Co-evolution of Metamodels and Models through Consistent Change Propagation

Andreas Demuth*, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon, Alexander Egyed

Institute for Software Systems Engineering, Johannes Kepler University (JKU), Altenberger Strasse 69, 4040 Linz, Austria

Abstract

In Model-Driven Engineering (MDE), metamodels and domain-specific languages are key artifacts as they are used to define syntax and static semantics of domain models. However, metamodels are evolving over time, requiring existing domain models to be co-evolved. Though approaches have been proposed for performing such co-evolution automatically, those approaches typically support only specific metamodel changes. In this paper, we present a vision of co-evolution between metamodels and models through consistent change propagation. The approach addresses co-evolution issues without being limited to specific metamodels or evolution scenarios. It relies on incremental management of metamodel-based constraints that are used to detect co-evolution failures (i.e., inconsistencies between metamodel and model). After failure detection, the approach automatically generates suggestions for correction (i.e., repairs for inconsistencies). A case study with the UML metamodel and 23 UML models shows that the approach is technically feasible and also scalable.

Keywords: Metamodel co-evolution, consistency checking, consistent change propagation, model repairs

1. Introduction

In *Model-Driven Development (MDD)* (Schmidt, 2006), metamodels are key artifacts that represent real-world domains. They define the language of models; that is, the different elements available for modeling along with their interdependencies. Metamodels also impose structural and semantic constraints on models (France and Rumpe, 2007). Although metamodels are often perceived as static artifacts that ought not change, it has been shown that the opposite is the case: metamodels do evolve over time (Di Ruscio et al., 2012; Herrmannsdoerfer et al., 2009b). There are a variety of reasons for this (Di Ruscio et al., 2011). For instance, metamodels often reflect domain models which are inherently volatile and today there is a trend for flexible design tools with adaptable metamodels that can be tailored to different domains (e.g., Manders et al., 2006). Indeed, there is even the FlexiTools workshop series dedicated to exploring such tools. Other common sources for metamodel evolution are refactorings that focus on improving a metamodel's structure and usability or the continuous evolution of tools and their languages (Di Ruscio et al., 2012).

Co-evolution of models denotes the process of adapting models as a consequence of metamodel evolution (Mens et al., 2005; Rose et al., 2009). This is a nontrivial process, and incorrect co-evolution may cause models to no longer comply with their metamodels. Several incremental approaches have been proposed to support this process (e.g., Herrmannsdoerfer et al., 2009a)). Unfortunately, proposed solutions are typically limited to specific metamodels or do not fully support all kinds of possible changes (e.g., restriction of metaproperty) (Cicchetti et al., 2008). In particular, existing generic approaches do not take into account domain-specific model constraints. Thus, they might produce results that are conforming to the updated metamodel but that are not valid (or intended) in the specific modeling context. There-

*Corresponding author. Tel.:+43 732 2468-4389.
Email addresses: andreas.demuth@jku.at (Andreas Demuth), markus.riedl@jku.at (Markus Riedl-Ehrenleitner), roberto.lopez@jku.at (Roberto E. Lopez-Herrejon), alexander.egyed@jku.at (Alexander Egyed)

fore, co-evolution of metamodels and models remains an open issue.

In this paper, we outline a generic approach that does not try to automate co-evolution in general, but that detects co-evolution failures and suggests model adaptations to co-evolve a model correctly. Specifically, our approach does not focus only on traditional model conformance, but it also takes into consideration additional domain- or even project-specific constraints (environment-based constraints). The approach relies on incremental constraint management that allows for efficient detection of co-evolution failures (including the absence of co-evolution). If such failures are detected, the resulting inconsistencies between metamodel and model – along with all design constraints imposed on the model – are used for finding suitable model adaptations (repairs). These repairs establish conformance of the model with the updated metamodel by removing any constraint violations. Moreover, repairs also take into consideration environment-based constraints to ensure that the repair leads to a model that is also valid with respect to the specific modeling context (i.e., it does not violate any metamodel- or environment-based constraints). This use of additional information and the resulting validity of co-evolved models presents a major qualitative benefit over existing approaches. The proposed approach follows the principles of *consistent change propagation (CCP)*, a *change propagation* (Hassan and Holt, 2004) process which is driven by arising inconsistencies. It does not, in contrast to other change propagation processes, try to maintain consistency automatically by using on heuristics or transformation rules (e.g., Eramo et al., 2012).

This paper is an extended version of previous work (i.e., Demuth et al., 2013a,c). Additional contributions include a more detailed description of the concepts of CCP in general and its application in different metamodel evolution scenarios in particular, an in-depth analysis of the feasibility and the scalability of the approach, especially of the constraint management part, using a new prototype implementation that does support a broader variety of metamodels and models compared to Demuth et al. (2013c), and a more comprehensive and extensive discussion of related work.

The remainder of the paper is organized as follows. In Section 2 we introduce a motivating example that is used throughout the paper to illustrate our approach and to highlight differences to existing approaches. In Section 3, we give an overview of state-of-the-art approaches for metamodel-to-model co-evolution and discuss the key aspects of consistent change propagation in general. Moreover, we discuss how the general concept of CCP is tailored to the problem of metamodel-to-model co-evolution. In Section 4 and Section 5, we respectively present in detail how our approach manages constraints incrementally and generates possible repairs. In Section 6 we present a case study with the *Unified Modeling Language (UML)* (Object Management Group, 2013b) metamodel and industrial UML models and that was used to validate our approach. We discuss the developed prototype implementation and present scalability results. In Section 7, we further analyze those results, the general applicability of our approach, and possible threats to validity. Related work is discussed in Section 8 and in Section 9 we conclude the paper and briefly describe future research paths on the topic.

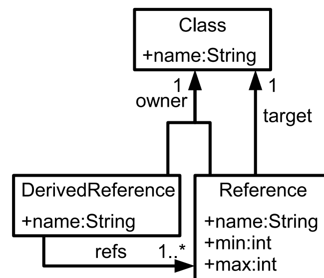

Figure 1: Metametamodel.

2. Motivating example

Let us now present a motivating example which we will use throughout the paper to illustrate our work. We begin with a description of the metamodel.

2.1. (Meta)Metamodel

To illustrate our work, we use a minimalistic metametamodel, as shown in Fig. 1, to define a metamodel for component-oriented systems with high availability requirements, as shown in Fig. 2. The metametamodel we use in our example contains three kinds of elements that are available for building metamodels: `Class`, `Reference`, and `DerivedReference`. Instances of all three elements must provide the attribute `name`. Instances of `Reference` must provide a minimum (`min`)

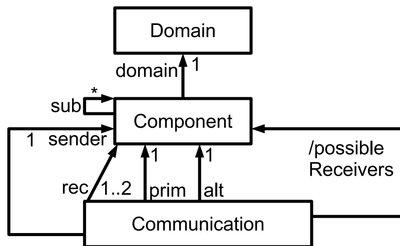

Figure 2: Metamodel.

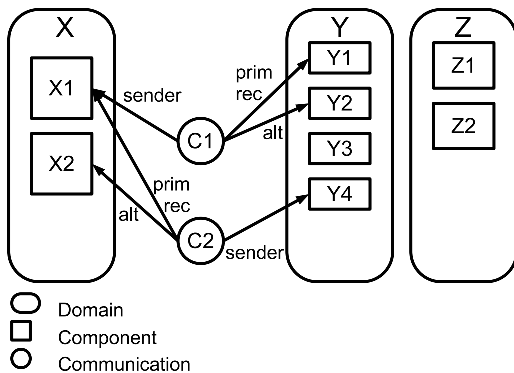

Figure 3: Model.

and maximum (`max`) number of referenced classes. Moreover, each reference must specify an `owner` class as well as a `target` class. Derived references are used to aggregate navigation results obtained by other references. Thus, a `DerivedReference` must specify at least one `Reference` for which it aggregates the results (with the reference `refs`).

The metametamodel was used to construct a metamodel, as shown in Fig. 2, that defines three classes: `Component`, `Domain`, and `Communication`. Components can have an arbitrary number of `sub`-components and must belong to a `Domain`. Domains include components that are responsible for fulfilling a common task in the system. Communications occur between a `sender` and either one or two receiver (`rec`) components. To increase the change of a successful communication, different components can be specified as primary (`prim`) and a alternative (`alt`) target of a communication. All possible targets of a communication (i.e., the components reached through the references `prim` and `alt`) are aggregated by the derived reference `possibleReceivers`.

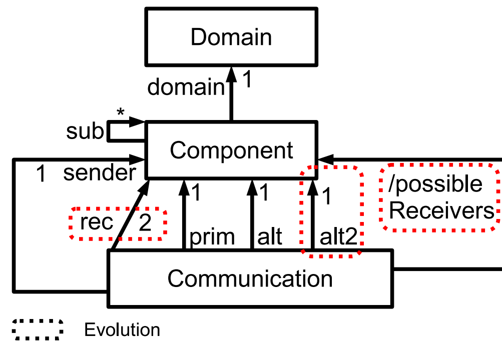

Figure 4: Updated metamodel.

2.2. Sample model

In Fig. 3, a model that complies with the metamodel is depicted. The derived reference `possibleReceivers` is omitted for readability reasons. The model contains three domains: X, Y, and Z. Domain X consists of only two components ($X1$ and $X2$), Y consists of four components in total ($Y1 - Y4$), and Z consists of two components ($Z1$ and $Z2$). The model also contains two communications ($C1$ and $C2$), drawn as circles.

2.3. Metamodel evolution

Let us now consider a simple metamodel evolution. To increase the availability of systems and reduce the chance of communication failures, a second alternative communication target (called `alt2`) is added to the metamodel, as shown in Fig. 4. Moreover, a communication can now have up to two receiver components at the same time, indicated by the change of the cardinality of `rec` from `1..2` to `2`. Finally, the derived reference `possibleReceivers` is updated to include the results obtained through the new reference `alt2`. The places of evolution are encircled in Fig. 4. Intuitively, this metamodel evolution, specifically the addition of `alt2` and corresponding change of `possibleReceivers`, requires the model in Fig. 3 to be co-evolved as the reference `alt2` is mandatory and it is owned by class `Communication` which is instantiated twice in the model. Moreover, the cardinality of the reference `rec` has been narrowed to a single value. However, as we will show below, finding suitable adaptations, even for such a small metamodel evolution, is a challenging task that is far from trivial.

In the next section, we will illustrate how existing co-evolution approaches handle this evolution scenario and describe how our proposed approach

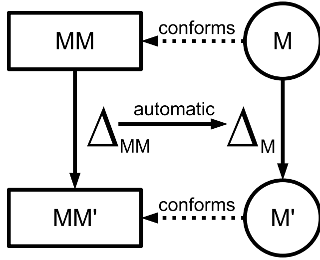

Figure 5: Traditional co-evolution approach.

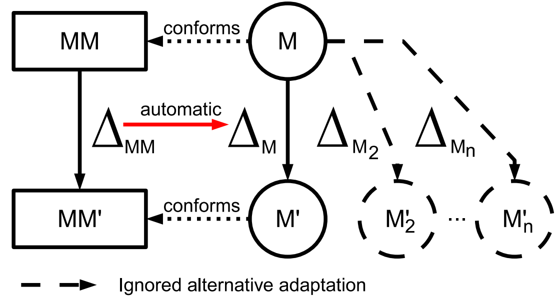

Figure 6: Issue arising with traditional co-evolution approaches.

of co-evolution through consistent change propagation tackles this problem.

3. Co-evolution through consistent change propagation

To the metamodel evolution scenario presented above, we propose to perform co-evolution through consistent change propagation. However, before we discuss different aspects of our approach in detail, let us first illustrate how co-evolution is typically handled by existing approaches. Then, we provide an overview of the key concepts involved in consistent change propagation and discuss the key differences between our approach and existing ones.

3.1. Traditional co-evolution approaches

Since co-evolution of metamodels and models is an important issue, various approaches have been proposed. Basically, there are two major strategies for tackling the issue: i) metamodel-change-based model adaptation and ii) strategy-based model migration. Approaches of the former kind (e.g., Cicchetti et al., 2008, 2009; Wimmer et al., 2010; Wachsmuth, 2007) typically consider metamodel changes and, based on a set of rules, derive corresponding model changes that are executed to transform a model that conformed to the old metamodel to a model that conforms to the new version of the metamodel (e.g., Eramo et al., 2012). This general approach is visualized in Fig. 5 where the changes (depicted by Δ_{MM}) between the original and updated versions of a metamodel (i.e., MM and MM') are translated to changes (denoted by Δ_M) that transform the original model M that conforms to MM to an updated version M' that conforms to MM'. Typically, traditional co-evolution approach – at least try to – automate the generation and execution of model changes.

Even though those approaches produce updated models that conform to the updated metamodel, some issues remain unsolved. First, model adaptations are typically built using translation rules that map specific metamodel changes to corresponding target model changes. Indeed, it has been shown that this leads to conforming models (e.g., Cicchetti et al., 2008, 2009; Eramo et al., 2012). However, such translation rules do produce a single model adaptation although there might a large – probably even infinite – number of possible adaptations that lead to conforming updated models, as illustrated in Fig. 6 where the dashed arrows indicate valid but ignored alternative model adaptations (Δ_{M1} – Δ_{Mn}) that would lead to other conforming models (M'_2 – M'_n). When applied to complex models, it is thus likely that such fully automated approaches produce a correct, yet undesired solution.

Approaches of the second kind do not base model adaptations on the performed metamodel changes, but instead they rely on user-defined model migration strategies (e.g., Rose et al., 2010b,a; Sprinkle and Karsai, 2004; Narayanan et al., 2009; Jakumeit et al., 2010). These migration strategies (or model transformations) can indeed be tailored to specific domain and even individual projects. However, while it is of course possible that migration strategies are written in a way that produce not only conforming but actually valid updated models, this requires the migration strategy author to be aware of all desired model characteristics (i.e., all metamodel- and environment-based constraints). However, this is an unrealistic assumption as environment-based constraints may be of high complexity. Moreover, the desired overall characteristics of models is often a composition of hundreds of constraints which may be aggregated from diverse sources. Therefore, writing migra-

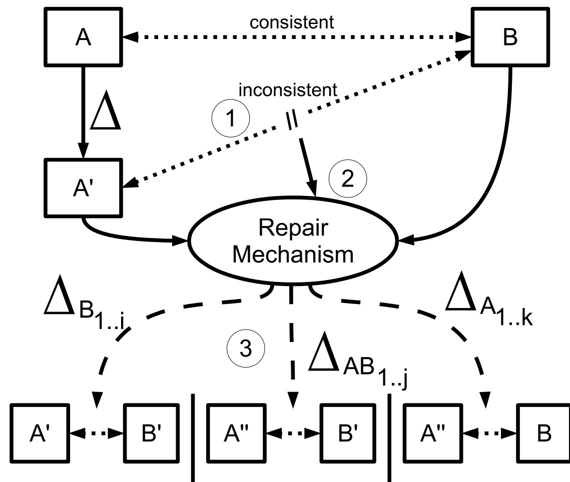

Figure 7: Consistent change propagation.

tion strategies manually always imposes a high risk of producing unintended and thus invalid results. Furthermore, a migration strategy is a traditional model transformation whose support for handling specific characteristics of the transformed model is rather limited (i.e., the migration rules transform all model elements of certain type uniformly according to the specification; there is little to no consideration of the context in which the transformed model element is used). Thus, using migration-strategies imposes the chance of taking premature design decisions and not using information available at migration time.

3.2. Principles of consistent change propagation

Consistent change propagation (CCP) is an approach that keeps development artifacts, such as design models for example, consistent during evolution. To achieve this, CCP relies on consistency rules (also called constraints) that define invariants between artifacts, that is, conditions that must hold in order to identify them as consistent. The overall goal of CCP is to guide developers through necessary adaptations after individual parts of a system have been evolved.

CCP is a generic, three-step evolution handling process that is illustrated in Fig. 7.

3.2.1. Step 1

In the first step, CCP determines the new consistency status of artifacts after evolution based on a set of constraints and existing consistency checking technologies (constraints and consistency checker are not depicted in Fig. 7). Note that in Fig. 7 an evolution of the artifact A (denoted by Δ) causes an inconsistency between A' (the updated version of A) and artifact B. Thus, the first step constitutes a typical *change impact analysis* (Iovino et al., 2012). However, note that neither the performed change itself nor the information about the changed element (e.g., its type) is used for reasoning, but the actual effect of the performed change is analyzed using existing constraints. Thus, our approach by default supports any kind of metamodel adaptation.

3.2.2. Step 2

In the second step, CCP determines how artifacts can be adapted in order to remove inconsistencies (i.e., it derives possible repairs). This is done by a repair mechanism that takes into account the involved artifacts as well as all existing constraints. As shown in Fig. 7, possible repairs (depicted with dashed arrows) may adapt a single artifact (e.g., B could be evolved to B'), as shown in the bottom-left and bottom-right parts of Fig. 7, or multiple artifacts (e.g., A' could be evolved to A'' and B could be evolved to B'), as shown in the bottom-center part of Fig. 7. For each of those categories, there might be a large number of possible adaptations, as indicated by the respective subscripts in Fig. 7.

3.2.3. Step 3

In the third and final step, inconsistency information and possible repairs are presented to artifact developers (e.g., designers) who can then select the most appropriate solution. In contrast to the first two steps (i.e., inconsistency detection and repair option generation) which are performed automatically, this step of repair execution usually require manual intervention.

3.3. Consistent propagation of metamodel changes

As discussed above, consistent change propagation focuses on maintaining consistency between artifacts. This of course includes consistency between the artifacts *metamodel* and *model* (i.e., conformance). In this and the following sections, we will not only illustrate how the use of CCP for handling metamodel evolution addresses the issue of unintended model adaptations we discussed above, but we will also discuss how using CCP can improve the quality of found solutions by taking into account other factors that are generally ignored by other co-evolution approaches (e.g., Cicchetti et al.,

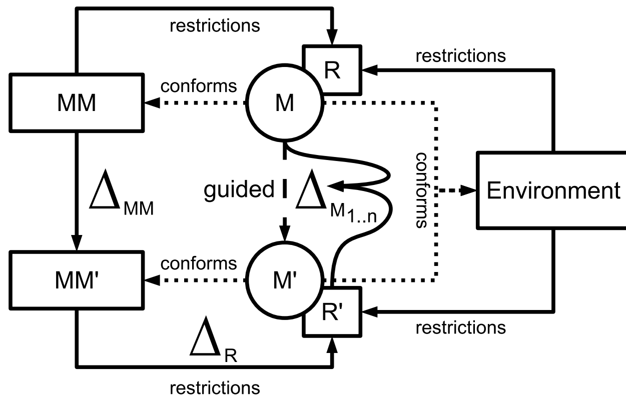

Figure 8: Co-Evolution through Consistent Change Propagation.

2009). That is, our approach not only considers model constraints imposed by the metamodel, but it also considers other, typically domain- or project-specific, constraints.

However, applying CCP to address the issue of metamodel evolution and model co-evolution does require some adaptations of the standard three-step process described above. Specifically, the conformance rules that are used to determine whether a model is conforming to its metamodel do change as the metamodel evolves. For instance, recall the metamodel evolution described in Section 2 in which a new mandatory reference was added to the metamodel. Intuitively, this requires a new constraint being added to the consistency checker. Therefore, a mechanism for constraint management is required to ensure that a correct set of constraints is used for consistency checking. Moreover, CCP generally provides repair options for inconsistencies that may include changes to all involved artifacts. However, based on the typical understanding of co-evolution we assume that repair operations should not change the metamodel but only the model. Thus, we restrict the repair options to those that include model adaptations only.

The adapted CCP process for handling metamodel evolutions consists of two major phases:

Phase 1: Detect co-evolution failures. In this phase, the approach performs an automatic update of constraints and detects locations where co-evolution is not performed correctly (i.e., where inconsistencies occur). This, of course, includes the situation of plainly missing co-evolution.

Phase 2: Derive options for correction of failures In this phase, options for a correct propagation of the metamodel change to the affected model are derived.

In Fig. 8, the adapted process is depicted. In contrast to Fig. 7, in which bidirectional arrows were used to depict *(in-)consistency* between artifacts, in Fig. 8 unidirectional arrows are used to indicate that the model M must *conform* to the metamodel MM. Thus, the model is the artifact to be adapted in order to establish conformance – adaptations of the metamodel are not desired. Furthermore, the figure includes a new artifact called *Environment*. This artifact includes all other influences besides the metamodel that determine desired model characteristics. Both, the metamodel and the environment imposes restrictions (i.e., constraints) on models. This is depicted by the solid arrows named *restrictions*. The imposed restrictions themselves are drawn as boxes partially hidden by the models. A model that does not violate any of the imposed restrictions does not only conform to its metamodel, but also to the general environment. Note that the initial metamodel update (Δ_{MM}) that leads to MM' is no longer used directly to find model adaptations. Instead, the updated version of the metamodel (MM') is used to updated the restrictions imposed on models automatically (note the Δ_R), leading to an updated set of restrictions R'. Based on the initial model version M and the updated restrictions R', possible repairs $\Delta_{M1..n}$ are derived automatically. Note again that, even though their derivation is done automatically, no repair is executed automatically. Instead, those possible repairs guide developers who manually select the repair option that seems most suitable for the specific model. This manual step is illustrated through the use of a dashed arrow between M and M' instead of the solid arrow that was used in Fig. 5.

We will now discuss the two major phases in more detail and show how the approach handles the evolution scenario presented above. The focus of this paper, however, is on the constraint management part in Phase 1 as this part of the process is critical but generally not handled by existing technologies.

4. Phase 1: Co-evolution failure detection

Although metamodel evolution is likely to require model adaptations, this is not a necessity – a metamodel may also change in ways that do not affect the validity of existing models. For example, when

an optional reference is added. Additionally, models may be changed manually by designers or automatically by tools after a metamodel evolution occurred, trying to co-evolve the model. Therefore, it is necessary after a metamodel change – and subsequent model adaptations – to determine whether an affected model is consistent with the updated metamodel. If it is, co-evolution was performed correctly and no further intervention is required. If, however, the model is inconsistent with the updated metamodel, co-evolution failed and additional model adaptations are necessary.

4.1. Constraint management

As we have shown above, constraints can be used to check whether a model conforms to its metamodel. However, when the metamodel evolves, constraints that are based on metamodel information may become invalid. Specifically, individual constraints may no longer be required, they may require updating, or new constraints may become required. By using an incremental constraint management approach, it is possible to update constraints after metamodel changes – ensuring that models are always validated with an up-to-date set of constraints.

We propose the use of constraint templates to automate the co-evolution of metamodels and their constraints. Based on metamodel elements, model constraints are built automatically through template instantiation. Basically, templates contain the static aspects that constraints have in common (e.g., fragments of a constraint string) and define the points of variability. Thus, a single template defines a *constraint family*. In our example, we use the *Object Constraint Language (OCL)* (Object Management Group, 2013a) since it is a well-known and commonly used constraint language. However, any constraint language may be used in principle. As a metamodel evolves, templates are filled with specific data – to reflect the evolution – and instantiated to automatically generate or update the constraints.

Before we discuss in detail how templates are written and constraints are generated, let us return to our motivating example and show the constraints that are necessary to check whether a model is valid as well as the effects of metamodel evolution on the correctness of those constraints.

4.1.1. Constraints for motivating example

The defined reference cardinalities (e.g., `1` for `prim`) in Fig. 2 implicitly define model constraints. For example, an instance of `Communication` must reference exactly one `Component` via `prim`. However, to apply consistent change propagation, we define those constraints explicitly so that they can be passed to a state-of-the-art consistency checker. In particular, the metamodel in Fig. 2 defines the following seven constraints:[1]

RM1 Each component belongs to a single domain that is reached via `domain`.

RM2 Each component may have an arbitrary number of sub-components reached via `sub`.

RM3 Each communication has a sender component reached via `sender`.

RM4 Each communication has one or two receiver components reached via `rec`.

RM5 Each communication has a single primary target reached via `prim`.

RM6 Each communication has a single alternative target reached via `alt`.

RM7 Each communication provides the components reached via `prim` and `alt` (i.e., a collection of all possible receivers) via `possibleReceivers`.

However, to ensure that only intended models can be built, the metamodel-based constraints above have been amended with the following three explicit, environment-based, project-specific constraints:

RE1 All possible receivers of a `Communication` must be located within a single `Domain` (i.e., a set of components with a common purpose).

RE2 A communication may only occur between components of different component domains.

RE3 It is not permitted that a single component is used as primary and alternative target.

By stating model constraints explicitly, two important aspects become clear. First, metamodel-based constraints often have a similar structure

[1] We shall express the constraints more formally later by using OCL.

Table 1: Template structure.

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

Table 2: Definition of template T1.

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

(e.g., $RM1$–$RM6$). Therefore, generating those constraints instead of writing them manually seems intuitive. Second, metamodel-based constraints may easily be affected by metamodel evolution. Recall the evolution scenario from Section 2.3 in which new references were added and existing ones where changed. Specifically, the evolution scenario has three effects on constraint validity: 1) the cardinality change in the reference `rec` invalidates the constraints $RM4$, 2) the addition of the reference `alt2` requires a new metamodel-based constraint $RM8$ that checks this reference in models, and 3) the addition of `alt2` to the set of references aggregated by the derived reference `possibleReceivers` invalidates the constraint $RM7$. Therefore, constraints need to be updated after the metamodel evolution in order to obtain correct validation results. Although these effects are easy to track and to address in our example because of its simplicity, note that manually handling such simple evolution scenarios in industrial projects with complex metamodels and hundreds or even thousands of constraints is error-prone and often practically infeasible.

Next, we illustrate how constraint templates can be written and how they are managed by a template engine to automate constraint generation and updating.

4.1.2. 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 1, and the information it requires. The *instantiation context (IC)* defines for which elements, or combinations thereof, a template should be instantiated. 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)*. Let us now show how we can write a template $T1$ for the constraint family of $RM1$–$RM6$ (i.e., the metamodel-based constraints for references).

Template for cardinalities. The top-right section "Template definition" in Fig. 9 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 2, creates a constraint for every instance of the metametamodel element `Reference`, for example when the reference `rec` 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 `Reference` to the template in order to create a new constraint. Note that templates are reusable for other metamodels that are based to the same metametamodel. 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 $RM4$ for the reference `Communication.rec` in Fig. 2, which can be formalized as the following OCL constraint:

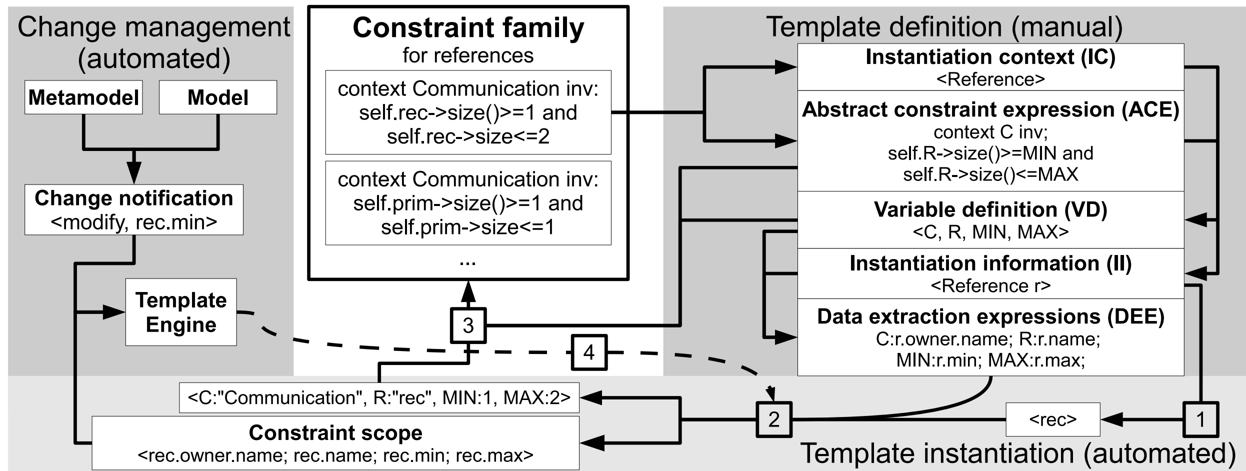

Figure 9: Example of steps performed during template definition, instantiation, and change management.

```
context Communication inv:
  self.rec->size()>=1 and
    self.rec->size()<=2
```

We replace the two values 1 and 2 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 `rec` with `R` for the used reference. The result is the abstract constraint expression:

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

as defined in Table 2 with the variable parts (VD) being `<C, R, MIN, MAX>`. As shown in Fig. 9, 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. We have now completed the template definition for $T1$.

Template for derived references. We use the same process to write template $T2$, as shown in Table 3, based on the constraint $RM7$ as an example for the constraint family that checks derived references. The constraint $RM7$ can be expressed in OCL using the following expression:

```
context Communication inv:
  self.possibleReceivers->includesAll(
  Set{self.prim, self.alt}->flatten())
```

As a simplification, we replace the set of references (`Set{self.prim, self.alt}->flatten()`) 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.possibleReceivers`, the resulting constraint is:

```
context Communication inv:
self.possibleReceivers->includesAll(
     Set{''prim'',''alt''}
     ->collect(x|self.{x}))
```

The following expression collects all the elements returned by the expressions `self.prim` and `self.alt`:

```
Set{''prim'',''alt''}->collect(x|self.{x})
```

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

4.1.3. 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. 2.

Table 3: Definition of template T2.

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

For each occurrence of the IC `<Reference>`, the template is instantiated once. In Fig. 2 there are six references and thus $T1$ is instantiated exactly six times. However, we focus on a detailed discussion of the instantiation process for the reference `Communication.rec`, as illustrated in the bottom box "Template instantiation" in Fig. 9. The process starts with the instantiation information (1). In this case, it contains the reference `rec`. The data extraction expressions are applied to the element to retrieve the names (i.e., `Communication` and `rec`) and the cardinality values (i.e., `1` and `2`). This is shown in Fig. 9(2). In order to allow later updates of the generated constraints, traceability is generated automatically by building the *constraint scope* during the execution of the DEEs in step (2). This scope contains all elements that are accessed by the DEEs during the instantiation for the given instantiation information (i.e., `rec`). The scope for the constraint $RM4$ is therefore `<rec.owner.name, rec.name, rec.min, rec.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. 2, template $T1$ was instantiated once for every reference (i.e., six times in total), template $T2$ was instantiated once to generate the constraint for the only derived reference (i.e., `possibleReceivers`) 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 illustrate how automatic constraint updates are performed.

4.1.4. Evolution handling

In Section 4.1.1 we discussed the effects of our sample metamodel evolution scenario on the validity of constraints. We will now present how such metamodel evolutions are handled automatically by the template engine.

In our approach, evolution handling is an event-based and incremental process. After every modification of the metamodel, the template engine is notified about the modification, as shown in the top-left box "Change management" in Fig. 9. The change notification includes information about the changed metamodel elements which the engine uses to determine the actions that are required to adapt the current set of constraints to the new version of the metamodel.

After the addition of a metamodel element, the engine looks for templates that can be instantiated (i.e., templates where the type of the added metamodel element matches the instantiation context). When a metamodel element is deleted, constraints that are based on this element (i.e., that were generated through instantiation of templates with the 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. 4. We first replaced the lower bound value 1 of the constraint $RM4$ with the value 2. The change notification that is passed to the engine indicates that the metamodel element `rec.min` was modified. Since the scope of the constraint $RM4$ 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, $RM4$ 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 (4) in Fig. 9, and replacing the outdated values in the constraint expression with the newly retrieved ones. In our example, `rec.min`

now returns the value 2. Replacing the old value results in the new constraint expression

```
context Communication inv:
 self.rec->size()>=2 and
   self.rec->size()<=2
```

And the constraint co-evolution of $RM4$ to $RM4'$ was completed successfully. Note that currently we delete the existing constraint and re-instantiate the template to generate an updated constraint. Incremental updates of single values or logical fragments in existing constraints without re-instantiation of a template will be addressed in future work.

The second metamodel modification we have to consider is the addition of the new reference `alt2` to `Communication`. When the template engine is informed that a reference has been added, it automatically discovers that this element matches the instantiation context of template $T1$. Therefore, template instantiation is triggered and the instantiation information `<alt2>` is used by the data retrieval expressions to retrieve the values that are then used to replace the variables in $T1$ in order to produce the required constraint $RM8$:

RM8 Each communication has a single second alternative target reached via `alt2`.

This constraint has the following formal representation in OCL:

```
context Communication inv:
 self.alt2->size()>=1 and
   self.alt2->size()<=1
```

The third metamodel evolution we discussed earlier was the adaptation of the derived reference `possibleReceivers` to include the newly added reference `alt2`. This adaptation affects the property `possibleReceivers.refs`, which is in the scope of the instantiation of template $T2$ for the derived reference *possibleReceivers* that generated the constraint $RM7$. Therefore, $RM7$ must be updated to $RM7'$ by re-instantiating $T2$. After updating, the expression of $RM7'$ has changed to the following:

```
context Communication inv:
self.possibleReceivers->includesAll(
    Set{''prim'', ''alt'',
 ''alt2''}->collect(x|self.{x}))
```

For the sake of completeness, let us finally consider what would happen if we remove the derived reference `Communication.possibleReceivers` in another evolution step. In that case, the template engine would identify $RM7'$ as the only constraint that was generated by instantiating a template with the removed element. Therefore, it would remove the no longer needed constraint $RM7'$ automatically.

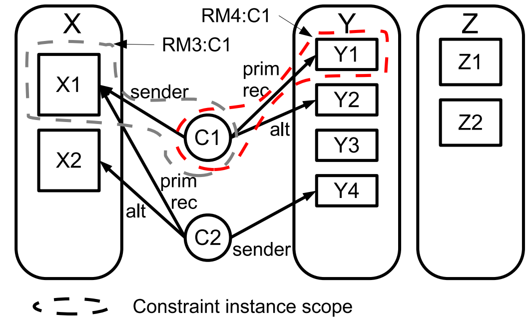

Figure 10: Constraint instance scopes for validation of $RM3$ and $RM4$ on $C1$.

4.2. Consistency checking

After updating the set of constraints imposed by the metamodel, standard consistency checking mechanisms can be used to detect inconsistencies (e.g., Nentwich et al., 2002; Groher et al., 2010). While the typical scenario is that an unchanged model becomes inconsistent after metamodel evolution, it is also possible that a previously inconsistent model becomes consistent after a metamodel evolution without any model adaptations. Additionally, model adaptations that are performed for the purpose of co-evolution, for example by automatic co-evolution approaches, may be incorrect and actually introduce new inconsistencies in case there are environment-based model constraints. If a consistency checker that uses an up-to-date set of constraints detects inconsistencies after a metamodel evolution, model adaptations are required and co-evolution was not done correctly.

4.2.1. 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 (Egyed, 2006). This set of elements can be either directly observed or

calculated from differences between model versions (Blanc et al., 2009; Reder and Egyed, 2010). The existing approach automatically defines *constraint instances* that validate whether specific model elements violate a given constraint (Egyed, 2006). 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. 10 shows scopes for *constraint instances* of the constraints $RM3$ and $RM4$ for the communication $C1$ that require $C1$ to have exactly one sender component and between 1 and 2 receiver component, respectively. The scope of a constraint instance consists of all elements that are accessed during the validation of the constraint for a specific model element. For example, the scope of the constraint instance $RM4 : C1$ includes $C1$ itself and all elements that are reached through its `rec` reference (i.e., `C1.rec`). Changes falling within scope of a constraint instance, for example the addition of another component reached via `rec` to $C1$ (i.e., a change of `C1.rec`), would lead to a revalidation of only those constraint instances that include the modified model element in their scope. This means that in Fig. 10 only the constraint instance $RM4 : C1$ would be re-evaluated whereas the instance $RM3 : C1$ would not be affected (it includes only `C1` and `C1.sender`).

State-of-the-art incremental consistency checkers automatically create, re-evaluate, and destroy constraint instances according to changes in the models. Moreover, those approaches do handle the addition and removal of constraints efficiently. For example, passing a new constraint to the consistency does not cause any existing constraint instances to be re-evaluated. Thus, we do suggest for scalability reasons that consistent change propagation is performed with incremental consistency checkers. However, any kind of consistency checking approach can be used in principle. Next, we discuss how the metamodel evolution scenario from Section 2 and the automatic constraint management discussed in Section 4.1.4 affect the consistency status of our sample model.

4.2.2. Consistency effects of sample evolution

After updating $RM4$ to $RM4'$, $RM7$ to $RM7'$, and adding the new syntactical constraint $RM8$, as presented above, any standard consistency checker will find that the previously consistent model in Fig. 3 contains the following inconsistencies after

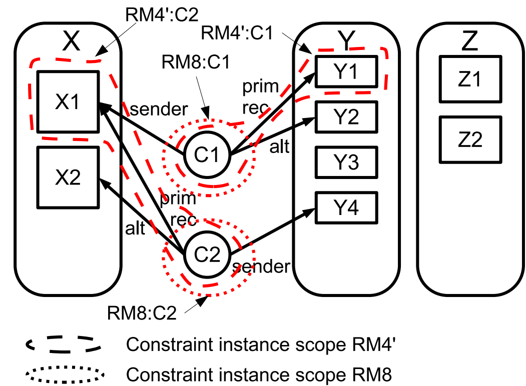

Figure 11: Scopes of inconsistent constraint instances after constraint updates.

the set of applied constraints was updated, as shown in Fig. 11: neither $C1$ nor $C2$ provide a second alternative target. Therefore, both communications do violate the newly required constraint $RM8$. Moreover, neither of those communications does provide the required number of two concurrently active receivers. Thus, they also violate the updated constraint $RM4$. Overall, our initial assumption that additional model adaptations are necessary for correct co-evolution has been confirmed. However, note that the model is still consistent with respect to $RM7'$ as the reference `alt2` for both communications `C1` and `C2` in Fig. 3 does not return any elements (causing the model to be inconsistent with respect to $RM4'$) and therefore the existing set of elements obtained through `possibleReceivers` is still correct (e.g., `Y1` and `Y2` for `C1`).

5. Phase 2: Co-evolution correction

Once co-evolution failures have been detected, our approach reaches Phase 2 in which those failures are corrected.

5.1. Repair options

To correct co-evolution failures and propagate metamodel changes correctly, it is necessary to find model adaptations (i.e., repair options) that transform the inconsistent model into a consistent one. Unfortunately, finding suitable adaptations is nontrivial as every change to a model may not only eliminate the violation of a constraint, but it may also cause other constraints to become violated. However, single changes can of course also remove

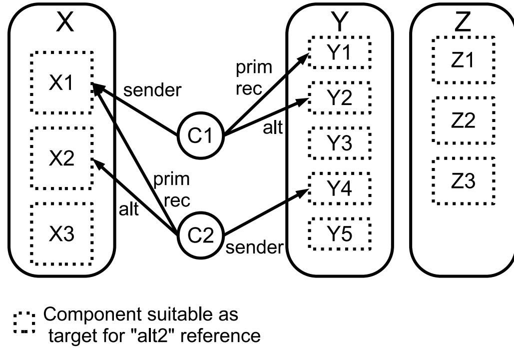

Figure 12: Repair options for missing reference `alt2`: traditional approaches.

several inconsistencies at once. Due to those *side effects* (Reder and Egyed, 2012a) of model adaptations, finding suitable corrections is a complex task that should not be performed in an ad-hoc manner. Our approach relies a reasoning mechanism that takes into consideration all design constraints present in a model to find suitable adaptations (Reder and Egyed, 2012a). Note that using not only those constraints that are actually based on the metamodel but also those based on the project environment, repair options can be computed with higher precision as more information is available for the reasoning engine when computing side effects.

Let us come back to our example and discuss the differences between traditional co-evolution approaches and co-evolution through CCP.

5.1.1. Repair options in traditional approaches

First, we look at the repair options that are typically considered by traditional co-evolution approaches. In particular, we focus on the addition of the reference `alt2`, as shown in Fig. 4, and the model shown in Fig. 3. For traditional approaches, conformance can be established by adding an appropriate reference in the model for both communications $C1$ and $C2$. As shown in Fig. 12, any existing `Component`, or also a newly added component in any existing or also newly added domain (the latter option is not depicted in the figure), would be a valid solution. Overall, 12 different solutions are possible to fix each of the two communications, which makes 144 possible solutions in total. However, it is not clear which element should be chosen for $C1$ or $C2$, respectively. If an approach performs adaptations automatically, it is likely that a solutions is chosen that is different from one that would have been produced by an informed developer. If, on the other hand, an approach does not automate the adaptation but presents possible repairs to developers, note that some of the components marked as valid second alternative targets in Fig. 12 are actually invalid. For instance, selecting the component $X1$ would be seen as valid option even though this would violate the environment-based constraint $RE1$. This simple example demonstrates that taking into account metamodel conformance only is not sufficient to derive actually valid suggestions for model adaptations.

For the changed cardinality of the reference `rec` from `1..2` to `2`, the validity of derived repair options does depend on the quality of the co-evolution approach and in particular on the complexity and level-of-detail of the used translations of metamodel changes. If only simple translations are used, traditional approaches may generate and randomly select repair options that include existing and also newly created components. If, however, more complex translation rules are used, it is possible that only valid repair options (i.e., options that suggest a component reached via either `alt` or `alt2`) are derived and executed.

Regarding the addition of the new reference `alt2` to the aggregated references in `possibleReceivers.refs`, we have discussed above that this does not cause an inconsistency immediately. Note that with any automated approach that is based on a translation of changes, the order of rule execution may be critical. In our example, let us consider first the case in which the reference `alt2` is set in an affected model and then a translation rule for the changed derived reference is executed. In this case, a sophisticated translation rule that considers not only the metamodel change itself but also the status of the affected model might be able to produce a correct solution in which the newly set second alternative target of a communication is included in the set of possible receivers. However, if the addition of the reference `alt2` was not handled before the change of the derived reference is processed, either because of rule execution scheduling issues or simply because the used approach does not chose solutions randomly when a translation is ambiguous due to multiple valid solutions, even a sophisticated translation cannot produce a valid solution as the second alternative target is not set in the model at the time the corresponding element should be added to the set of pos-

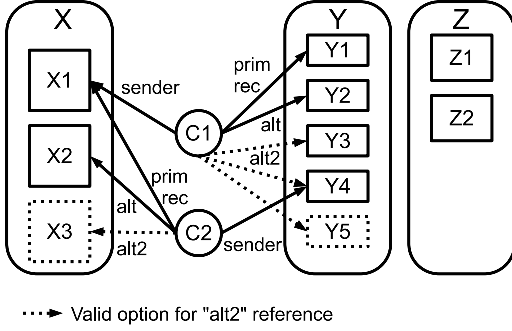

Figure 13: Repair options for missing reference `alt2`: CCP.

sible receivers. Thus, the translation would produce a result that is actually correct at the time of the translation, but not after the co-evolution process has finished and `alt2` was finally set. Even worse, such a translation would be performed automatically as there is no ambiguity involved (the second alternative target is simply added to the list of possible receivers). Therefore, later model adaptations in which that target is set do cause inconsistencies without informing artifact developers.

5.1.2. Repair options with CCP

Let us now show how the additional consideration of environment-based constraints addresses the issue we just pointed out. During Phase 1 of our approach, two inconsistencies caused by the elements $C1$ and $C2$ are detected in the model. First, we consider the inconsistency involving $C1$. To correct the syntax and remove the violation of constraint $RM8$, a reference $alt2$ to any component is sufficient. Moreover, in each domain a new component could be created and used as second alternative target for $C1$. Of course, it would also be possible to create a new component in an entirely new domain. Therefore, there are 12 options available in total: one for each of the eight existing (i.e., solid drawn) components in Fig. 13, one for each of the three domains, and one for a new domain with a new component – these are the same options that were also identified by traditional approaches as discussed above. However, by also taking into account the domain-specific constraints $RE1$ – $RE3$ from Section 4.1.1, our approach computes side effects for each of those options. Due to constraint $RE2$, adding either $X1$ or $X2$ as second alternative target to $C1$ is not a valid adaptation as this would violate $RE2$ by adding a possible receiver that is within the sender's domain. Additionally, the existing references `prim` and `alt` from $C1$ to components of domain Y disallow the use of any components that belong to a domain other than Y, according to constraint $RE1$. This rules out any remaining options that involve a second alternative receiver in domain Z or in a newly created domain. Finally, constraint $RE3$ disallows $Y1$ and $Y2$ as options because they are already possible receivers. Note that this means a reduction from 12 options – from which 9 are actually invalid when considering restrictions imposed by the project environment – to only 3 options that co-evolve $C1$ correctly. Those are drawn dotted in Fig. 13.

For the communication $C2$, the constraints $RM8$ and $RE1$ – $RE3$ can only be satisfied by adding a new component to domain X that is used as second alternative receiver, as indicated by the dotted drawn component $X3$ in Fig. 13.

Overall, using consistent change propagation reduced the number of repair options for each violation of $RM8$ from 12 to 3 and 1 for $C1$ and $C2$, respectively. This means that only a total of 3 different solutions remains, compared to 144 different solutions with traditional approaches.

For the cardinality change of `rec`, consistent change propagation will again make sure that only those components are suggested that are actually valid with respect to the defined constraints. For the change of the derived reference `possibleReceivers`, note that an inconsistency can only be detected and thus a repair options can only be computed after `alt2` was set for a communication in the model. As soon as a second alternative target has been chosen (which is indeed necessary to remove an inconsistency with respect to $RM8$, as discussed previously), an inconsistency is detected. Then, valid repair options are computed and presented to developers automatically. With co-evolution through consistent change propagation, inconsistencies are detected at all times during model evolution as soon as they occur.

5.2. Change execution

Although each derived repair option fixes a model, some of them may seem more intuitive and more logical to stakeholders than others. Therefore, stakeholders should choose manually which of the available repair options should be executed. However, repair options could of course be selected and executed automatically if model characteristics such as readability are of low importance.

In our example, the co-evolution of $C2$ can be done automatically as there is only one repair option. To repair the inconsistency of $C1$, a user has to decide between only three options that propagate the metamodel change correctly to the model.

6. Case study

To validate our approach, we conducted a case study with the UML metamodel and 23 UML models. In this case study, we systematically adapted the UML metamodel and observed the effects on the correctness of UML models. Moreover, we generated repair options for introduced inconsistencies. However, the case study focuses on assessing the scalability of the constraint management part of our approach as a) it is a key contribution of this paper, and b) it has not been validated before – in contrast to the technologies we employed for consistency checking and repair generation.

6.1. Prototype implementation

The individual parts of the approach have been implemented and validated in previous work. For the constraint management part, a template-based transformation engine had been developed that generates and updates metamodel-based model constraints (Demuth et al., 2013c). For the consistency checking, the Model/Analyzer consistency checking framework that allows for efficient incremental addition and removal of models constraints had been used (Reder and Egyed, 2010). Finally, for the repair option generation, a generic inconsistency repair mechanism that builds upon the Model/Analyzer framework had been employed (Reder and Egyed, 2012a).

However, for this case study we developed a new prototype with enhanced functionality that integrates those individual parts of constraint management, consistency checking, and repair generation, and thus presents a coherent framework for co-evolution through CCP. All involved components (i.e., template engine, Model/Analyzer consistency checker, and repair generation engine) have been adapted to use a single data repository, which not only contains metamodel and model but also consistency information. Moreover, the prototype supports arbitrary metamodels and models whereas previous implementations were limited to EMF-based metamodels and models.

As our approach, in general, is not limited to a specific consistency checker, the constraint management part was slightly adapted to reduce the requirements an employed consistency checker must meet. In particular, constraint generation was changed: instead of generating a single constraint at each template instantiation, the template engine generates one additional constraint for each subtype of the original constraint's context. For example, the constraints generated from $T1$ for the reference `rec` has the context `Communication`, as discussed in Section 4. Thus, all communications should be checked with this constraint. This, of course, also includes instances of any subtypes of `Communication`. Even though some consistency checkers are capable of handling type inheritance, our prototype would generate an additional constraint for each subtype of `Communication` (if existent). While this leads to more complexity in the constraint management component, it allows for a broader variety of consistency checkers to be used. If, however, a consistency checker is capable of handling type hierarchies, the generation of constraints for subtypes of the desired constraint context can be deactivated.

6.2. UML metamodel & UML models

The UML metamodel was chosen as the subject for our case study because it is a well known and commonly used language for modeling software systems (Lange, 2006). We argue that its size and high level of complexity make it ideal for our purposes because the sample evolutions we performed simulate typical evolutions of metamodels in general. Additionally, numerous industrial software models are available (Egyed, 2011).

For the consistency checking and repair generation parts, 23 UML models with sizes between 103 and 65,157 model elements (average: 3,929 model elements) were used. Those models typically include class and sequence diagrams as well as state machines.

6.3. Constraint templates

For our case study, we focused on templates that generated syntax constraints for UML models. The UML metamodel itself was represented in Ecore. To ensure correctness of UML models, references and attributes in the UML metamodel were translated into specific constraints. Two templates were used to generate constraints that check the correct

cardinalities of references and attributes defined in the UML metamodel. Specifically, the template $T1$ was used with the IC and II being `EReference` and `EAttribute` for references and attributes, respectively.

6.4. Metamodel evolution & test setup

Different metamodel modifications and common refactorings have been discussed in literature (e.g., Herrmannsdoerfer et al., 2009a; Cicchetti et al., 2008; Hassam et al., 2011; Markovic and Baar, 2005; Sunyé et al., 2001; Wachsmuth, 2007). During most common metamodel evolutions, references or attributes are added, removed, or they 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.

Specifically, for each of the references and attributes used in the UML metamodel (i.e., all instances of `EReference` and `EAttribute`) the following sequence of changes was performed: 1) the element was removed from the metamodel, 2) the element was added back to the metamodel, 3) the element's property `name` was changed.

In order to thoroughly assess the scalability of our prototype, the metamodel evolution scenarios were executed with systematically varied parameters. First, the number of templates may affect the time needed by the template engine for finding templates suitable for instantiation after the addition of a model element. Therefore, tests were executed with 1 to 4,000 active templates. The added templates were both copies of the templates described in Section 6.3 that had to be instantiated for metamodel elements and also dummy templates that were not instantiated. Thus, there is no perfect collinearity between the number of templates and the number of template instances, which allowed the independent investigation of the effects of increasing numbers of templates and the effects that may be caused by increasing numbers of template instances. Overall, tests were conducted with 116 to 968,402 template instances managed by the template engine. Note that an increasing number of template instances may affect the time required for finding those instances that become obsolete after the removal of a metamodel element. Last, the number of scope elements managed by the template engine was varied independently of the number of template or template instances.

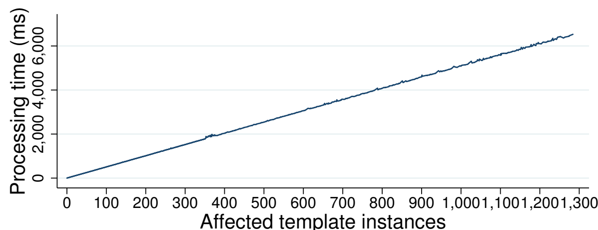

Figure 14: Total processing time for metamodel element addition depending on evolution impact.

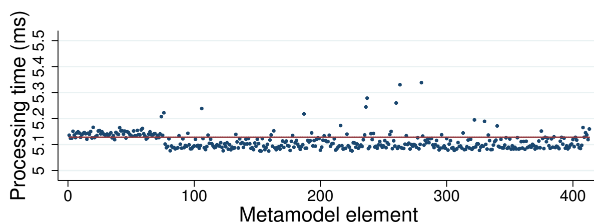

Figure 15: Processing time for metamodel element addition per affected instance depending on changed element.

Each evolution scenario, as described above, was executed 100 times for each reference and attribute in the UML metamodel and the median execution times for handling the metamodel change were used for analysis. This was repeated for each individual parameter configuration. All tests were executed on an Intel Core i5-650 machine with 8GB of memory running Windows 7 Professional. All processing times were measured in nanoseconds using standard Java functionality.

6.5. Constraint management

Next, we present the performance and scalability results regarding the constraint management part of our approach, categorized by evolution scenario. For each of the following three scenarios, the observed processing times include all processing that is required to handle a metamodel change and perform a required update of the set of constraints applied onto a model. That is, the time between the initial information about a metamodel change and the completion of all required changes to the set of applied constraints.

6.5.1. Add metamodel element

For the addition of a new metamodel element, a strong correlation between the number of newly required constraints (i.e., template instances) was observed. This correlation is visualized in Fig. 14

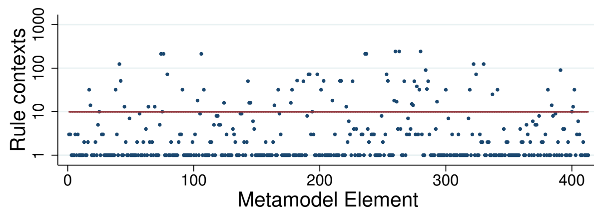

Figure 16: Number of constraint contexts depending on metamodel element.

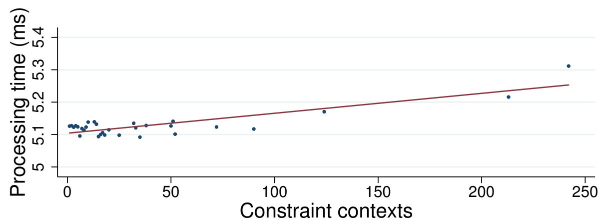

Figure 17: Processing time for metamodel element addition per affected instance depending on constraint contexts.

where the total processing time for a metamodel element addition is depicted as function of the number of affected (i.e., newly required) template instances. Note that this graph grows linearly. Furthermore, note that a single metamodel element addition – as with an element removal – can only cause a single instantiation per managed template. The practical implications of this will be discussed in Section 7.1.

The average processing time for a single template instantiation (i.e., the processing time per affected template instance) for each adapted metamodel element is depicted in Fig. 15. As mentioned above, in our prototype, the instantiation of a template produces one constraint for each type for which a constraint needs to be validated. Thus, the processing time may be affected by the number of types for which constraints need to be built. In Fig. 16, the number of constraint contexts required for each adapted metamodel element are depicted. Note that the processing time for a single instantiation is significantly above average in Fig. 15 for those metamodel elements that require constraints being generated for a large number of contexts. However, Fig. 16 also shows that in the UML metamodel most changes only required the generation of a single constraint per template instantiation. Moreover, as depicted in Fig. 17 where the time required for a single template instantiation is drawn as a function of the number of constraint contexts, the actual effect of the number of rule contexts on the instantiation

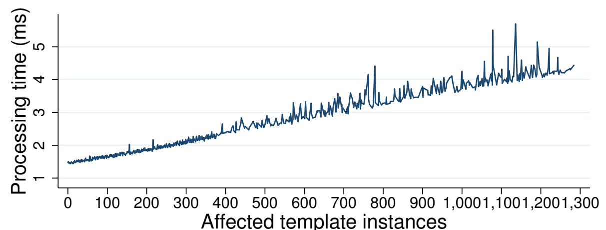

Figure 18: Total processing time for metamodel element removal depending on evolution impact.

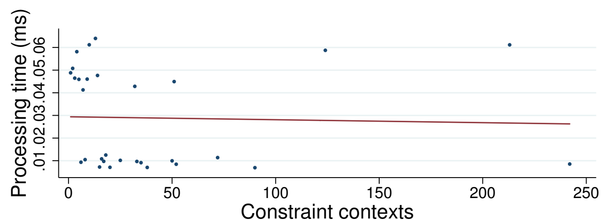

Figure 19: Processing time for metamodel element removal per affected instance depending on constraint contexts.

time is negligible in practice.

6.5.2. Remove metamodel element

For the removal of a metamodel element, Fig. 18 depicts the total processing time for handling the change depending on the number of template instantiations that become obsolete. As with element addition, the time for processing element removal does increase linearly with the evolution impact. However, note that the overall processing times (i.e., the time required for finding existing template instantiations that become obsolete) are much lower than times for element addition. This is explained by the fact that during instantiation, values must be read from the metamodel, constraints are generated, and traces are built. For removal, however, previously generated traces are used to find obsolete instantiations efficiently. By using existing traces and supported by the fact that no new objects must be generated, the number of constraint contexts for obsolete constraints does not affect processing times significantly. In Fig. 19, the average processing times per obsolete template instantiation is depicted as a function of the constraint contexts for the corresponding constraint. Note that a) the processing time per affected template instance stays below 0.1 ms, and b) there is no correlation between constraint contexts and processing time, as indicated by the slope of the regression line.

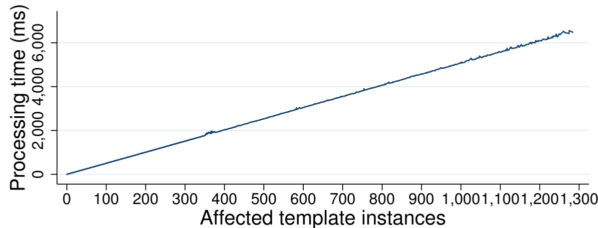

Figure 20: Total processing time for metamodel element update depending on evolution impact.

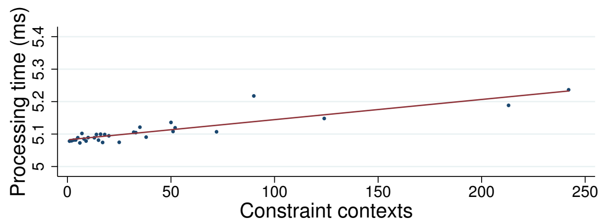

Figure 21: Processing time for metamodel element update per affected instance depending on rule contexts.

6.5.3. Update metamodel element

For the update of a metamodel element, the total processing time again was affected significantly by the number of affected template instantiation (i.e., template instantiations that had to be updated with new values), as shown in Fig. 20. Similar to the case of metamodel element addition, the observed growth is linear. Moreover, the total processing times do not differ significantly for a given change impact size between metamodel element addition and update. This is caused by the fact that the removal of outdated template instances does not contribute significantly to the total time for updating affected instantiations (as it only varies between 1 ms and 5 ms). The re-instantiation of an affected template does require the same steps as an initial instantiation except that for a re-instantiation the templates to instantiate are found based on existing traces instead of being looked up by their instantiation context. Therefore, the processing time per affected template instance does depend on the number of constraint contexts required, as shown in Fig. 21. However, the overall increase caused by an increase of constraint contexts was less than 2 ms.

6.5.4. Summary

The results for the individual evolution scenarios indicate that constraint management can be done efficiently and that our prototype implementation does scale well even for large change impact sizes. To sum up the case study results for the constraint management part, Table 4 shows the effects of different aspects on total processing times.

The results for a regression of the total processing time on the varied factors (i.e., template count, number of total template instantiations, number of scope elements, number of affected template instances, and number of rule contexts for affected constraints) confirms our expectation that the number of active templates as well as the total number of template instantiations are both insignificant and thus irrelevant for processing times. Even though the number of scope elements is statistically significant, its contribution to the total processing time is negligible. As observed above, the main contributors to total processing times are the rule contexts of affected constraints and primarily the number of affected template instantiations. Note that the considered factors explain more than 99% of variation for metamodel element addition and updating, and nearly 45% of variation for metamodel element removal. Overall, the results confirm our expectations and suggest that constraint management is scalable.

6.6. Consistency checking & repair options

Even though the focus of the case study was to evaluate constraint management, the developed prototype was used to check UML models with constraints that have been adapted after metamodel evolution. This was done primarily to demonstrate the integration of the consistency checker with the constraint management component.

After the introduction of new attributes and references in the UML metamodel and the generation of a corresponding constraint, the employed incremental consistency checker did find inconsistencies in the checked UML models, indicating that co-evolution is required. Since the employed Model/-Analyzer consistency checker has been thoroughly validated before and our case study with a slightly adapted version did not reveal any new information or significant differences to previously observed behavior, we refer to Reder and Egyed (2010, 2012b) for further details on this component's scalability and performance.

As with the consistency checking part, we employed an existing technology, proposed by Reder and Egyed (2012a). For each inconsistency detected in a model after a metamodel evolution and the corresponding update of applied constraints, an abstract repair (i.e., a repair is non-executable but

Table 4: Regression results for total processing times.

Time (ms)	Add	Remove	Update
Templates	0.0212	0.000129	-0.00674
	(0.0260)	(0.000251)	(0.0243)
Template instances	-0.000143	5.87e-08	-0.0000149
	(0.0000736)	(0.000000710)	(0.0000687)
Scope elements	0.000396***	0.00000250***	0.000190**
	(0.0000709)	(0.000000684)	(0.0000662)
Affected instances	5.108***	0.00173***	5.101***
	(0.0271)	(0.000261)	(0.0253)
Rule contexts	0.217***	0.000416***	0.296***
	(0.00958)	(0.0000924)	(0.00894)
Constant	-6.785***	1.448***	-8.672***
	(0.661)	(0.00637)	(0.617)
Observations	91391	91391	91391
R^2	0.997	0.448	0.997

Standard errors in parentheses
* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

still guides a tool user) was generated. The times required for finding repairs and also the behavior in terms of scalability were consistent with those reported in Reder and Egyed (2012a), thus we omit a detailed discussion here. However, note that repairs are generated within milliseconds per inconsistency.

Overall, the observed results for consistency checking and repair generation demonstrate by example that co-evolution through CCP is technically feasible and that chaining the individual steps does not lead to significant overheads at either step.

7. Discussion

Next, we further discuss the performance results obtained from our case study, the applicability of our approach, and also possible threats to validity.

7.1. Case study results

As we have briefly mentioned above, a single metamodel element addition or removal can only cause a single instantiation per managed template to be affected. The large sizes of change impacts in our case study may therefore be unlikely to be reached in practice, as a single template produces a family of constraints. Although a typical number of required templates cannot be given – as this depends not only on the specific metamodel and the characteristics that consistent models should have, but also on how templates, and the constraints they produce, are written – we believe that typically very few templates are sufficient to produce constraints that check metamodel conformance. For instance, if in our example the correct types of references should also be checked, this could be achieved in two ways: 1) by adding the required OCL statements to the constraints that are produced by $T1$ (i.e., add information to the abstract constraint expression and define new variables and extraction expressions), or 2) by writing a new template that produces type-checking constraints.

With a rather small set of templates being sufficient for typical conformance checking, we expect that most change impacts for element addition and removal remain smaller than 100 template instances (which already requires at least 100 active templates). For a change impact with exactly 100 affected template instances, mean total processing times for constraint management of 510 ms and 1.7 ms have been observed for element addition and removal, respectively. An update of a metamodel element may, however, cause change impacts larger than the number of templates because a sin-

gle metamodel element may – in the most extreme case – be used during each instantiation of each template. Thus, in the worst case scenario each existing template instantiation must be updated after a metamodel element update. However, even in this unlikely case the results of our case study suggest that processing time grows linearly and thus scales.

7.2. Applicability

We have illustrated how our approach updates constraints and derives options for correcting co-evolution failures. Although we have used the proposed solution in isolation to keep the example simple and focused, it is compatible with existing automatic co-evolution techniques. When used in isolation, our approach detects the absence of necessary model adaptations as co-evolution failures. When combined with other approaches, it also detects co-evolution failures that are based on incorrect model adaptations. Therefore, our solution is not a substitute but a complement to existing technologies.

7.2.1. Metamodels and languages

The prototype implementation is generic and supports arbitrary metamodels and models; the employed consistency checker uses OCL constraints. However, by using a different consistency checking technology, other constraint languages may be used. The generation of repair actions based on inconsistencies and with consideration of side effects may also be exchanged. Thus, the presented approach is generic and can be implemented using different technologies.

The presented approach is applicable to any metamodel that imposes constraints onto conforming models. Note that the generated constraints are very specific in that they include metamodel information in the actual constraint text. Moreover, they are defined for specific kinds of model elements (e.g., communications in our motivating example). Thus, our approach does not require a model to provide navigation functionality that allows the consistency checker to read metamodel values directly during constraint validation. Furthermore, the specific constraints generated by our approach do support incremental and thus efficient re-validation of individual constraints on individual model elements, compared to generic approaches were a single constraint checks multiple aspects of a model element. In our motivating example, for instance, a generic constraint might check whether the `Communication` does provide all the attributes and references that are defined in the metamodel. Typically, such a generic constraint has to be re-validated each time the attributes and references provided by an instance of `Communication` are changed. With a specific constraint, as it is generated with our proposed template approach, however, only those constraints that are actually affected by the change in attributes and references have to be re-validated.

7.2.2. Kinds of constraints

In the running example and also the conducted case study, we focused on syntax constraints. However, our approach also supports the evolution of language semantics and the corresponding constraints.

Static semantics of a modeling language can typically be expressed as constraints (e.g., Object Management Group, 2013b; Egyed, 2011). For instance, several well-formedness rules exist for the UML that reach far beyond simple syntax constraints (Egyed, 2011). Note that also these semantics constraints may depend on metamodel information. As an example, consider a constraint that requires messages in UML sequence diagrams to be reflected by operations in UML class diagrams. This can be expressed as follows in OCL:

```
context Message inv:
self.receiveEvent.covered->forAll(x|
x.represents.type.ownedOperation->exists(y|
y.name=self.name))
```

If the metamodel attribute `Message.receiveEvent` is changed to `Message.receiver`, for instance, this requires an update of the constraint to:

```
context Message inv:
self.receiver.covered->forAll(x|
x.represents.type.ownedOperation->exists(y|
y.name=self.name))
```

Handling such changes with our proposed constraint management approach is indeed possible, as it was demonstrated in (Demuth et al., 2013b). When using the templating approach presented in this paper, a *singleton template* can be used that is instantiated exactly once to produce the desired semantic constraint. Such a singleton template instantiation provides a constraint scope that is used to determine whether a metamodel change requires a re-instantiation.

It is of course also possible that metamodel semantics are evolved manually by adding, adapting, or removing existing semantic constraints that are not produced through template instantiation. In this case, the consistency checker handles the incremental update to the set of applied constraints. If this update of constraints leads to inconsistencies, co-evolution of the model is required in order to (re-)establish well-formedness. Note that this is not different from handling syntax changes. Thus, we omitted this scenario in our case study.

Of course, for changes of the environment the argument is valid as well. If the environment changes, so must environment specific constraints. Even though such constraints may be managed by our constraint management, this is beyond the scope of this paper. However, we refer to Demuth et al. (2013b) for details about how environment-based constraints can be updated automatically.

Overall, the only requirement to apply our approach of co-evolution through CCP is that there must be some constraints imposed by a metamodel on models. However, in a well-established model-driven engineering process such constraints should exist anyway to ensure correctness and validity of models. Therefore we argue that often only minimal effort is needed to implement the proposed approach in a development process and to migrate existing models; especially when compared to other approaches that always require migration strategies to be developed.

7.3. Benefits of co-evolution through CCP

As we have discussed in Section 3.1, co-evolution of metamodels and models is an issue that has been partially addressed by various approaches. While our case study has shown that our approach handles evolution efficiently and derives repairs within milliseconds, the major benefits of our approach over existing work is of qualitative nature. To our knowledge, there are no other approaches that consider constraints imposed by metamodel syntax or semantics and also environment-based constraints for co-evolving models. Existing approaches either derive model adaptations from metamodel changes or they execute user-definable model migration strategies. Both kinds of approaches do not necessarily consider metamodel semantics or evironmental constraints. Hence, they may produce updated models that are syntactically correct but are still not valid with respect to language semantics or the modeling environment. However, all three aspects are considered automatically when our approach of co-evolution through CCP is employed.

Moreover, our approach does not prescribe a single strategy that updates models, but instead it provides guidance for semi-manual model adaptation. In doing so, premature design decisions that have to be made during authoring of transformation rules or migration strategies are avoided and situation-specific contexts can be considered when designers choose between possible repairs. Thus, we argue that the adaptations performed with our approach are of higher quality than adaptations performed with automatic approaches – it produces models that are not only valid with respect to syntax, semantics, and the environment, but that are also intended by designers.

7.4. Threats to validity

Although the feasibility and the applicability of our approach to a complex metamodel such as the UML has been demonstrated through the conducted case study, some threats to validity remain.

7.4.1. Usability

While our approach considers more information than other existing approaches when looking for model adaptations, it certainly requires active participation of designers that choose specific model adaptations to be performed. Although we have yet to assess the usability of co-evolution through CCP from a user's perspective, we argue that usability can generally be achieved in common useage scenarios by combining the approach with existing technologies. There are several factors which determine the usability of our approach for a specific evolution: the number of inconsistencies, the number of repair options per inconsistency, and the effort needed for a developer to evaluate different repair options and selected the most appropriate one. Without employing other technologies, the usability of our approach declines with increasing numbers of inconsistencies. However, studies (Herrmannsdoerfer et al., 2009b, e.g.) have shown that for individual metamodel changes often only a small number of model adaptation is necessary (i.e., less than 100). This indicates that our approach is usable in practice. Indeed, if a metamodel evolution causes a large number of inconsistencies (e.g., hundreds), fixing each inconsistency in a guided manner by selecting one of several repair options might seem impractical. However, our approach can be chained

with automatic repair option execution. For example, it is possible that designers fix several inconsistencies in a semi-automatic manner by using provided guidance, and decide that the remaining inconsistencies should be fixed automatically. Indeed, executing one of the available repairs automatically for every remaining inconsistency is straightforward and can be done easily. However, note that in this case either a certain kind of repair can be chosen by a designer to be executed, or a strategy for finding the best repair for every inconsistency may be used. For example, a strategy may always execute the repair that requires the least atomic model changes. Either way, designers are initially informed about all points of co-evolution failures and can decide which parts of a model should be updated manually with guidance and which parts are less critical so that they can be updated automatically. In the most extreme case, a model may be co-evolved automatically by automatically executing repairs for all inconsistencies. Note that in this case user interaction is no longer required and the approach still produces an updated model that has, at least, a higher chance of being valid compared to other co-evolution approaches because of the additional information that is used for computing repair options (i.e., environment-based constraints).

For those inconsistencies that ought to be fixed in a guided manner by designers, the number of repair options affects usability. For common inconsistencies in software design models we have found in previous work that the number of repair options per inconsistency remains rather small and can be handled easily by designers (Reder and Egyed, 2012a). For inconsistencies with a large number of repair options, multiple similar concrete repairs (i.e., repairs that can be executed without further input) can typically be combined into as abstract repairs that requires some sort of user input (e.g., defining a specific value, such as a class name). For inconsistencies with concrete repair options, the effort required for designers to choose the most appropriate option depends not only on the specific metamodel but also on the specific project.

Overall, we believe that usability depends primarily on how information is presented to designers and which technologies are employed for further automation. However, a case study regarding usability in realistic modeling projects with evolving metamodels is part of future work.

7.4.2. Generalizability of results

Since the case study presented in the paper relies on a single metamodel that was evolved and a limited set of two kinds of templates that were instantiated, the question whether the observed results are generalizable can be raised. However, in our case study we systematically varied relevant factors such as the number of applied templates or the actual impact size of a metamodel change.

The metamodel size and complexity is not relevant for scalability or performance as handling metamodel changes by the template engine never requires searches of the metamodel. Specifically, for new metamodel elements, only the contexts of active templates are relevant. For removed metamodel elements, only template instantiations that are based on the removed elements are of interest. For updates metamodel elements, only those template instantiations with affected scopes are relevant. Thus, a search of the metamodel is not necessary to handle metamodel changes.

Indeed, the complexity of the metamodel may affect template instantiation and scope sizes because in metamodels with higher complexity it may require more complex data extraction expressions to obtain specific values for variables. However, the scope of an instantiation is composed of individual scope elements (a metamodel element and a property that is accessed). Those individual scope elements are also stored individually and traces between each scope element and the template instantiations that rely on it are maintained incrementally by the template engine. Since a single metamodel element update corresponds to exactly one scope element (i.e., the scope element that reflects the metamodel element and its changed property), finding affected template instantiations does not depend on the complexity of data extraction expressions. Of course, more managed scope elements may increase the time for finding the single scope element that corresponds to a metamodel change. However, the number of overall scope elements was varied systematically in the case study and no correlation between processing times and the number of scope elements was observed. Moreover, our prototype uses a unified method for accessing arbitrary metamodels and models. Thus, we argue that the relevant factors of variation are considered in our case study and that our findings can be generalized.

7.4.3. Performed metamodel changes

The changes we performed only involved the removal, addition, and updating of individual references and attributes in the UML metamodel. We did not consider more complex changes such as, for example, the addition of new classes or the removal of existing ones in the UML metamodel. Neither did we consider typical metamodel refactorings. However, those metamodel changes can also be performed through a series of atomic changes that are equivalent to the ones we used in our case study. Thus, processing times for complex metamodel changes and refactorings should depend primarily on the number of atomic changes that is necessary to reach the desired metamodel state.

7.4.4. Repair option generation

For the generation of repairs, we focused on generating abstract repairs that present to tool users general guidelines on how to fix an inconsistency instead of a list of executable (concrete) repair options. However, the mechanisms employed for constraint generation are the same for both abstract and concrete repairs. Since we focused on metamodel-based constraints and did not consider evironment-based constraints in our case study, the effects of existing constraints on the number of concrete repair options for inconsistencies were not considered. The analysis of such effects will be the focus of future investigations. However, previous research about the effects of constraint dependencies has shown that generally the number of concrete repair options in UML models does decrease significantly when multiple constraints restrict single model elements (Nöhrer et al., 2011). Moreover, research has shown that the time needed for generating repairs is not affected significantly by the model size or the total number of applied constraints (Reder and Egyed, 2012a).

8. 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 by 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. (2005) as one of the major challenges in software evolution. Since then, various approaches have been proposed to deal with this challenge. As co-evolution is a very active field of research and there has been significant effort made to describe the field in its entirety by Rose et al. (2012), the discussion here remains incomplete on purpose and focuses on those approaches only that are closest to ours.

Wachsmuth (2007) addresses the issue of metamodel changes by describing them as transformational adaptations that are performed stepwise instead of big, manually performed ad hoc changes. 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. (2009) classify possible metamodel changes and decompose differences between model versions into sets of changes of the same modification-class. 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. (2008) also classified coupled metamodel changes and investigated how far different adaptations are automatable. *COPE* (Herrmannsdoerfer et al., 2009a) handles atomic metamodel changes through reusable model operations that are executed automatically whenever specific metamodel changes occur. 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 designers to perform adaptations of a model with guidance based on specific constraints, including constraints that are based on sources other than the metamodel, and their own domain knowledge.

Wimmer et al. (2010) follow a different approach by merging two versions of a metamodel to a *unified metamodel* and then applying co-evolution rules to the models . They instantiate new metaclasses and remove existing elements that are no longer needed. At first, they encountered problems regarding typecasts and instantiation so they had to change some

co-evolution rules. Our prototype can handle the instantiation of created metaclasses as well as arbitrary typecasts of instances.

Rose et al. (2010b) presented *Epsilon Flock*, a language which mixes declarative and imperative parts and that is designed specifically for defining and executing model migration strategies. Jakumeit et al. (2010) proposed the *GrGen.NET* graph rewrite system that enables convenient model migration based on user-definable rules. Narayanan et al. (2009) presented the *Model Change Language (MCL)*, a language that allows for simply modeling of model changes. They use the MCL to define metamodel changes. Existing instance models are then migrated automatically based on these changes. Di Ruscio et al. (2011) proposed *EMFMigrate*, an approach that lets users define migration rules that are triggered by, also user-definable, metamodel changes. In contrast to other approaches, EMFMigrate supports model migration of different development artifacts in a uniform way by relying on migration libraries. Each migration library is related to a given metamodel. Our approach of co-evolution through CCP does also support arbitrary metamodels and handles arbitrary metamodel changes by default. Moreover, note that those approaches rely on user-defined migration rules or strategies that have to be written manually. Without additional verification and validation, which is one of the main challenges in model transformation, writing such rules or strategies manually is error-prone. Furthermore, those strategies do not nesessarily consider any semantic or environment-based constraints at all. Our approach, to the contrary, considers such aspects by default and does not require any transformations to be written manually after a metamodel evolution. Moreover, any proposed repair options are safe to execute without causing semantics or other constraints to be violated.

General-purpose transformation approaches such as ATL (OBEO and INRIA, 2014) or QVT (Object Management Group, 2014) are also commonly used for performing model migration. However, using these technologies imposes the same drawbacks as do the migration-specific transformation approaches mentioned above.

In terms of constraint co-evolution, Büttner et al. (2005) discuss various metamodel modifications and how they affected constraints. 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.

Constraint generation. Some approaches have been proposed to generate constraints automatically. Previously, we have shown that traditional model transformation approaches can be used to generate model constraints automatically, which is called *Constraint-driven Modeling (CDM)* (Demuth et al., 2013b). In particular, constraints that express dependencies between design models can be generated through model transformation. A significant difference between CDM and co-evolution through CCP is the focus of application. While the former focuses on consistency issues between (and within) design models and how constraint generation can address typical model transformation issues such as rule scheduling or model merging, the latter handles co-evolution between metamodels and models. Moreover, we propose to use constraint templates and an efficient constraint management engine in this paper, whereas CDM promotes the use of standard model transformation techniques. While model transformations may also be used in the co-evolution scenario, the presented template-based approach is highly efficient, as it avoids issues traditional transformation engines have to address (e.g., scheduling the execution of transformation rules), and does not require metamodel designers to learn a transformation language.

The major difference between our proposed constraint management and other constraint-generating approaches such as, for example, the ones presented by Büttner et al. (2012) or Cabot et al. (2010) is that the goals of these approaches are different from ours. Those approaches typically generate constraints in order to do verification or validation of transformation rules through sophisticated reasoning over those rules and target models (e.g., derive an optimal order of execution for a set of transformation rules). Thus, they are designed for different problems (e.g., ensuring syntactical correctness of target models or checking validity of transformation rules).

Consistency checking. The concept of incremental consistency checking is getting much attention because of the increasing popularity of software product lines and their large and complex

variability models that make batch-validation of constraints a task that cannot be performed on-the-fly anymore (Vierhauser et al., 2010), our previous work on the Model/Analyzer (Reder and Egyed, 2010) that we leveraged for our prototype has also been adapted to product lines to address this issue. Our work extends the performance benefits of incremental consistency checking by the automation of constraint generation that makes it possible to define templates for constraint generation once and then let the template engine do the work of updating the constraints as required.

Repair option generation. Although we used an adapted version of the approach presented by Reder and Egyed (2012a) in prototype, any technology that generates repairs for inconsistent models, such as those presented by Nentwich et al. (2003), da Silva et al. (2010), or Xiong et al. (2009) may be used instead. We opted for the approach by Reder and Egyed (2012a) because there was an implementation available that was built for usage with the employed incremental consistency checker. Moreover, this approach does not require fixing-related statements to be added to the applied OCL constraints, as it is in Xiong et al. (2009). Neither does it try to execute adaptations automatically as it is proposed by da Silva et al. (2010) and Xiong et al. (2009).

Flexible and multilevel modeling. Atkinson and Kühne (2001) identified several issues in the field of multilevel (meta)modeling, namely the so-called *shallow instantiation* of the UML that was a motivation for us to rely our implementation on a data storage that integrates arbitrary metamodels and models and provides the flexibility required for flexible, multilevel modeling. 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 implementation's underlying representation of arbitrary metamodels and models. Ossher et al. lately presented the *BITKit* tool (Ossher et al., 2010) 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.

9. Conclusion and future work

In this paper, we have presented the outline of a novel approach for supporting the co-evolution of metamodels and models. The paper introduced the general concept of consistent change propagation and illustrates how this concept is tailored to the problem domain of co-evolving models. Our approach is generic and relies on the detection of inconsistencies that occur after metamodel evolution. Those inconsistencies, which can be detected reliably because of incremental updates of constraints that ensure an up-to-date set of constraints being used at all times, serve as input for a reasoning mechanism that provides as output a set of possible model adaptations for repairing – that is, co-evolving – an affected model. The benefits of using CCP over traditional approaches – for example, the avoidance of unintended model adaptations produced by fully automatic approaches – were illustrated by a running example. The prototype implementation demonstrates that the approach is feasible and good performance results with realistic case studies suggest that it also works efficiently.

For future work we plan to apply the generic concept of CCP to other problems such as versioning systems in which inconsistencies are introduced and have to be resolved frequently. Moreover, we plan to conduct experiments that evaluate in detail the benefits of our approach from a designer's perspective.

Acknowledgments

The research was funded by the Austrian Science Fund (FWF): P25289-N15 and P25513-N15, FWF Lise-Meitner Fellowship M1421-N15, and the Austrian Center of Competence in Mechatronics (ACCM).

References

Atkinson, C., Kühne, T., 2001. The essence of multilevel metamodeling. In: UML. pp. 19–33.

Blanc, X., Mougenot, A., Mounier, I., Mens, T., 2009. Incremental detection of model inconsistencies based on model operations. In: CAiSE. pp. 32–46.

Büttner, F., Bauerdick, H., Gogolla, M., 2005. Towards transformation of integrity constraints and database states. In: DEXA Workshops. pp. 823–828.

Büttner, F., Egea, M., Cabot, J., Gogolla, M., 2012. Verification of ATL transformations using transformation models and model finders. In: ICFEM. pp. 198–213.

Cabot, J., Clarisó, R., Guerra, E., de Lara, J., 2010. Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83 (2), 283–302.

Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A., 2008. Automating co-evolution in model-driven engineering. In: EDOC. pp. 222–231.

Cicchetti, A., Di Ruscio, D., Pierantonio, A., 2009. Managing dependent changes in coupled evolution. In: ICMT. pp. 35–51.

da Silva, M. A. A., Mougenot, A., Blanc, X., Bendraou, R., 2010. Towards automated inconsistency handling in design models. In: CAiSE. pp. 348–362.

Demuth, A., Lopez-Herrejon, R. E., Egyed, A., 2013a. Co-evolution of metamodels and models through consistent change propagation. In: ME@MoDELS. pp. 14–21.

Demuth, A., Lopez-Herrejon, R. E., Egyed, A., 2013b. Constraint-driven modeling through transformation. Software and Systems ModelingDOI: 10.1007/s10270-013-0363-3.

Demuth, A., Lopez-Herrejon, R. E., Egyed, A., 2013c. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In: MoDELS. pp. 287–303.

Di Ruscio, D., Iovino, L., Pierantonio, A., 2011. What is needed for managing co-evolution in mde? In: IWMCP. IWMCP '11. ACM, New York, NY, USA, pp. 30–38. URL http://doi.acm.org/10.1145/2000410.2000416

Di Ruscio, D., Iovino, L., Pierantonio, A., 2012. Coupled evolution in model-driven engineering. IEEE Software 29 (6), 78–84.

Egyed, A., 2006. Instant consistency checking for the UML. In: ICSE. pp. 381–390.

Egyed, A., 2011. Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Software Eng. 37 (2), 188–204.

Eramo, R., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A., 2012. A model-driven approach to automate the propagation of changes among architecture description languages. Software and Systems Modeling 11 (1), 29–53.

France, R. B., Rumpe, B., 2007. Model-driven development of complex software: A research roadmap. In: FOSE. pp. 37–54.

Groher, I., Reder, A., Egyed, A., 2010. Incremental consistency checking of dynamic constraints. In: FASE. pp. 203–217.

Hassam, K., Sadou, S., Gloahec, V. L., Fleurquin, R., 2011. Assistance system for OCL constraints adaptation during metamodel evolution. In: CSMR. pp. 151–160.

Hassan, A. E., Holt, R. C., 2004. Predicting change propagation in software systems. In: ICSM. pp. 284–293.

Herrmannsdoerfer, M., Benz, S., Jürgens, E., 2008. Automatability of coupled evolution of metamodels and models in practice. In: MoDELS. pp. 645–659.

Herrmannsdoerfer, M., Benz, S., Jürgens, E., 2009a. COPE - automating coupled evolution of metamodels and models. In: ECOOP. pp. 52–76.

Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G., 2009b. Language evolution in practice: The history of gmf. In: SLE. pp. 3–22.

Iovino, L., Pierantonio, A., Malavolta, I., 2012. On the impact significance of metamodel evolution in mde. Journal of Object Technology 11 (3), 3: 1–33.

Jakumeit, E., Buchwald, S., Kroll, M., 2010. Grgen.net - the expressive, convenient and fast graph rewrite system. STTT 12 (3-4), 263–271.

Lange, C. F. J., 2006. Improving the quality of uml models in practice. In: ICSE. pp. 993–996.

Manders, E.-J., Biswas, G., Mahadevan, N., Karsai, G., 2006. Component-oriented modeling of hybrid dynamic systems using the generic modeling environment. In: MBD/MOMPES. pp. 159–168.

Markovic, S., Baar, T., 2005. Refactoring OCL annotated UML class diagrams. In: MoDELS. pp. 280–294.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M., 2005. Challenges in software evolution. In: IWPSE. pp. 13–22.

Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G., 2009. Automatic domain model migration to manage metamodel evolution. In: MoDELS. pp. 706–711.

Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A., 2002. xlinkit: a consistency checking and smart link generation service. ACM Trans. Internet Techn. 2 (2), 151–185.

Nentwich, C., Emmerich, W., Finkelstein, A., 2003. Consistency management with repair actions. In: ICSE. pp. 455–464.

Nöhrer, A., Reder, A., Egyed, A., 2011. Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track. In: ICSE. pp. 864–867.

OBEO, INRIA, 2014. ATLAS transformation language (ATL). http://www.eclipse.org/atl/.

Object Management Group, 2013a. Object Constraint Language (OCL). http://www.omg.org/spec/OCL/.

Object Management Group, 2013b. Unified Modeling Language (UML). http://www.uml.org/.

Object Management Group, 2014. Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/. URL http://www.omg.org/spec/QVT/

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

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

Reder, A., Egyed, A., 2012a. Computing repair trees for resolving inconsistencies in design models. In: ASE. pp. 220–229.

Reder, A., Egyed, A., 2012b. Incremental consistency checking for complex design rules and larger model changes. In: MoDELS. pp. 202–218.

Rose, L. M., Herrmannsdoerfer, M., Mazanek, S., Van Gorp, P., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B., et al., 2012. Graph and model transformation tools for model migration. Software and Systems Modeling, 1–37.

Rose, L. M., Kolovos, D. S., Paige, R. F., Polack, F., 2010a. Migrating activity diagrams with epsilon flock. Transformation Tool Contest, 30.

Rose, L. M., Kolovos, D. S., Paige, R. F., Polack, F. A. C., 2009. Enhanced automation for managing model and metamodel inconsistency. In: ASE. pp. 545–549.

Rose, L. M., Kolovos, D. S., Paige, R. F., Polack, F. A. C., 2010b. Model migration with epsilon flock. In: ICMT. Springer, pp. 184–198.

Schmidt, D. C., 2006. Guest editor's introduction: Model-

driven engineering. IEEE Computer 39 (2), 25–31.
Sprinkle, J., Karsai, G., 2004. A domain-specific visual language for domain model evolution. J. Vis. Lang. Comput. 15 (3-4), 291–307.
Sunyé, G., Pollet, D., Traon, Y. L., Jézéquel, J.-M., 2001. Refactoring UML models. In: UML. pp. 134–148.
Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W., 2010. Flexible and scalable consistency checking on product line variability models. In: ASE. pp. 63–72.
Wachsmuth, G., 2007. Metamodel adaptation and model co-adaptation. In: ECOOP. pp. 600–624.
Wimmer, M., Kusel, A., Schönböck, J., Retschitzegger, W., Schwinger, W., Kappel, G., 2010. On using inplace transformations for model co-evolution. In: MtATL. INRIA & Ecole des Mines de Nantes.
Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H., 2009. Supporting automatic model inconsistency fixing. In: ESEC/SIGSOFT FSE. pp. 315–324.