we generate one rule for each asset constraint that is associated with an asset, as shown in Eq. (4.32).

$$\begin{aligned} \text{genRules} &: \text{A} \to \{\text{PM} \to \{true, false\}\}, \\ a &\mapsto \{x | \exists ac \in \text{acon}(a) : x = \text{genRule}(ac)\} \end{aligned} \tag{4.32}$$

The rule derivation for asset constraints is defined in Eq. (4.33).

$$\begin{aligned} \text{genRule} &: \text{AC} \to (\text{PM} \to \{true, false\}), \\ \langle k, s, t \rangle &\mapsto \begin{cases} \langle f, a \rangle \mapsto t \subseteq a & \text{if } k = requires \\ \langle f, a \rangle \mapsto a \cap t = \varnothing & \text{if } k = excludes \\ \langle f, a \rangle \mapsto a \cap t \neq \varnothing & \text{if } k = or \\ \langle f, a \rangle \mapsto |a \cap t| = 1 & \text{if } k = xor \end{cases} \end{aligned} \tag{4.33}$$

For example, the asset `Client` in 4.2(b) has one requires-constraint with target `Video`. The resulting rules for `Client` would thus consist of one rule which checks that a product's used assets include the class `Video`. Note that we overloaded the functions **`genRules`** and **`genRule`** to enable generic reuse.

To obtain the consistency status of a specific product with respect to a single rule we can define the function shown in Eq. (4.34).

$$\begin{aligned} \text{result}_{\text{S}} &: (\text{PM}, \text{PM} \to \{true, false\}) \to \{true, false\}, \\ (c, r) &\mapsto r(c) \end{aligned} \tag{4.34}$$

Moreover, we can define a function that checks consistency for a specific product and a set of rules as in Eq. (4.35).

$$\begin{aligned}\text{result} : (\text{PM}, \{\text{PM} \to \{true, false\}\}) \to \{true, false\},\\ (c, r) \mapsto \forall x \in r : \text{result}_{\text{S}}(c, x)\end{aligned} \tag{4.35}$$

For an individual product consistency can be checked using the function defined in Eq. (4.36).

$$\begin{aligned}\text{check} : \text{PM} \to \{true, false\},\\ c \mapsto \text{result}(c, \text{rules}(c))\end{aligned} \tag{4.36}$$

Finally, the entire product line is valid and free of errors if the function defined in Eq. (4.37) returns *true*.

$$\begin{aligned}\text{valid}_{\text{PL}} : \text{PL} \to \{true, false\},\\ \langle \text{FM}, \text{AM}, \text{PM} \rangle \mapsto \forall c \in \text{PM} : \text{check}(c)\end{aligned} \tag{4.37}$$

In our running example, `P2` is valid while `P1` and `P3` are invalid.

Note that even with inconsistent feature or asset models, products may be valid regarding feature selection and asset usage. Moreover, products that are invalid because of incorrect feature selection may still be working if the asset usage is consistent. To determine whether the asset combination of a product is valid, it is sufficient to only apply rules that are based on the used assets. To derive these rules, the function in Eq. (4.38) can be used.

$$\begin{aligned}\text{rules}_{\text{working}} : \text{PM} \to \{\text{PM} \to \{true, false\}\},\\ \langle f, a \rangle \mapsto \text{rules}(\varnothing, a)\end{aligned} \tag{4.38}$$

And, to check whether a product is working without taking into account the correctness of the feature selection, the function shown in Eq. (4.39) can be used.

$$\begin{aligned}\text{working} : \text{PM} \to \{true, false\},\\ c \mapsto \text{result}(c, \text{rules}_{\text{working}}(c))\end{aligned} \tag{4.39}$$

By substituting Eq. (4.26) by Eq. (4.38) in all functions that will be presented in the following sections, the validity of products with respect to only asset usage can be checked.

Next, we discuss how we manage consistency rules and determine the consistency of products incrementally.

## 4.4 Incremental Rule Management and Consistency Checking

Above, we have described the generic consistency rules for feature and asset models as well as functions for generating consistency rules for individual products. Now we will discuss how our approach handles various kinds of changes to the feature model, the asset model, or products.

### *4.4.1 Standard Product Configuration ($\Delta_P$)*

Let us begin with the standard product configuration process as this is the most common action – an evolution of a single product ($\Delta_\mathrm{P}$) in Fig. 4.1 – and the functions derived for product configuration will be reused later for other actions.

Selecting a new feature in a product means that the feature is added to the set of selected features and the assets that implement the feature are added to the set of used assets. Unselecting a feature in a product means that the unselected feature is removed from the set of selected features and that all assets that were exclusively used for implementing the removed feature are removed. Based on this behavior, we can calculate sets of features and assets that will be added (or removed) through an (un)selection, as shown in Eq. (4.40) and Eq. (4.41), which we call *product deltas*.

$$\begin{gathered}\mathrm{select}_\Delta : (\mathrm{PM}, \mathrm{F}) \rightarrow \mathrm{PM},\\ (\langle f, a\rangle, x) \mapsto \langle \Delta f, \Delta a\rangle | \Delta f = \{x\} \wedge \\ \Delta a = \mathrm{impl}(x) \backslash a\end{gathered} \tag{4.40}$$

$$\begin{gathered}\text{unselect}_{\Delta} : (\text{PM}, \text{F}) \rightarrow \text{PM}, \\ (\langle f, a\rangle, x) \mapsto \langle \Delta f, \Delta a\rangle | \Delta f = \{x\} \wedge \Delta a = \text{impl}(x) \setminus \\ \{e | \exists y \in f, y \neq x : e \in \text{impl}(y)\}\end{gathered} \tag{4.41}$$

Using these product deltas we can now easily obtain the rules that are required (or obsolete) after the change has been executed by applying the function from Eq. (4.26).

However, deriving the newly required (or obsolete) rules is not sufficient as the product change can lead to changed results for other rules. In order to derive the set of potentially affected rules, we first define the helper function, as shown in Eq. (4.42) that relates rules to the elements for which they were created (i.e., associations, feature constraints, or asset constraints).

$$\begin{gathered}\text{context} : (\text{PM} \rightarrow \{true, false\}) \rightarrow \text{FA} \cup \text{FC} \cup \text{AC}, \\ x \mapsto y \in \text{FA} \cup \text{FC} \cup AC : x = \text{genRule}(y)\end{gathered} \tag{4.42}$$

For finding the rules whose consistency results may have changed as consequence of a product change, we can use the function in Eq. (4.43).

$$\begin{gathered}\text{affected}_{\text{R}} : (\text{PM}, \text{PM}) \rightarrow \{\text{PM} \rightarrow \{true, false\}\}, \\ (\langle f, a\rangle, \langle \Delta f, \Delta a\rangle) \mapsto \{x | x \in \text{rules}(\langle f, a\rangle) \wedge \\ \exists y \in \Delta f \cup \Delta a : y \in \text{target}(\text{context}(x))\}\end{gathered} \tag{4.43}$$

This function returns all rules for which the context (i.e., the element for which they were created) is an element that contains an added or removed element in its targets. That is, the returned set includes all rules that need to be re-evaluated. The results of unaffected rules can be reused.

For checking the consistency status of an individual product for which a feature was selected or unselected, we can then use the functions Eq. (4.44) and Eq. (4.45) with the first parameter being the updated product and the second parameter being a product delta:

$$\begin{aligned} \text{check}_{add} : (\text{PM}, \text{PM}) &\rightarrow \{true, false\}, \\ (c, d) &\mapsto \text{result}(c, \text{rules}(c) \backslash \text{affected}_{\text{R}}(c, d)) \wedge \\ \text{result}(c, \text{affected}_{\text{R}}(c, d)) &\wedge \text{result}(c, rules(d)) \end{aligned} \tag{4.44}$$

$$\begin{aligned} \text{check}_{remove} : (\text{PM}, \text{PM}) &\rightarrow \{true, false\}, \\ (c, d) &\mapsto \text{result}(c, \text{rules}(c) \backslash (\text{affected}_{\text{R}}(c, d) \cup \text{rules}(d))) \wedge \\ &\text{result}(c, \text{affected}_{\text{R}}(c, d)) \end{aligned} \tag{4.45}$$

Note that we did split the formulas into sub-terms. The first part calculates the result for the unchanged rules, which allows reuse of previously calculated results. The second part re-validated the rules that already exist but may have a new result, and (in the case of the addition) the third part derives the newly required rules and validates them.

Let us illustrate this scenario with an example. If we added the feature `TV` to the product `P2`, the product delta would consist of `TV` and the asset for the high-definition video preparation statement in `Streamer.go()`. The set of affected rules would contain the excludes constraint from the already used low-definition video preparation statement. When checking the effect of the feature selection, the existing excludes-constraint for the low-definition video is violated because of the newly used high-definition video asset, which in turn excludes the low-definition video. In total, the new feature selection thus would lead to two violated constraints.

### *4.4.2 Changing Variability ($\Delta_F$ or $\Delta_A$)*

Let us now discuss evolution of either the feature of the asset model that changes the variability of the product line.

*Effects on asset model consistency.*

Adding or removing an asset constraint ($\Delta_\mathrm{A}$) may affect the consistency of the asset model Eq. (4.46) shows how the assets for which the consistency status may be affected are derived.

$$\begin{aligned}\mathrm{affected_A} : (\mathrm{AM}, \mathrm{AC}) &\to \mathcal{P}(A), \\ (\langle a, ac\rangle, x) &\mapsto \{e : e \in a \wedge \\ \mathit{source}(x) \in \mathit{getRequiredA}(e) &\cup \mathit{getExcludedA}(e)\} \\ &\cup \{\mathit{source}(x)\}\end{aligned} \tag{4.46}$$

The new consistency status after addition/removal of an asset constraint can then easily be computed using the following function:

$$\begin{aligned}\mathrm{valid_{AC}} &: (\mathrm{AM}, \mathrm{AC}) \to \{\mathit{true}, \mathit{false}\}, \\ (\langle a, ac\rangle, x) \mapsto \forall y \in a \backslash \mathrm{affected_A}&(\langle a, ac\rangle, x) : \mathrm{valid_A}(y) \wedge \\ \forall z \in \mathrm{affected_A}&(\langle a, ac\rangle, x) : \mathrm{valid_A}(z)\end{aligned} \tag{4.47}$$

The first part of the expression does not require rule re-validation but reuses existing results. Only the consistency results for affected assets have to be recomputed.

*Effects on feature model consistency.*

The effects of adding or removing associations or feature constraints to the feature model ($\Delta_\mathrm{F}$) are derived in way similar to the addition or removal of asset constraints and thus are omitted.

*Effects on product consistency.*

When a new element is added to either the feature associations ($\Delta_\mathrm{F}$), the feature constraints ($\Delta_\mathrm{F}$), or the asset constraints ($\Delta_\mathrm{A}$), the set of affected products contains all products that include the new element's source feature or asset, as defined in

Eq. (4.48).

$$\begin{aligned} \text{affected}_{\text{PV}} &: (\text{PL}, \text{FA} \cup \text{FC} \cup \text{AC}) \rightarrow \mathcal{P}(\text{PM}), \\ (\langle f, a, c \rangle, x) &\mapsto \{e | e \in c \ \wedge \ \text{source}(x) \in \text{selected}(e) \cup \text{uses}(e)\} \end{aligned} \tag{4.48}$$

The new rule can simply be generated by executing the function defined in Eq. (4.29), Eq. (4.31), or Eq. (4.33), respectively. For each affected product, only the new rule has to be validated. To determine the product's new consistency status, results of existing rules can be reused. The results of unaffected products can also be reused when determining the new consistency status of the product line. Eq. (4.49) shows the corresponding function.

$$\begin{aligned} \text{valid}_{add} &: (\text{PL}, \text{FA} \cup \text{FC} \cup \text{AC}), \\ (\langle f, a, c \rangle, x) \mapsto \text{valid}_{\text{PL}}(\langle f, a, c \backslash \text{affected}_{\text{PV}}(\langle f, a, c \rangle, x) \rangle) &\wedge \\ \forall y \in \text{affected}_{\text{PV}}(\langle f, a, c \rangle, x) : \text{result}(y, \text{rules}(y)) &\wedge \\ \text{result}(y, \text{genRule}(x)) & \end{aligned} \tag{4.49}$$

When removing a feature association or constraint, the set of affected products is computed in the same way as with an addition. However, in this case the rule associated with the removed element can simply be removed and the previously calculated results for the remaining rules can be used directly for determining the new consistency status of each product, as shown in Eq. (4.50).

$$\begin{aligned} \text{valid}_{remove} &: (\text{PL}, \text{FA} \cup \text{FC} \cup \text{AC}), \\ (\langle f, a, c \rangle, x) \mapsto \text{valid}_{\text{PL}}(\langle f, a, c \backslash \text{affected}_{\text{PV}}(\langle f, a, c \rangle, x) \rangle) &\wedge \\ \forall y \in \text{affected}_{\text{PV}}(\langle f, a, c \rangle, x) : \text{result}(y, \text{rules}(y) \backslash \{\text{genRule}(x)\}) & \end{aligned} \tag{4.50}$$

For both the addition and the removal of elements, the consistency results for unaffected products can be reused. Moreover, the results of existing rules can be reused for determining the new consistency status of affected products. In the case of an addition, the rule for the new element has to be generated and validated. In the case of a removal, no validation of rules is required at all.

As an example, let us consider the refactoring of the product line as illustrated in Section 4.1.1 where new constraints for the class `Player` and its method `play` are introduced (see Fig. 4.3) – an evolution of the asset model ($\Delta_\mathrm{A}$). Applying Eq. (4.48) returns the affected products `P2` and `P3` because both use the assets to which the new constraints are added. To check the validity of all products, the consistency status of the unaffected product `P1` can be reused. For `P2` and `P3`, the existing consistency results are combined with the results for the rules derived from the added constraints, which turn out to be inconsistent as both products do not use the newly required assets.

### *4.4.3 Changing Traces between Features and Assets ($\Delta_T$)*

Next, we have to consider the case of adding or removing the traces between an asset and a feature ($\Delta_\mathrm{T}$) in Fig. 4.1. For example, the assets for the class `Player` and its method `play` are added to the feature `play` after the refactoring of `Client.play(Video v)` in order to make products valid again.

For the addition/removal of an implementing asset to/from a feature, the set of affected products can be computed with the function defined in Eq. (4.51).

$$\begin{gathered}\mathrm{affected}_{\mathrm{PT}} : (\mathrm{PL}, \mathrm{F}, \mathrm{A}) \rightarrow \mathcal{P}(\mathrm{PM}),\\ (\langle f, a, c\rangle, x, y) \mapsto \{z | z \in c \wedge x \in \mathrm{selected}(z) \wedge \\ (\nexists v \in \mathrm{selected}(z) : v \neq x \ \wedge y \in \mathrm{impl}(v))\}\end{gathered} \tag{4.51}$$

The change affects those products which have the changed feature selected and do not contain other features that also use the added/removed asset. To check consistency, we reuse the results for the unaffected products and re-validate the affected products by adding or removing the rules for the asset, as shown in the functions defined in

Eq. (4.52) and Eq. (4.53).

$$\begin{aligned} \text{valid}_{add} &: (\text{PL}, \text{F}, \text{A}) \rightarrow \{true, false\}, \\ (\langle f, a, c\rangle, x, y) &\mapsto \text{valid}(\langle f, a, c \backslash \text{affected}_{\text{PT}}(\langle f, a, c\rangle, x, y)\rangle) \wedge \\ \forall z \in \text{affected}_{\text{PT}}(\langle f, a, c\rangle, x, y) &: \text{check}_{add}(z, \langle \varnothing, \{y\}\rangle) \end{aligned} \tag{4.52}$$

$$\begin{aligned} \text{valid}_{remove} &: (\text{PL}, \text{F}, \text{A}) \rightarrow \{true, false\}, \\ (\langle f, a, c\rangle, x, y) &\mapsto \text{valid}(\langle f, a, c \backslash \text{affected}_{\text{PT}}(\langle f, a, c\rangle, x, y)\rangle) \wedge \\ \forall z \in \text{affected}_{\text{PT}}(\langle f, a, c\rangle, x, y) &: \text{check}_{remove}(z, \langle \varnothing, \{y\}\rangle) \end{aligned} \tag{4.53}$$

When adding the assets newly introduced in the refactoring described in Section 4.1.1 to the feature `play`, the affected products according to Eq. (4.51) are `P2` and `P3`. To re-check the validity of all products, we use Eq. (4.52) where we reuse the validity status of `P1` and re-calculate the status for `P2` and `P3` with a product delta that contains the added asset. After adding the two new assets – one at a time – we find out that this addition fixed `P2`.

### *4.4.4 Adding/Removing Individual Assets to/from Products ($\Delta_P$)*

The presented functions Eq. (4.44) and Eq. (4.45) can also be used when adding or removing assets directly (i.e., not as part of a feature (un)selection) to or from a product. This action is typically done to quickly fix an invalid product without changing the product line definition. In this case, the involved assets and an empty set of features can be used as product delta to determine the effects of the change.

When adding the newly introduced assets from Section 4.1.1 to `P2`, the effect can be derived using a product delta with an empty feature set and the two added assets and the updated product as input for Eq. (4.44). Note that this is the same product delta already used above when we discussed the effects of adding the assets to a feature. Thus, the result is also the same and the action fixes the product `P2`.

### *4.4.5 Adding/Removing Assets or Features ($\Delta_A$ or $\Delta_F$)*

Of course it is also common during evolution of the product line to add and remove features and assets.

*Asset Addition ($\Delta_A$):* When an asset is added to the asset model, products are not affected until it is linked to other assets through an asset constraint, as discussed above. It is sufficient to check the validity of the added asset using Eq. (4.21).

*Asset Removal ($\Delta_A$):* When an asset is removed, this means that the asset is removed from all products, affecting consistency as discussed in Section 4.4.4. Additionally, the asset is removed from all the features it implements and the corresponding asset constraints are removed. Note that these two actions may affect only asset model consistency as discussed in Section 4.4.2 because the assets where already removed from all products.

*Feature Addition ($\Delta_F$):* Adding a feature does not affect products as a new feature that is not linked to other features cannot be required or excluded by other features. When the feature is linked to others through an association or constraint, the affects have been discussed in Section 4.4.2.

*Feature Removal ($\Delta_F$):* The effect of removing a feature from the product line is equal to an unselection of the feature in all products, which we already discussed. After unselecting the feature, the association to the feature is removed from the product line definition, a step that was also discussed above. At this point, the new consistency status for all affected products is derived and the following removal of links between the removed feature and the implementing assets is not affecting any products.

## 4.5 Evaluation and Analysis

To validate the feasibility of our approach, we developed a prototype implementation and employed it for checking a large-scale benchmark software product line.[2]

### *4.5.1 Implementation*

We implemented the formal product line framework presented in Section 4.2 in Java using *Eclipse Modeling Framework (EMF)* [86]. For checking consistency, we also implemented an incremental consistency checker as well as an incremental rule management mechanism based on the formalisms presented in Sections 4.3 and 4.4. The implementation does not employ traditional model checking solutions such as SAT- or CSP-solvers but relies on straightforward implementations of the presented functions using typical object-oriented techniques. Although we did not formally proof the correctness of the implementation, the correct behavior was thoroughly tested with unit-tests for the evolution scenarios presented in this chapter.

### *4.5.2 Case Study – Crash Management System*

To assess the applicability of our approach to product lines of realistic size and complexity, we tested our prototype with one of the few large-scale product lines publicly available with design models: the *Barbados Crash Management System Product Line* [104], available in ReMoDD [105]. The product line defines a crash management software system that is modeled using UML diagrams (class- and sequence diagrams, state machines) and traditional object oriented techniques.

The feature model consists of 66 features and has a total of 35 associations along with 2 cross-tree feature constraints. The analysis of the feature model with the FAMA tool [115] revealed a total of 440,640 valid configurations. The models used in our tests contained 4,236 model elements in total.

[2] The prototype is available online at http://www.sea.jku.at/tools/sc

### *4.5.3 Experimental Setup*

We converted the feature model as well as the software design models into a product line according to the definitions presented in Section 4.2.

*Asset extraction.*

For each class, we generated a representing asset. Moreover, we generated an asset for each property or operation defined in the class. Property or operation assets require the corresponding class asset. Associations were also mapped to assets that require the assets of the linked classes. Each sequence diagram of the reference variant was directly translated to an asset. Each sequence diagram fragment related to features other than the root was also modeled as an individual asset that requires the sequence diagram asset when used. Each asset generated for sequence diagrams – or fragments thereof – also requires the assets associated with the method calls modeled in the diagram or fragment, respectively. In total, 921 assets were extracted from the available models. Between those assets, 1360 asset constraints were generated based on dependencies in the underlying models.

*Product construction.*

We generated products by selecting all common features (i.e., features that must be selected in any product) and added more (non-mandatory) features randomly, meaning that products that may violate constraints based on both the feature and the asset model. All tests were performed with sets of 1, 10, 50, 100, 250, 500, 750, and 1,000 concurrently loaded products to test scalability. Each set contained the fully configured product that included all available features and assets; this product required 1398 rules to be enforced.

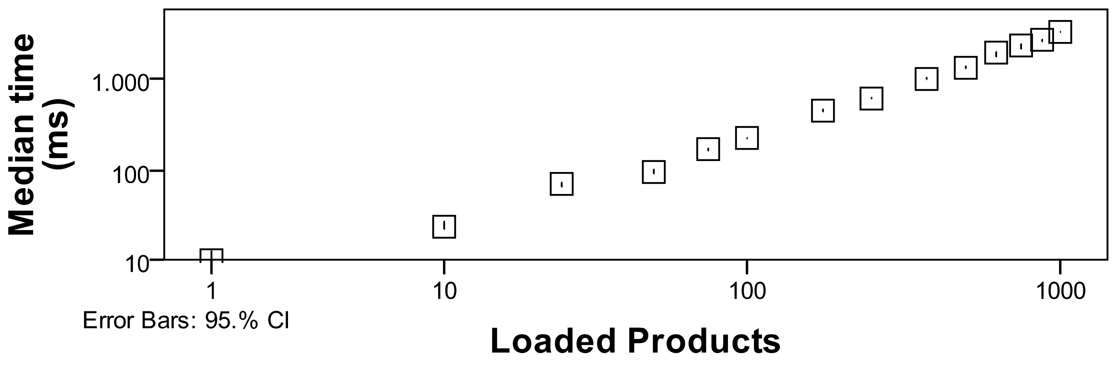


FIGURE 4.4: Median times for initializing required rules

### *4.5.4 Initializing Rules*

Our approach requires rules to be initialized on a concrete set of products. Thus, to be practical, this initialization process should be performed in a reasonable time even for large numbers of products (i.e., within a few seconds). Hence, we also assessed the time needed for initializing (i.e., generating and validating) rules for different sets of products evaluated. The initialization time grows linearly with the number of products and rules, as depicted in Fig. 4.4; for 1,000 products and a total of 701,867 rule validations, initialization takes less than 3,500 ms.

### *4.5.5 Performing Changes*

The performed changes are based on the scenarios discussed in Section 4.4. For each test, we measured the overall time required: a) for performing the evolution, and b) for updating and revalidating affected products and rules. The benchmarks were run on an Intel Core i5-650 machine with 8GB of memory running Windows 7 Professional. They were executed 100 times and median times were used for our statistics.

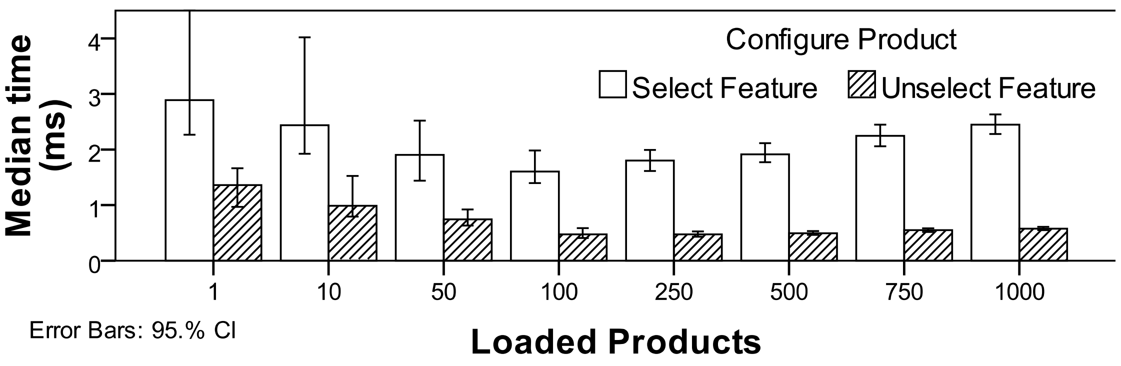


FIGURE 4.5: Median times for (un)selecting feature in product ($\Delta_P$)

*Product configuration ($\Delta_P$).*

First, we evaluated the efficiency of changing single product configurations. Features of a product were unselected and selected again. This test was executed for every product and all its selected features. However, for the statistics we only considered (un)selections that actually affected rules and changed the product's used assets. The results, depicted in Fig. 4.5, show that the median time for selecting a feature in a product are below 3 ms, the median times for unselecting a feature are even lower at about 1 ms as this does not require the application of new rules. Note that the median processing time for changes that affected more than 250 rules was 15.2 ms and the maximum required total processing time measured for a feature selection was 99.0 ms. Moreover, the number of loaded products does not affect the required processing time significantly.

*Association or constraint change ($\Delta_F$ and $\Delta_A$).*

Next, we evaluated the performance when the feature or asset model changes. First, each existing association, feature constraint, and asset constraint was removed from the product line and added again – with different numbers of products loaded at the time of evolution. We synthesized the results for changes of FA, FC, and AC because they trigger the same updates (as discussed in Section 4.4.2). As shown

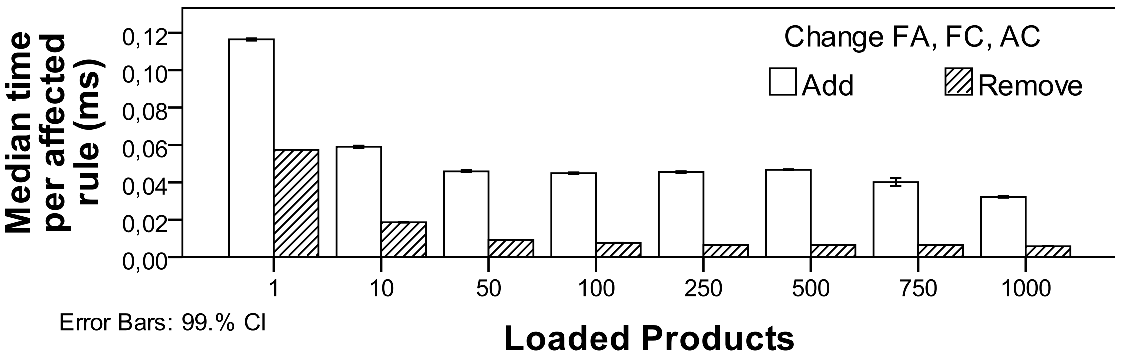


FIGURE 4.6: Median times for changing variability ($\Delta_{\mathrm{F}}$ and $\Delta_{\mathrm{A}}$) per affected rule

in Fig. 4.6, the median times per affected rule are nearly stable for the addition of elements between 10 and 1,000 loaded models at 0.045 ms. However, there is a slight decreases with increasing numbers of loaded products. We suspect that this is caused by the one-time transformation costs that are required for generating or deleting rules. Since the costs of finding and checking affected products should increase linearly with the number of loaded products, those one-time costs reduce the average costs per affected rule. The faster processing of element removal is caused by the fact that this case does not require re-validation of rules or the addition of rules to products. Even when considering only element additions that affected more than 500 products the median total processing time was 16.4 ms and the median time per affected rule was 0.032 ms. The maximum value observed for the total processing time was 131.0 ms.

*Feature implementation change ($\Delta_T$).*

For the third scenario, we removed and added implementing assets to/from features. For each feature, we removed and added implementing assets. In Fig. 4.7, the observed median time per affected rule for adding and removing implementing assets to/from features are depicted. Note that the median time for addition is nearly stable at 0.06 ms between 250 and 1,000 loaded models, which shows the high efficiency

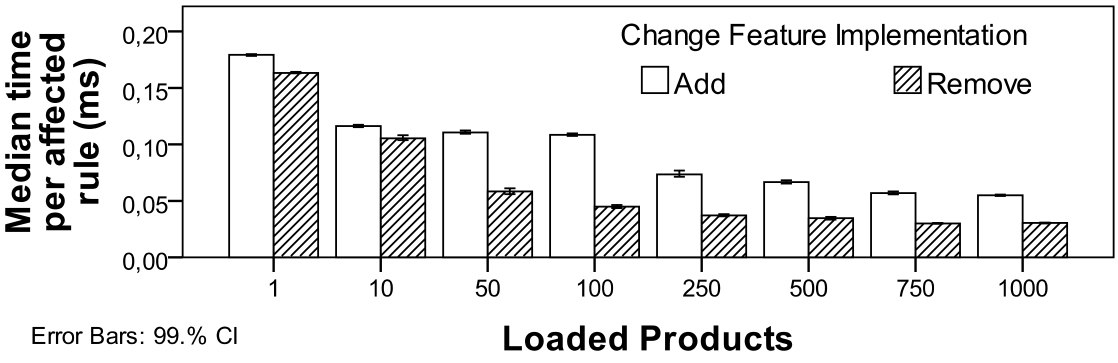


FIGURE 4.7: Median times for changing feature implementing assets ($\Delta_T$) per affected rule

of our approach. As before, the implementation overhead becomes less important with growing numbers of loaded products and thus affected products and affected rules. Analyzing only additions that affected more than 500 products, the median total processing time is still 39.77 ms with a median time per affected rule of 0.05 ms. The maximum total processing time observed was 437.9 ms for a change that affected 11,736 rules. The removal of implementing assets is always faster than the addition.

### *4.5.6 Comparison to SAT-based Checking*

To asses the performance benefits of our incremental safe configuration approach, we compared it to a more traditional SAT-based checking approach. For the comparison, we checked the validity of each loaded product individually, using CNF-clauses equivalent to the rules applied to the product with safe configuration. The total time required for the translation to CNF and satisfiability-checking was 13 ms for a single product and reached a stable median value of about 1 ms per product when multiple products were checked. Thus, processing time increased linearly with number of loaded products as no incremental technique was used to determine which products where actually affected by a change.

In contrast, processing time with safe configuration does not depend on the number of products but only on the number of affected rules. When using safe configuration for checking changes that affect 500 or more products (i.e., in more than 500 products at least one rule was affected by the change), the median processing time is reduced by 96% for association or constraint changes and by 92% for feature implementation changes compared to SAT-based checking. We did not observe a single case in which the SAT-based approach was faster than safe configuration.

Note that using a SAT-solver in combination with the incremental updating of applied rules is also possible. In this case, the actual validation of a product is done by a SAT-solver and the clauses it uses are updated incrementally based on the theories presented above. However, this combination – though significantly faster than non-incremental checking with SAT – in our tests was still slower than the prototype implementation.

### *4.5.7 Alternative Implementation*

The results discussed above were achieved with a specialized rule management system and a consistency checker that were developed specifically for the incremental checking of SPLs. To highlight that the concepts presented in this chapter are an application of the principles presented earlier in Chapter 2, however, we also implemented incremental rule management for SPLs using model representations of SPLs and products and employing the generic transformation engine and consistency checker we already used there. To do this, we had to translate the functions for constructing rules into transformation rules.

#### *Transformation rules.*

Sample transformation rules that generate typical product constraints from a feature model are shown in Listing 4.1.[3] We focused on commonly accepted product constraints as described, for example, in [112]. The required product constraints were

[3] We use the term *product constraints* now to indicate that we generate restrictions applied to EMF-based model elements.

generated by seven transformation rules.

The first transformation rule, shown in Listing 4.1 (lines 1–7), is executed for all mandatory associations. Such mandatory associations mean that every target feature must be selected in a product if the source feature is selected. The second transformation rule we defined, shown in Listing 4.1 (lines 9–15), is used for or-associations that require at least one of the target features to be selected in a product if the source is selected. The third transformation rule for xor-associations, shown in Listing 4.1 (lines 17–23), is similar to the second one. Note that for xor-associations, exactly one instead of at least one target feature must be selected if the source is selected. The fourth and the fifth transformation rule are used to handle requires and excludes cross-tree constraints, as shown in Listing 4.1 (lines 25–31) and Listing 4.1 (lines 33–39), respectively. These two transformation rules are very similar to *tMandatory* (Listing 4.1, lines 1–7) and *tXor* (Listing 4.1, lines 17–23) relations. For *tRequires*, only the source element type has was changed to `Requires`. For *tExcludes*, the source element type was changed to `Excludes` and the size of the result must be equal to 0.

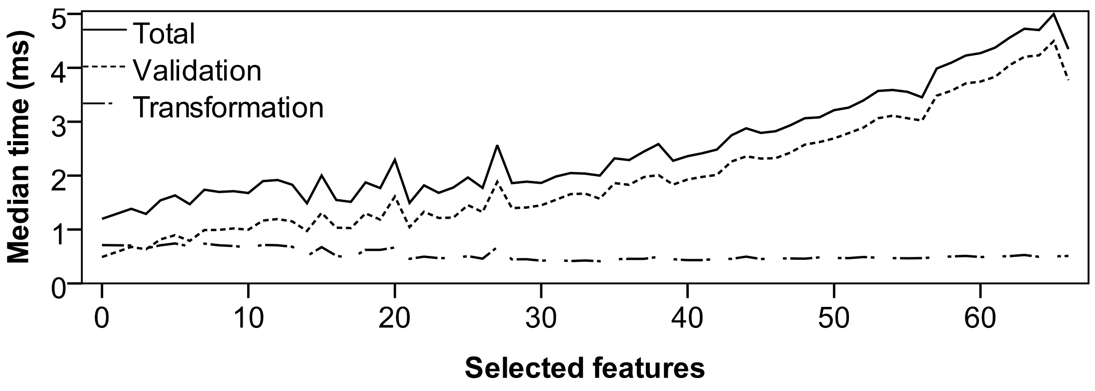
The sixth transformation rule, shown in Listing 4.1 (lines 41–48), generates product constraints for the generic product invariant that a feature (that is not the root feature) may only be selected in a product if its parent feature is also selected. A guarding statement is used so that it is only executed for features that actually have a parent (i.e., all features but the root feature). The seventh transformation rule, shown in Listing 4.1 (lines 50–56), generates a product constraint that requires the root feature of the feature model to be selected in a product.

*Performance comparison.*

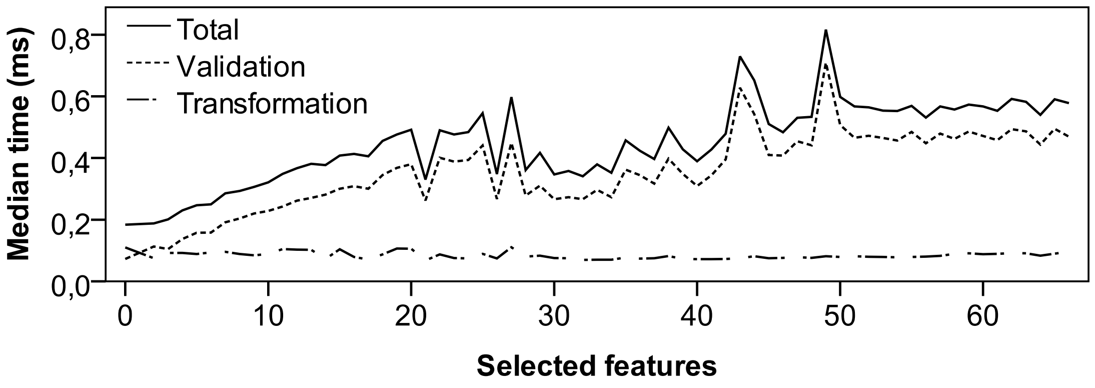
In order to compare the efficiency of the specialized implementation discussed above and the alternative implementation that makes use of more generic technologies, we

```
rule tMandatory
  from
    s : FeatureModel!Mandatory
  to
    t : ConstraintModel!Constraint (
      context <- Product,
      inv <- "self.features->includes("+s.sourceFeature+") implies self.features->
          includesAll("+s.targetFeatures+")")

rule tOr
  from
    s : FeatureModel!Or
  to
    t : ConstraintModel!Constraint (
      context <- Product,
      inv <- "self.features->includes("+s.sourceFeature+") implies self.features->
          select(x|"+s.targetFeatures+"->includes(x))->size()>0")

rule tXor
  from
    s : FeatureModel!Xor
  to
    t : ConstraintModel!Constraint (
      context <- Product,
      inv <- "self.features->includes("+s.sourceFeature+") implies self.features->
          select(x|"+s.targetFeatures+"->includes(x))->size()=1")

rule tRequires
  from
    s : FeatureModel!Requires
  to
    t : ConstraintModel!Constraint (
      context <- Product,
      inv <- "self.features->includes("+s.sourceFeature+") implies self.features->
          includesAll("+s.targetFeatures+")")

rule tExcludes
  from
    s : FeatureModel!Excludes
  to
    t : ConstraintModel!Constraint (
      context <- Product,
      inv <- "self.features->includes("+s.sourceFeature+") implies self.features->
          select(x|"+s.targetFeatures+"->includes(x))->size()=0")

rule tParent
  from
    s : FeatureModel!Feature (
      s.parent<>null)
  to
    t : ConstraintModel!Constraint (
      context <- Product,
      inv <- "self.features->includes("+s+") implies self.features->includes("+s.
          parent+")")

rule tRoot
  from
    s : FeatureModel!FeatureModel
  to
    t : ConstraintModel!Constraint (
      context <- Product,
      inv <- "self.features->includes("+s.root+")")
```

Listing 4.1: Transformation rules for product lines



(a) Processing times for addition



(b) Processing times for removal

FIGURE 4.8: Processing times for changing variability ($\Delta_F$ and $\Delta_A$) of single product

conducted a test similar to the one discussed in Section 4.5.5, in which we performed changes in the variability of the SPL (i.e., we removed and added each association and feature constraint). For this test, we used the traditional transformation rules and observed the processing times required when only a single product was loaded. By using the transformation engine and the consistency checker employed already in Chapter 2, it was possible to distinguish between transformation and validation performance. For the test, we used products that were composed of randomly selected features. The used products included the empty product (i.e., zero features selected) and the fully configured product (i.e., all 66 features selected). The results for this test are shown in Fig. 4.8. For the addition of a new association or feature constraint, Fig. 4.8(a) shows that the total time for processing in all test executions stayed below

5 ms. Note that the time required for the transformation (i.e., the creation of the required rule) remains nearly unchanged at under 1 ms. However, the time for validation increases linearly with the number of features selected in the product. This is caused by the different nature of applied rules. In the previous tests, the rules were validated only if they were actually relevant for a product. In this test, on the other hand, each rule was always applied to the product and the result of the validation was set to consistent if the rule was actually not applicable (note the "implies" construct used in Listing 4.1). Although this overhead increased the median overall processing time for a change from about 0.12 ms (see Fig. 4.6) to a value between 1 ms and 5 ms depending on product configuration, the processing is still fast and can be done live in a tool.[4]

For the removal of associations or cross-tree constraints from the feature model, Fig. 4.8(b) shows that the total processing time stays under 1 ms for all performed changes. Again, note the constant time of about 0.1 ms required for the transformation (i.e., removal of the existing rules that become obsolete after the change). Because there is no actual validation required for the removal of rules, the slope of the line indicating the validation is less than for the addition. However, the slope is still positive because the consistency checker must discard data that was captured during previous validation of the removed rules and the amount of that data correlates with the number of selected features.

Overall, the test results demonstrate that the employed transformation and consistency checking technologies can affect performance significantly. However, observed results also increase confidence in the general applicability of the principles presented in Chapter 2.

[4] Note that this is a valid comparison because the total processing times shown in Fig. 4.6 for a single loaded product are based solely on changes that affected only a single rule.

### *4.5.8 Threats to Validity*

Though the validation results are quite promising, there are possible threats to validity we have to discuss. First, our case study is based on a single product line only. While this means that our results may not be representative for all product lines used in practice, the results still indicate that our approach performs well at least with product lines of similar characteristics. The SPL used for the case study was developed by experts in the field of MDE and is commonly used as a benchmark SPL. Moreover, it provides similar characteristics to other product lines typically used for validation purposes from the *Software Product Lines Online Tools (SPLOT)* repository [116]. The feature models available in SPLOT have an average size of 25 features, whereas our case study SPL includes 66 features. Additionally, our sample SPL (including the asset model) is rather complex with a constraints-to-model-elements ratio of 1.38 (i.e., for each feature or asset, there are 1.38 constraints present). Therefore, we are confident that the case study results are representative for a large number of industrial product lines.

Another aspect of our case study that might be challenged is the way we extracted assets and asset constraints. Indeed, it would have been possible to increase the number of assets by loading not only top-level model elements (e.g., classes) but also generating assets for low-level information (e.g., generate an asset for each reference cardinality). However, we chose to rely on top-level elements for asset generation because those elements are typically linked to other model elements whereas low-level model elements are not referenced by elements other than their parent top-level element. Note that by including additional assets and asset constraints for low-level model elements, the total number of constraints would increase significantly. However, because of the incremental nature of our approach, such an increase in asset model size would have only a minimal impact on processing times compared to other approaches that rely on batch validation and where each additional asset constraint increases validation effort inevitably.

## 4.6 Related Work

Product lines and consistency checking are active fields of research and various approaches for checking the consistency of product lines have been proposed. In this section we discuss those closest to our work.

One of the approaches most relevant to ours is *Safe Composition* [98]. Safe composition would typically infer all possible consistency rules and check whether all products that can be configured for a given feature model are valid with respect to the asset model (i.e., that all products are working) [117]. This means that the approach cannot be used when configured products are intentionally not conforming to the feature model because of unexpected customer requirements. Furthermore, safe composition is not designed for evolving product lines and thus not supports incremental updates of consistency results after evolution. The strength of our approach is indeed the flexibility it gives product line designers when evolving the product line or configuring products. Moreover, there are known issues with safe composition as the number of possible products can grow exponentially [118]. Our approach avoids these problems by considering only existing products. While this does not guarantee safety for all possible products, it reduces scalability issues and enables checking of products that are not valid with respect to the feature model.

One of biggest differences between safe configuration and approaches such as [102, 103, 98, 119, 120, 111, 121] is that it does not use traditional model-checking techniques such as SAT-solvers or propositional formulas for finding inconsistencies but determines consistency results for individual products by applying specific sets of rules incrementally.

Shi et al. [119] use feature models and implementation information to build a feature dependency graph and a control flow graph. Those are combined to an interaction tree which is used as basis for reasoning about product line consistency. Classen et al. [121] use feature transition systems and computation tree logic for finding inconsistencies through symbolic model checking. Cordy et al. [120] investigate the effects of product line evolution. Based on feature transition systems, different categories of features, and behavioral analysis they determine which products have to be model checked after feature model evolution. Our approach uses a unified mechanism for determining effects of changes regardless of the kind of feature, asset, association, or constraint involved. Sabouri et al. [122] proposed an approach for handling the evolution of feature models. They use state-spaces for possible products and support the reuse of consistency results for the existing state space when checking the consistency of the evolved product line using a *Spin* model checker. The mentioned approaches rely on model checkers to find inconsistencies in affected products after evolution. Safe configuration, on the other hand, changes the set of rules that are enforced on individual products incrementally as the product line evolves, allowing for quick and efficient validation of only affected products without using heavyweight model checking techniques. However, our work focuses on checking strucutural properties while the approaches above focus on behavioral properties, thus we do not replace model checking for product lines but present an efficient alternative for checking their structural integrity (e.g., for checking whether a configured product does compile).

State machines and product deltas are also used by Lochau et al. [123] for managing test suites for product line regression testing. They use these deltas to determine which test cases become required or obsolete as the consequence of evolution. In safe configuration, such test cases can be modeled as assets and can be linked to features and other assets which they check, thus our approach can also be used for this purpose.

## 4.7 Summary

In this chapter, we presented the formal foundation for safe configuration, an incremental approach for checking consistency of evolving product lines that is based on the concepts of CDM. The approach is generic and handles evolution of the feature model, the asset model, and of individual products. This provides more flexibility to product line designers as the approach validates arbitrary configured products. We demonstrated its feasibility by developing a prototype implementation and showed its efficiency and scalability in realistic evolution scenarios of a large-scale SPL. The validation results show that the approach scales and performs well even when a large number of products is affected by evolution; the worst time for a single evolution was under 500 ms for a change that affected 1,000 products and required the generation and validation of 11,736 rules.

The approach of safe configuration illustrates that automatic consistency rule generation and management is a concept applicable to SPLs.

# 5 Discussion and Conclusion

In the previous chapters, we have introduced the concepts of constraint generation and illustrated how the approach can address various issues in software engineering. In this chapter, we discuss how the data presented relates to the research questions defined in Chapter 1. We also briefly discuss possible future work.

## 5.1 RQ1: Feasibility

The feasibility of incremental constraint generation and management has been discussed both theoretically and practically in Chapter 2. In that chapter, we have illustrated how existing technology can be employed to realize an incremental constraint management framework. Using typical MDE-scenarios, we demonstrated by example that knowledge available from existing models is sufficient for generating meaningful model constraints that enable further consistency checking and inconsistency fixing technologies. In Chapter 3, we applied the principles of CDM to metamodeling and generated important structural constraints that were managed automatically during metamodel evolution – another example where existing knowledge was used as primary basis for generating constraints. Overall, both chapters clearly demonstrate that the proposed solution of incremental constraint management is feasible.

## 5.2 RQ2: Incrementality and Performance

The performance of our approach was demonstrated in different realistic case studies. For typical MDE-scenarios, the case study presented in Chapter 2 showed that the incremental constraint-generating transformation can be performed within milliseconds. Moreover, the case study revealed that there are consistency checkers already available that perform the validation of updated constraints also within milliseconds. Since the total processing time for both transformation and validation stayed clearly below 1,000 ms for all performed changes, we concluded that CDM in general can be applied without interrupting modeling tool users.

The case study conducted in Chapter 3 supports this conclucions. The results in that case study showed that both the transformation and the validation time per validated model element were nearly constant for different model sizes. Moreover, the total processing time did not exceed 1,000 ms for more than 99.9% of the metamodel changes. Therefore, we conclude that our approach scales and does not interrupt tool users even for large change impacts.

Overall, the case study results provide evidence that in realistic scenarios our approach scales and does not impose serious performance penalties.

## 5.3 RQ3: Different MDE Issues Addressed

In Chapter 2 we discussed that CDM can be used to effectively address issues that arise with the use of model transformations in some specific scenarios that involve uncertainties and in which multiple possible solutions are possible for a modeling problem. In particular, we have used constraint management to address the MDE issues of: i) ambiguity in transformations, ii) rule scheduling, iii) model merging, iv) concurrent modifications in synchronized models.

In Chapter 3, we showed how the approach addresses the issue of evolving languages and metamodels and co-evolving model constraints that are based on them.

In addition to those typical MDE issues, we have also applied CDM in the domain of software product lines in Chapter 4. Specifically, we showed how incremental rule generation can be used to realize instant consistency checking of existing products when product variability is changed.

## 5.4 RQ4: General Applicability

Note that we not only applied CDM to three rather different domains in total (i.e., model transformations, metamodeling, and SPLs), but that we also used three different approaches for realizing the core concepts (i.e., traditional model transformations, constraint templates, and mathematic functions). This demonstrates the general applicability and shows that the concepts are generic enough to be applied in different ways.

## 5.5 Future Work

Though the research presented in this thesis clearly showed that constraint generation is feasible and useful in various domains, we believe there are still many other domains to which our approach can be applied. For instance, CDM may be used to determine constraints imposed on self-adaptive systems. By using sophisticated fixing technologies, systems may then be reconfigured based on detected inconsistencies. We plan to further investigate the usability of CDM and to integrate such approaches for automatic inconsistency fixing in our prototypes. Moreover, we plan to do further research on finding contradictory constraints – or even prevent that such constraints can be generated – and the effects of CDM when used with bidirectional transformations. For our templates, we want to add support for partial re-instantation to increase efficiency even further. Regarding incremental safe configuration, we want to expand our prototype to support more specialized product line concepts that include, for example, feature groups.

## 5.6 Conclusion

In this thesis, we have outlined the possibilities of incremental constraint generation in different software engineering domains – namely in model transformation, metamodeling, and software product lines. We have discussed the various MDE issues such an approach can address and provided different implementations that were employed for conducting realistic case studies, proving the feasibility of the approach. In general, our approach provides additional guidance for modelers by enforcing not only generic but also automatically managed, domain-specific constraints that detect an increased variety of inconsistencies during all phases of MDE-oriented software processes and for arbitrary models. The different case studies not only demonstrated general applicability, but the obtained performance results also suggested that our approach scales in real-world scenarios. The approach was thoroughly compared to related technologies and we found that it should be seen as complementary approach that is capable of bridging gaps between existing MDE approaches. For future work, we have identified several fields to which incremental constraint generation and management can be applied.

# Bibliography

[1] D. C. Schmidt, “Guest editor’s introduction: Model-driven engineering,” *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] R. B. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *FOSE*, pp. 37–54, 2007.

[3] B. Hailpern and P. L. Tarr, “Model-driven development: The good, the bad, and the ugly,” *IBM Systems Journal*, vol. 45, no. 3, pp. 451–462, 2006.

[4] P. Mohagheghi and V. Dehlen, “Where is the proof? - a review of experiences from applying mde in industry,” in *ECMDA-FA*, pp. 432–443, 2008.

[5] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, “Empirical assessment of mde in industry,” in *ICSE*, pp. 471–480, 2011.

[6] D. Shirtz, M. Kazakov, and Y. Shaham-Gafni, “Adopting model driven development in a large financial organization,” in *ECMDA-FA*, pp. 172–183, 2007.

[7] E. J. Chikofsky and J. H. C. II, “Reverse engineering and design recovery: A taxonomy,” *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[8] B. Dobing and J. Parsons, “How UML is used,” *Commun. ACM*, vol. 49, no. 5, pp. 109–113, 2006.

[9] C. Atkinson and T. Kühne, “The essence of multilevel metamodeling,” in *UML*, pp. 19–33, 2001.

[10] E. K. Jackson and J. Sztipanovits, “Towards a formal foundation for domain specific modeling languages,” in *EMSOFT*, pp. 53–62, 2006.

[11] S. Oppl and C. Stary, “Tabletop concept mapping,” in *Tangible and Embedded Interaction*, pp. 275–282, 2009.

[12] IBM, “Rational Software Architect.” http://www-01.ibm.com/software/rational/products/swarchitect/.

[13] ArgoUML. http://argouml.tigris.org/.

[14] Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró, “Quick fix generation for DSMLs,” in *VL/HCC*, pp. 17–24, 2011.

[15] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, “Constraint-driven modeling through transformation,” in *ICMT*, pp. 248–263, 2012.

[16] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, “Constraint-driven modeling through transformation,” *Software and System Modeling*, 2013. DOI: 10.1007/s10270-013-0363-3.

[17] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, “Supporting the co-evolution of metamodels and constraints through incremental constraint management,” in *MoDELS*, 2013. Accepted for publication.

[18] S. Sendall and W. Kozaczynski, “Model transformation: The heart and soul of model-driven software development,” *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[19] K. Czarnecki and S. Helsen, “Feature-based survey of model transformation approaches,” *IBM Systems Journal*, vol. 45, no. 3, pp. 621–646, 2006.

[20] T. Mens and P. V. Gorp, “A taxonomy of model transformation,” *Electr. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, 2006.

[21] J. Etzlstorfer, A. Kusel, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, and M. Wimmer, “A survey on incremental model transformation approaches,” in *MODELS-ME*, 2013.

[22] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool,” *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008.

[23] Object Management Group, “Query/View/Transformation (QVT).” http://www.omg.org/spec/QVT/.

[24] A. Schürr, “Specification of graph translators with triple graph grammars,” in *WG*, pp. 151–163, 1994.

[25] D. D. Ruscio, R. Eramo, and A. Pierantonio, "Model transformations," in *SFM*, pp. 91–136, 2012.

[26] P. Stevens, "Bidirectional model transformations in QVT: semantic issues and open questions," *Software and System Modeling*, vol. 9, no. 1, pp. 7–20, 2010.

[27] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *IWPSE*, pp. 13–22, 2005.

[28] P. Stevens, "A landscape of bidirectional model transformations," in *GTTSE*, pp. 408–424, 2007.

[29] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider, "Flexible and scalable consistency checking on product line variability models," in *ASE*, pp. 63–72, 2010.

[30] M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires, "Performance in model transformations: Experiments with ATL and QVT," in *ICMT*, pp. 198–212, 2011.

[31] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software and System Modeling*, vol. 8, no. 1, pp. 21–43, 2009.

[32] F. Jouault and M. Tisi, "Towards incremental execution of ATL transformations," in *ICMT*, pp. 123–137, 2010.

[33] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, and H. Mei, "Instant and incremental QVT transformation for runtime models," in *MoDELS*, pp. 273–288, 2011.

[34] D. Hearnden, M. Lawley, and K. Raymond, "Incremental model transformation for the evolution of model-driven systems," in *MoDELS*, pp. 321–335, 2006.

[35] S. Johann and A. Egyed, "Instant and incremental transformation of models," in *ASE*, pp. 362–365, 2004.

[36] M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel, "On using inplace transformations for model co-evolution," in *MtATL*, INRIA & Ecole des Mines de Nantes, 2010.

[37] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *EDOC*, pp. 222–231, 2008.

[38] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *ICMT*, pp. 260–283, 2009.

[39] Object Management Group, "Object Constraint Language (OCL)." http://www.omg.org/spec/OCL/.

[40] J. Barwise, "An introduction to first-order logic," in *HANDBOOK OF MATHEMATICAL LOGIC* (J. Barwise, ed.), vol. 90 of *Studies in Logic and the Foundations of Mathematics*, pp. 5 – 46, Elsevier, 1977.

[41] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *ESEC/SIGSOFT FSE*, pp. 315–324, 2009.

[42] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, pp. 220–229, 2012.

[43] A. Reder and A. Egyed, "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *ASE*, pp. 347–348, 2010.

[44] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.

[45] Object Management Group, "Unified Modeling Language (UML)." http://www.uml.org/.

[46] Object Management Group, "Unified Modeling Language (UML) superstructure." http://www.omg.org/spec/UML/2.4.1/Superstructure, 2012.

[47] Z. Micskei and H. Waeselynck, "The many meanings of uml 2 sequence diagrams: a survey," *Software and System Modeling*, vol. 10, no. 4, pp. 489–514, 2011.

[48] I. Groher, A. Reder, and A. Egyed, "Incremental consistency checking of dynamic constraints," in *FASE*, pp. 203–217, 2010.

[49] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE*, pp. 455–464, 2003.

[50] F. Büttner, M. Egea, J. Cabot, and M. Gogolla, "Verification of ATL transformations using transformation models and model finders," in *ICFEM*, pp. 198–213, 2012.

[51] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.

[52] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer, "Flexible consistency checking," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 1, pp. 28–63, 2003.

[53] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards automated inconsistency handling in design models," in *CAiSE*, pp. 348–362, 2010.

[54] A. Reder and A. Egyed, "Incremental consistency checking for complex design rules and larger model changes," in *MoDELS*, pp. 202–218, 2012.

[55] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Software and System Modeling*, vol. 9, no. 1, pp. 69–86, 2010.

[56] S. M. A. Shah, K. Anastasakis, and B. Bordbar, "From UML to Alloy and back again," in *MoDELS Workshops*, pp. 158–171, 2009.

[57] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *MoDELS*, pp. 415–431, 2012.

[58] A. Nöhrer and A. Egyed, "C2o configurator: a tool for guided decision-making," *Autom. Softw. Eng.*, vol. 20, no. 2, pp. 265–296, 2013.

[59] M. Kessentini, H. A. Sahraoui, M. Boukadoum, and O. B. Omar, "Search-based model transformation by example," *Software and System Modeling*, vol. 11, no. 2, pp. 209–226, 2012.

[60] A. Nöhrer, A. Reder, and A. Egyed, "Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track," in *ICSE*, pp. 864–867, 2011.

[61] J. P. Puissant, R. V. D. Straeten, and T. Mens, "Badger: A regression planner to resolve design model inconsistencies," in *ECMFA*, pp. 146–161, 2012.

[62] E.-J. Manders, G. Biswas, N. Mahadevan, and G. Karsai, "Component-oriented modeling of hybrid dynamic systems using the generic modeling environment," in *MBD/MOMPES*, pp. 159–168, 2006.

[63] H. Ossher, R. K. E. Bellamy, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, M. Desmond, J. de Vries, A. Fisher, and S. Krasikov, "Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges," in *OOPSLA*, pp. 848–864, 2010.

[64] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Cross-layer modeler: A tool for flexible multilevel modeling with consistency checking," in *ESEC/SIGSOFT FSE*, pp. 452–455, 2011.

[65] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[66] T. J. Schaefer, "The complexity of satisfiability problems," in *STOC*, pp. 216–226, 1978.

[67] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*, pp. 632–647, 2007.

[68] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *SFM*, pp. 399–437, 2012.

[69] F. Büttner, H. Bauerdick, and M. Gogolla, "Towards transformation of integrity constraints and database states," in *DEXA Workshops*, pp. 823–828, 2005.

[70] M. Giese and D. Larsson, "Simplifying transformations of OCL constraints," in *MoDELS*, pp. 309–323, 2005.

[71] I. S. Bajwa and M. G. Lee, "Transformation rules for translating business rules to OCL constraints," in *ECMFA*, pp. 132–143, 2011.

[72] Object Management Group, "Semantics of Business Vocabulary and Rules (SBVR)." http://www.omg.org/spec/SBVR/.

[73] Y. Xiong, H. Song, Z. Hu, and M. Takeichi, "Supporting parallel updates with bidirectional model transformations," in *ICMT*, pp. 213–228, 2009.

[74] I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano, "Toward bidirectionalization of ATL with GRoundTram," in *ICMT*, pp. 138–151, 2011.

[75] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "JTL: A bidirectional and change propagating transformation language," in *SLE*, pp. 183–202, 2010.

[76] M. Tisi, S. M. Perez, F. Jouault, and J. Cabot, "Lazy execution of model-to-model transformations," in *MoDELS*, pp. 32–46, 2011.

[77] T. Saxena and G. Karsai, "MDE-based approach for generalizing design space exploration," in *MoDELS*, pp. 46–60, 2010.

[78] Á. Horváth and D. Varró, "Dynamic constraint satisfaction problems over models," *Software and System Modeling*, vol. 11, no. 3, pp. 385–408, 2012.

[79] A. Queralt and E. Teniente, "Verification and validation of UML conceptual schemas with OCL constraints," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 2, pp. 13:1–13:41, 2012.

[80] S. Zschaler, D. S. Kolovos, N. Drivalos, R. F. Paige, and A. Rashid, "Domain-specific metamodelling languages for software language engineering," in *SLE*, pp. 334–353, 2009.

[81] B. Gruschko, D. S. Kolovos, and R. F. Paige, "Towards synchronizing models with evolving metamodels," in *Proceedings of the International Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.

[82] A. Demuth, "Enabling dynamic metamodels through constraint-driven modeling," in *ICSE*, pp. 1622–1624, 2012.

[83] M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "COPE - automating coupled evolution of metamodels and models," in *ECOOP*, pp. 52–76, 2009.

[84] J. Pardillo, "A systematic review on the definition of UML profiles," in *MoDELS*, pp. 407–422, 2010.

[85] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Automatically generating and adapting model constraints to support co-evolution of design models," in *ASE*, pp. 302–305, 2012.

[86] Eclipse Foundation, "Eclipse Modeling Framework (EMF)." http://eclipse.org/modeling/emf/.

[87] A. Egyed, "Instant consistency checking for the UML," in *ICSE*, pp. 381–390, 2006.

[88] X. Blanc, A. Mougenot, I. Mounier, and T. Mens, "Incremental detection of model inconsistencies based on model operations," in *CAiSE*, pp. 32–46, 2009.

[89] C. Atkinson, B. Kennel, and B. Goß, "The level-agnostic modeling language," in *SLE*, pp. 266–275, 2010.

[90] K. Hassam, S. Sadou, V. L. Gloahec, and R. Fleurquin, "Assistance system for OCL constraints adaptation during metamodel evolution," in *CSMR*, pp. 151–160, 2011.

[91] S. Markovic and T. Baar, "Refactoring OCL annotated UML class diagrams," in *MoDELS*, pp. 280–294, 2005.

[92] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel, "Refactoring UML models," in *UML*, pp. 134–148, 2001.

[93] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, pp. 600–624, 2007.

[94] Object Management Group, "Meta-Object Facility (MOF)." http://www.omg.org/mof/.

[95] A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "Managing dependent changes in coupled evolution," in *ICMT*, pp. 35–51, 2009.

[96] M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "Automatability of coupled evolution of metamodels and models in practice," in *MoDELS*, pp. 645–659, 2008.

[97] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering - foundations, principles, and techniques*. Springer, 2005.

[98] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook, "Safe composition of product lines," in *GPCE*, pp. 95–104, 2007.

[99] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, and K. Pohl, "Variability issues in software product lines," in *PFE*, pp. 13–21, 2001.

[100] K. Pohl and A. Metzger, "Software product line testing," *Commun. ACM*, vol. 49, no. 12, pp. 78–81, 2006.

[101] M. Svahnberg and J. Bosch, "Evolution in software product lines: two cases," *Journal of Software Maintenance*, vol. 11, no. 6, pp. 391–422, 1999.

[102] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness ocl constraints," in *GPCE*, pp. 211–220, 2006.

[103] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *ASE*, pp. 269–280, 2009.

[104] A. Capozucca, B. H. Cheng, N. Guelfi, and P. Istoan, "Barbados crash management system." http://www.cs.colostate.edu/remodd/v1/content/bcms-spl-case-study-proposition-based-cloud-component-approach, 2011. [accessed 1-August-2012].

[105] ReMoDD Team, "Repository for model driven development (ReMoDD)." http://www.cs.colostate.edu/remodd/v1/, 2011.

[106] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *ASE*, pp. 99–108, 2008.

[107] K. Czarnecki and U. W. Eisenecker, *Generative programming - methods, tools and applications.* Addison-Wesley, 2000.

[108] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.

[109] J. Sun, H. Zhang, Y.-F. Li, and H. H. Wang, "Formal semantics and verification for feature modeling," in *ICECCS 2005*, pp. 303–312, 2005.

[110] M. Mendonca, A. Wasowski, and K. Czarnecki, "SAT-based analysis of feature models is easy," in *SPLC*, pp. 231–240, 2009.

[111] R. Mazo, R. E. Lopez-Herrejon, C. Salinesi, D. Diaz, and A. Egyed, "Conformance checking with constraint logic programming: The case of feature models," in *COMPSAC*, pp. 456–465, 2011.

[112] R. E. Lopez-Herrejon and A. Egyed, "Detecting inconsistencies in multi-view models with variability," in *ECMFA*, pp. 217–232, 2010.

[113] S. Apel, C. Kästner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *ICSE*, pp. 221–231, 2009.

[114] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE*, pp. 422–437, 2005.

[115] P. Trinidad, D. Benavides, A. R. Cortés, S. Segura, and A. Jimenez, "FAMA framework," in *SPLC*, p. 359, 2008.

[116] M. Mendonca, M. Branco, and D. D. Cowan, "S.P.L.O.T.: Software product lines online tools," in *OOPSLA Companion*, pp. 761–762, 2009.

[117] R. E. Lopez-Herrejon and A. Egyed, "On the need of safe software product line architectures," in *ECSA*, pp. 493–496, 2010.

[118] A. Classen, P. Heymans, T. T. Tun, and B. Nuseibeh, "Towards safer composition," in *ICSE Companion*, pp. 227–230, 2009.

[119] J. Shi, M. B. Cohen, and M. B. Dwyer, "Integration testing of software product lines using compositional symbolic execution," in *FASE*, pp. 270–284, 2012.

[120] M. Cordy, A. Classen, P.-Y. Schobbens, P. Heymans, and A. Legay, "Managing evolution in software product lines: a model-checking perspective," in *VaMoS*, pp. 183–191, 2012.

[121] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE*, pp. 321–330, 2011.

[122] H. Sabouri and R. Khosravi, "Efficient verification of evolving software product lines," in *FSEN*, pp. 351–358, 2011.

[123] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity, "Incremental model-based testing of delta-oriented software product lines," in *TAP*, pp. 67–82, 2012.

# Andreas Demuth

Johannes Kepler University (JKU) Linz
Institute for Systems Engineering and Automation (SEA)
Altenbergerstrasse 69
4040 Linz, Austria
Phone: +43 (0)732 2468-4389
Fax: +43 (0)732 2468-4385
Mobile: +43 (0)699 1089-5113
Email: andreas.demuth@jku.at

**EDUCATION**

*Bachelor of Science,* Business and Economics — Summer 2014 (expected)
Johannes Kepler University, Linz, Austria
Concentration: Applied Economics
Minor: Management, Business Informatics

*Doctor of Philosophy,* Technical Sciences — October 2013 (expected)
Johannes Kepler University, Linz, Austria
Major: Computer Science — Software Engineering
Thesis: Automatic Model-based Management of Design Constraints

*Dipl.-Ing.,* Software Engineering — October 2010
Johannes Kepler University, Linz, Austria
Thesis: General Purpose Dependency Viewer for the NetBeans Application Platform
Minor: Networks and IT-Security

*Bachelor of Science,* Computer Science — July 2009
Johannes Kepler University, Linz, Austria
Thesis: Visualization of Context-free Grammars

**EXPERIENCE**

*Researcher* — January 2011 – present
Johannes Kepler University, SEA, Linz, Austria

- Development, testing, analysis, publication, and presentation of theories on incremental constraint generation and management in different domains.
- Refactoring of incremental consistency checking framework to support generic design models.
- Acquisition of research funds.

*Programmer (Intern)* Fall 2009
Johannes Kepler University, SEA, Linz, Austria

- Development of Google Wave application to support the WinWin requirements engineering process.

*Programmer (Intern)* Fall 2008 – Spring 2009
Johannes Kepler University, Institute for System Software, Linz, Austria

- Development of visualization component for compiler generator framework.

**TEACHING**

*Johannes Kepler University, Linz, Austria* WS 2012/2013 – present
Supervision of Bachelor's and Master's projects

*Johannes Kepler University, Linz, Austria* SS 2013
Seminar: Software Engineering

*Johannes Kepler University, Linz, Austria* WS 2012/2013
Scientific Working and Presentation Skills

*Johannes Kepler University, Linz, Austria* WS 2011/2012
Scientific Working and Presentation Skills

**COMPUTER SKILLS**

*Languages & Frameworks:* Java, C/C++, C#, JavaScript, Perl, Python, Smalltalk, SQL, XML, HTML, LISP, Haskell, F#, FORMULA; .Net, Java Enterprise Beans, JSP, JSF, Spring, Hibernate, JUnit, Ajax, Eclipse RCP, NetBeans Platform.

*Software:* MS Office, Visual Studio, Eclipse IDE, NetBeans IDE, IBM Rational Software Architect, MySQL Workbench, VMWare, SPSS, Stata, R, LaTeX.

*Operating Systems:* Windows, Unix, Mac OS.

**LANGUAGES**

*German:* Native speaker
*English:* Excellent
*French:* Basic

**PUBLICATIONS (peer-reviewed)**

1. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Constraint-driven modeling through transformation," in *ICMT*, pp. 248–263, 2012.
2. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Constraint-driven modeling through transformation," *Software and System Modeling*, 2013, extended version, DOI: 10.1007/s10270-013-0363-3.
3. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Automatically generating and adapting model constraints to support co-evolution of design models," in *ASE*, pp. 302–305, 2012.
4. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Supporting the co-evolution of metamodels and constraints through incremental constraint management," in *MoDELS*, 2013, accepted for publication.
5. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Co-evolution of metamodels and models through consistent change propagation," in *MoDELS-ME*, 2013, accepted for publication.
6. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Cross-layer modeler: A tool for flexible multilevel modeling with consistency checking," in *ESEC/SIGSOFT FSE*, pp. 452–455, 2011.
7. A. Demuth, "Enabling dynamic metamodels through constraint-driven modeling," in *ICSE*, pp. 1622–1624, 2012.
8. A. Egyed, A. Demuth, A. Ghabi, R. E. Lopez-Herrejon, P. Mäder, A. Nöhrer, and A. Reder, "Fine-tuning model transformation: Change propagation in context of consistency, completeness, and human guidance," in *ICMT*, pp. 1–14, 2011.