Softw Syst Model (2015) 14:573–596
DOI 10.1007/s10270-013-0363-3

CrossMark

SPECIAL SECTION PAPER

Constraint-driven modeling through transformation

Andreas Demuth · **Roberto Erick Lopez-Herrejon** · **Alexander Egyed**

Received: 15 October 2012 / Revised: 5 June 2013 / Accepted: 16 June 2013 / Published online: 27 June 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract In model-driven software engineering, model transformation plays a key role for automatically generating and updating models. Transformation rules define how source model elements are to be transformed into target model elements. However, defining transformation rules is a complex task, especially in situations where semantic differences or incompleteness allow for alternative interpretations or where models change continuously before and after transformation. This paper proposes constraint-driven modeling where transformation is used to generate constraints on the target model rather than the target model itself. We evaluated the approach on three case studies that address the above difficulties and other common transformation issues. We also developed a proof-of-concept implementation that demonstrates its feasibility. The implementation suggests that constraint-driven transformation is an efficient and scalable alternative and/or complement to traditional transformation.

Keywords Model-driven engineering (MDE) · Model transformation · Ambiguity · Consistency checking · Incremental constraint management

1 Introduction

With the broadening use of *Model-Driven Engineering (MDE)* [1] for complex software systems, the generation of models from existing artifacts through *model transformation* [2] is a vital necessity. Various classifications and taxonomies have been published to compare the state of the art (e.g., [3,4]). Rich transformation languages are available, such as ATL [5] or QVT [6], which define *transformation rules* that are executed by a *transformation engine*. The transformation engine generates a target model from a source model, typically through a series of transformation rules. To date, various sophisticated transformation techniques exist that produce excellent results as long as the generated models are static and there are no uncertainties [7,8]. However, model transformation has yet to overcome several key challenges [4,9,10]:

- How to allow transformation without overwriting changes made by the designer?
- How to transform in the face of uncertainty where multiple target models satisfy a given source model?
- How to support bi-directional consistency where the direction of transformation is irrelevant?

Model transformation either generates a target model out of a source model if none existed or it overwrites the existing target model. The latter is only desired if overwriting the target model does not lead to information loss—a problem when the designer edited the target model prior to transformation. In an idealized model transformation scenario, the target model is generated but never modified after transformation (later re-transformation does not lead to loss), or the source model will not be modified after transformation (avoiding later re-transformation). There are some situations where models are stable enough for this idealized scenario to apply, but in context of iterative/incremental development, designers tend to change models continuously [7]. In such cases, re-transformation may seriously affect the designer's normal workflow [11,12] if they do overwrite changes made manually beforehand. Hence, incrementality is required to

Communicated by Prof. Juan de Lara and Prof. Zhenjiang Hu.

A. Demuth (✉) · R. E. Lopez-Herrejon · A. Egyed
Institute for Systems Engineering and Automation,
Johannes Kepler University (JKU), Linz, Austria
e-mail: andreas.demuth@jku.at

المنارة للاستشارات
www.manaraa.com

allow partial model transformation in order to limit such conflicts [12].

The problem of re-transformation is also related to another problem: that of uncertainty. Manual changes to a target model are necessary if the information present in the source model is insufficient to generate the target model. There are two possible reasons for this: (1) the source model is incomplete, and/or (2) the semantic differences between the source and target models make it impossible to generate the target model in its entirety. Either reason implies uncertainty and current state of the art is either not used in these cases or the transformation implements a heuristic that automatically decides. This severely reduces the usefulness of transformation: either because model transformation is not applicable (leading to less automation) or because transformation generates a potentially incorrect target model (requiring manual addition and changes to the target model after transformation). The issue is similar with *bidirectional transformations* [10, 13], which are often used to synchronize models or to keep them consistent, when both involved models are edited concurrently. Overall, these issues limit the usefulness of transformation.

This paper proposes *Constraint-driven Modeling (CDM)*, a generic approach that guides the construction of new models while conserving consistency with the related models and eliminates issues arising from re-transformation, uncertainties, and bidirectionality. CDM does not generate a target model out of a source model rather CDM generates *constraints* from the source model that represent the invariants that the target models must satisfy. CDM thus does not overwrite the target model (avoiding information loss) and, as will be shown, constraints are an ideal vehicle to express uncertainty and incompleteness correctly. The generated constraints, written in a *constraint language* (e.g., the *Object Constraint Language (OCL)* [14]), are then continuously validated by a *constraint checker* on the target model. The constraint checker then provides the designer with feedback on constraint violations for *guidance*. Constraint violations in turn help the designer to stepwise transform the existing target model to a target model that satisfies all constraints. Indeed, CDM may also be chained with existing state-of-the-art technologies for repairing constraint violations—with or without designer interaction. These existing technologies provide guidance on what to repair and, at times, even how to repair in order to achieve a correct target model. Indeed, if the constraints are only satisfiable in a single target model, then these repairs should be able to compute this model. Here, the big advantage over traditional transformation is that these repair mechanisms do consider the state of the target model as part of the repair—including any manual changes made before or after the transformation.

Thus, CDM can be seen as a complement to traditional model transformation and model generation approaches; it could also be seen as an alternative in concert with repair technologies. Do note that CDM was first introduced in a conference paper [15], and this paper explains CDM in more detail and adds three case studies illustrating correctness, usefulness, and scalability.

We evaluated our approach and showed its feasibility by implementing a prototype that generates constraints, enforces them incrementally, and informs the designer about existing constraint violations (i.e., inconsistencies). It should be noted that CDM does not prescribe a particular constraint checking or repair mechanism. As a proof of concept, we incorporated one such mechanism, called the Model/Analyzer [16]. The Model/Analyzer is a scalable, incremental constraint checking mechanism, which has been validated on numerous large industrial models [17]. However, any other technology for constraint checking and repairing may be used instead. The use of the Model/Analyzer is merely to demonstrate that there is at least one such technology available already. Since the Model/Analyzer has a proven scalability record [17], the evaluation will focus on the correctness and scalability of the generation of target model constraints from a source model. We will show in a case study that the median times for incremental transformation are close to 1 ms for typical UML design models, and we will demonstrate that the subsequent constraint validation times are similar to those observed in [17]. In two additional case studies, we also demonstrated the applicability and efficiency of CDM in the domains of metamodeling and software product lines. Our findings show that regardless of the applied domain, the approach scales and also provides instant feedback when involved models are edited.

2 Running example

To illustrate our work, we first present a scenario that is challenging for common model transformation approaches. Let us consider the sequence and class diagrams shown in Fig. 1a, b, respectively. In Fig. 1a, the unnamed instance of class `LightSwitch` receives a message named `activate`. According to the semantics of UML sequence diagrams, this message requires that the instance of `Light Switch`

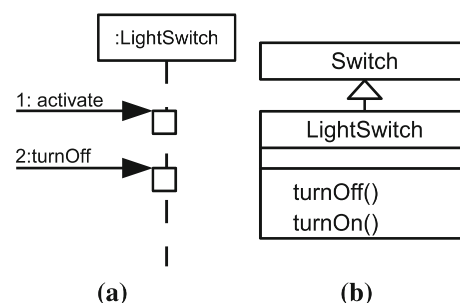


Fig. 1 Two UML models (**a**) and (**b**)

Springer
www.manaraa.com

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

Listing 1 Sample transformation to generate methods in class diagrams

provides a method named `activate` also [18,19]. At first glance, a simple transformation appears to solve the problem by automatically adding the method `activate` to class `LightSwitch` in Fig. 1b whenever a message is added to a sequence diagram whose name does not match any method in the class. An example for such a transformation rule written in ATL is shown in Listing 1. The rule is executed for every `Message` instance (defined in the `from`-block) for which it generates a `Method` instance (defined in the `to`-block). It assigns a name and an owner derived from the `Message` object for which it is executed to the attributes `name` and `owner` of the generated `Method`.

However, there is an issue with this approach: Should the method `activate` be added to `LightSwitch` or would it make more sense for the system to add it to the superclass `Switch`? Obviously, this question is hard to answer automatically. One possibility would be to make an assumption. For instance, always add the method to the target class, `LightSwitch` in our example. This option can lead to the generation of potentially unintended models where methods are not declared in the desired place or where methods are unnecessarily overridden. Another option is the use of heuristics to find the most suitable solution for the problem. However, heuristics typically employ metrics to evaluate possible solutions. Thus, heuristics may chose solutions that are optimal with respect to defined criteria but that may not be seen as optimal by designers.

3 Constraint-driven modeling

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

In contrast to standard model transformations, we propose to generate constraints on a model (to guide designers or enable automatic approaches that establish consistency) rather than generating the model itself whenever precise transformation results cannot be derived. For example, the added message `activate` on the source model should impose a constraint that a same named method should be available to class `LightSwitch` rather than saying it should be owned by it. If the method is already there, then the constraint is instantly satisfied. If the method does not exist, then the constraint's violation will identify this lack and further actions are required to deal with this problem—actions that must either come from a human or be derivable from the constraint violation (inconsistency) through reasoning (e.g., [20,21]).

When traditional model transformation approaches are used, the transformation process can be regarded as:

$$A \xrightarrow{T_m} B_g \quad (1)$$

where A is called the source model, consisting of an arbitrary number of model elements. T_m is a set of transformation rules (called the *transformation model*), which is used to transform A to the generated (denoted by the subscript $_g$) model B_g.

We expanded this notation and define our approach as:

$$A \xrightarrow{T_c} C \rightsquigarrow B_r \quad (2)$$

where the variable A denotes the source model and T_c is a set of model transformation rules. However, as the solid arrow from A to C and the changed subscript $_c$ of T indicate, the transformation model no longer generates a model (i.e., B_g), but instead it contains different transformation rules that are applied to A in order to generate a set of constraints, the constraint model C.[1] This constraint model consists of a set of constraints that are enforced by an incremental consistency checker on the model B_r, as indicated by the curvy arrow from C to B_r. The model B_r is no longer the generated model but is now called the restricted model, as indicated by the subscript r, which is either consistent or inconsistent with the constraint model C, and therefore a valid or invalid solution of the modeling problem.

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

3.1 Application: uncertainties

Let us come back to our running example from Sect. 2 where we illustrated that choosing the right class for a

[1] This is different from existing approaches (e.g., [22,23]) that derive constraints through interpretation or translation of transformation rules, which generate the target model. In CDM, transformation rules are executed by standard transformation engines to generate constraints—there is no interpretation or translation of rules involved.

المنارة للاستشارات
www.manaraa.com

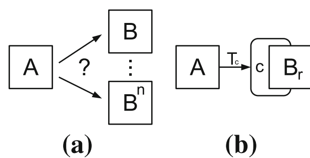


Fig. 2 From ambiguous model transformation (**a**) to constraint transformation (**b**)

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

Listing 2 Transformation rules to generate class (t1) and method (t2) constraints

required method cannot be fully automated. The traditional approach shown in Fig. 2a automatically generates one of several possible models, and we could at most use heuristics for deciding on which transformation to use (which never guarantees intended results). However, while the knowledge contained in Fig. 1a is insufficient to generate a correct update to the class diagram, it is sufficient to generate a correct constraint on that diagram. Such constraints can be generated incrementally by transformation rules whose execution is triggered by the addition/removal of class instances or messages in sequence diagrams that can be efficiently validated by state-of-the-art consistency checkers.

To automate constraint generation in our example, we define two transformation rules that are triggered by class instances or messages in sequence diagrams and that use information from the sequence diagram to generate very specific and expressive constraints. These rules are shown in Listing 2. The rule $t1$ takes an instance specification (e.g., a lifeline) and generates a constraints that requires the model's base package to provide a class with a matching name. Rule $t2$ is triggered by a message and generates a constraint that requires the message's receiver class to provide a method with a name equal to the message name. Note that, even though we use ATL-like syntax for this example, our approach can be used with any transformation language. After applying these rules to the motivating example from Sect. 2 as illustrated in Fig. 3 and according to Eq. (2), C consists of the following OCL constraints:

```
c1 context Package inv:
   self.classes->exists(c|c.name='LightSwitch')
c2 context Class inv:
   self.name='LightSwitch' implies
   self.providedMethods->exists(m|m.name=
   'activate')
c3 context Class inv:
   self.name='LightSwitch'implies
   self.providedMethods->exists(m|m.name=
   'turnOff')
```

In Fig. 3, we can see that the method required by the constraint $c2$ is not present in B_r, as indicated by the empty, dashed rectangle in the `LightSwitch` class, meaning that this particular model will be marked inconsistent. Note that, we use OCL as the constraint language in our example because it is a well-known and accepted language for writing constraints for design models and we have existing tool support for incrementally validating OCL constraints and repairing them. Nonetheless, in principle, any constraint language and consistency checker may be used (e.g., [24]).

Figure 2b illustrates the basic concept of the constraint-driven modeling approach. It is noteworthy that the approach does not modify the restricted model. It simply defines constraints that restrict it. The generated restriction—depicted as partial frame with rounded corners around the restricted model B_r—may be light in that there may be various options on how to change the restricted model to satisfy the constraints—hence supporting uncertainty. In such as case, the designer—or also a fully automated approach that uses heuristics—has the freedom to decide which of the options to select (e.g., add `activate` to `LightSwitch` or `Switch`) with the knowledge that the approach prevents options that are invalid. In the most extreme case, the restrictions may be severe enough to allow for one option only. In this case, the only remaining option could be chosen automatically, like in traditional transformation, to modify the target model B_r.

Figure 4 shows the visual notations we use throughout the paper to illustrate the differences between common model transformation and our approach. Figure 4a is equivalent to Eq. (1). Figure 4b shows the notations for our approach. Note that the constraint model C and the curved arrow to the restricted model in Eq. (2) are replaced by a partial, restricting frame around B; T_c is not depicted.

3.2 Incremental constraint model management

Let us take a closer look at the transformation that generates the constraint model C. As shown in Eq. (2) and Fig. 3, applying the transformation rules of the transformation model to the source model generates the constraint model.

المنارة للاستشارات
www.manaraa.com

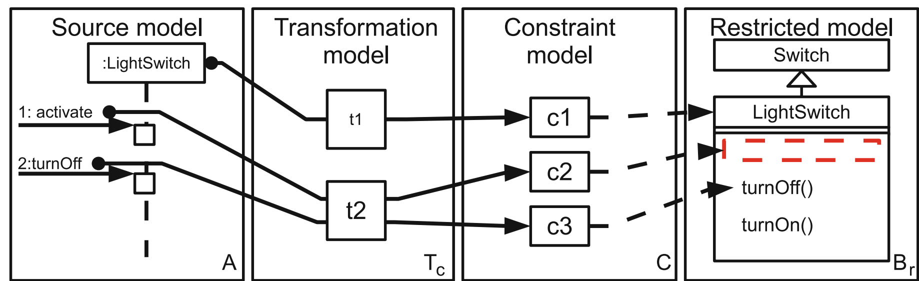


Fig. 3 Application of CDM to models from Fig. 1a, b

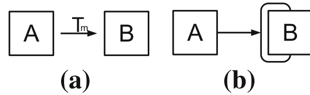


Fig. 4 Traditional (**a**) and constraining (**b**) approach

The transformation approach we use supports incrementality to allow updates of both the source model and the transformation model without performing a complete re-transformation.

3.2.1 Source model update

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

$$A \xrightarrow{\Delta A} A' \tag{3}$$

where ΔA is a sequence of modifications (i.e., ⟨model element, action $\in \{add, remove, update\}$⟩) done to A (e.g., add a new model element or update an existing element). Based on ΔA and the transformation model T_c, the set ΔC can be generated, as shown in Eq. (4).

$$\Delta A \xrightarrow{T_c} \Delta C \tag{4}$$

ΔC includes pairs of constraints and actions (i.e., ⟨constraint, action $\in \{add, remove\}$⟩) that define whether the constraint should be added or removed from the existing constraint model C. By applying ΔC on C, the updated constraint model C' is generated:

$$C \xrightarrow{\Delta C} C' \tag{5}$$

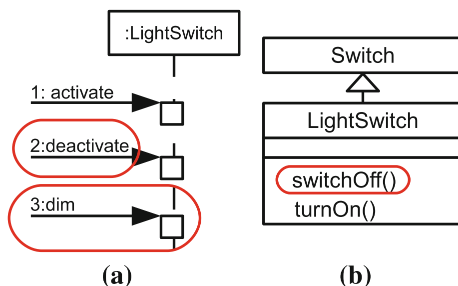


Fig. 5 Evolved versions of Fig. 1a, b

Let us consider the evolution of the models shown in Fig. 1a, b to the versions shown in Fig. 5a, b where the name of the message #2 was updated to `deactivate`, the message #3 was introduced, and the name of the method `turnOff` was changed to `switchOff`. For the changes in the source model, the corresponding ΔA is $\langle\langle Message2, update\rangle, \langle Message3, add\rangle\rangle$.

To build ΔC, the transformation engine executes those transformation rules that use elements in ΔA (i.e., message #2 and message #3) to generate the corresponding constraints, as defined in Eq. (4) and shown in Fig. 6.

For $\langle Message2, update\rangle$, the constraint $c3'$ is generated and the information $\langle c3', add\rangle$ is added to ΔC.

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

Since the constraint $c3$ was already generated from the same element as $c3'$, message #2, $\langle c3, remove\rangle$ is also added to ΔC in order to remove the now outdated constraint $c3$. For $\langle Message3, add\rangle$, the transformation rule $t2$ is executed to generate a new constraint $c4$ and $\langle c4, add\rangle$ is added to ΔC.

```
c4 context Class inv:
    self.name='LightSwitch'implies
    self.providedMethods->exists(m:Method|m.name
    ='dim')
```

Note that in this example, both elements in ΔA are used as context of a transformation. However, if an element in ΔA was accessed in any way during the execution of a transformation rule, the transformation also has to be re-executed (e.g., $c3$ would have been updated to $c3'$ also if not the message name, but the receiver's name would had changed). At this point, ΔC is $\{\langle c3, remove\rangle, \langle c3', add\rangle, \langle c4, add\rangle\}$.[2]

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

المنارة للاستشارات
www.manaraa.com

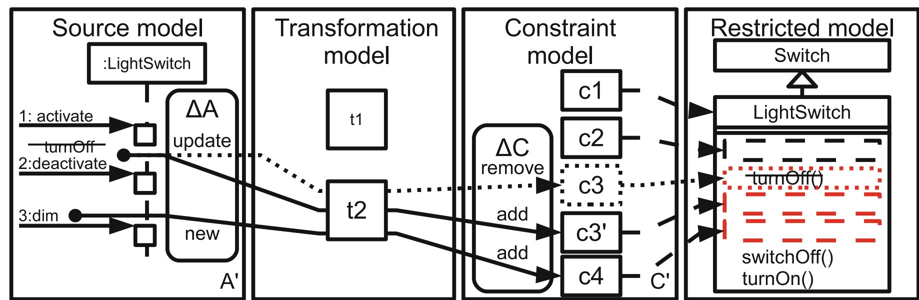


Fig. 6 Constraint model updates after source model changes

When these changes are applied to $C = \{c1, c2, c3\}$ as defined in Eq. (5) and shown in Fig. 6, the resulting updated constraint model is $C' = \{c1, c2, c3', c4\}$. We used dotted lines for removed elements that is $c3$ and the corresponding inconsistency in `LightSwitch`. As Fig. 6 indicates, the constraints $c3'$ and $c4$ are violated by the restricted model since the class `LightSwitch` does not provide the required methods `deactivate` and `dim`.

3.2.2 *Transformation model update.*

If the transformation model T_c is updated to $T_c^{'}$, the changes ΔT_c, which have the form $\langle$transformation rule, action $\in \{add, remove\}\rangle$, are derived and used to generate ΔC from A, as shown in Eq. (6).

$$A \xrightarrow{\Delta T_c} \Delta C \tag{6}$$

ΔC can then be used as shown in Eq. (5) to update the constraint model to C'.

Based on the model versions after the discussed source model update, let us consider further changes—this time to the transformation model—as depicted in Fig. 7: the removal of the transformation rule $t1$ and the addition of the new transformation rule $t3$, which is triggered by elements of the UML type `Instance`. ΔT in this case is $\langle\langle t1, remove\rangle, \langle t3, add\rangle\rangle$. For the removal of $t1$, the constraints that have been generated for this rule are removed from the constraint model and no transformation rules are executed. Since $c1$ is the only constraint generated by $t1$, $\langle c1, remove\rangle$ is added to ΔC. For the addition of $t3$, the transformation executes just this transformation rule and produces the new constraint $c5$ and adds $\langle c5, add\rangle$ to ΔC, which finally is $\{\langle c1, remove\rangle, \langle c5, add\rangle\}$.

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

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

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

$$C' \rightsquigarrow B_r \tag{7}$$

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

3.3 Constraint validation and solution space

We define the *solution domain* of a modeling problem to include all possible instances of a metamodel (there are likely infinite). The *solution space* includes all valid models of the solution domain. Constraints define certain criteria that valid models must meet. Validating a constraint on a specific model determines whether the model meets those criteria—if it does not, it is not part of the solution space. Therefore, applying a constraint decreases the size of the solution space. For example, Fig. 8a shows that the solution space in our running example is reduced to the specific models $B1$, $B2$, $B3$, and $B5$ (drawn as black filled circles) when the constraints $c1$–$c4$ are applied after the source model changes performed in Sect. 3.2.1. In our illustrations, we show the remaining solution space toward the center of the circle, and models outside the solution space are drawn as an unfilled circle.

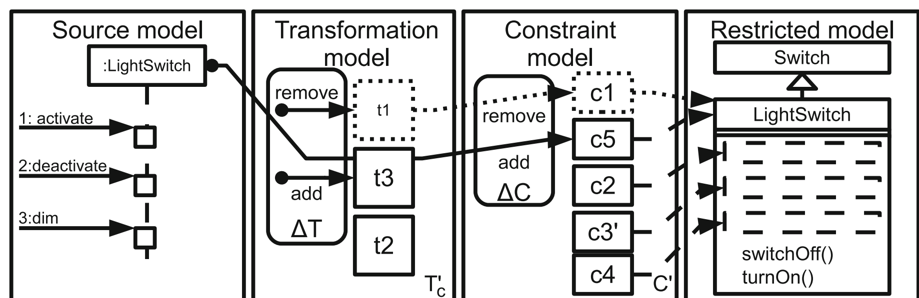


Fig. 7 Constraint model updates after transformation model changes

المنارة للاستشارات
www.manaraa.com

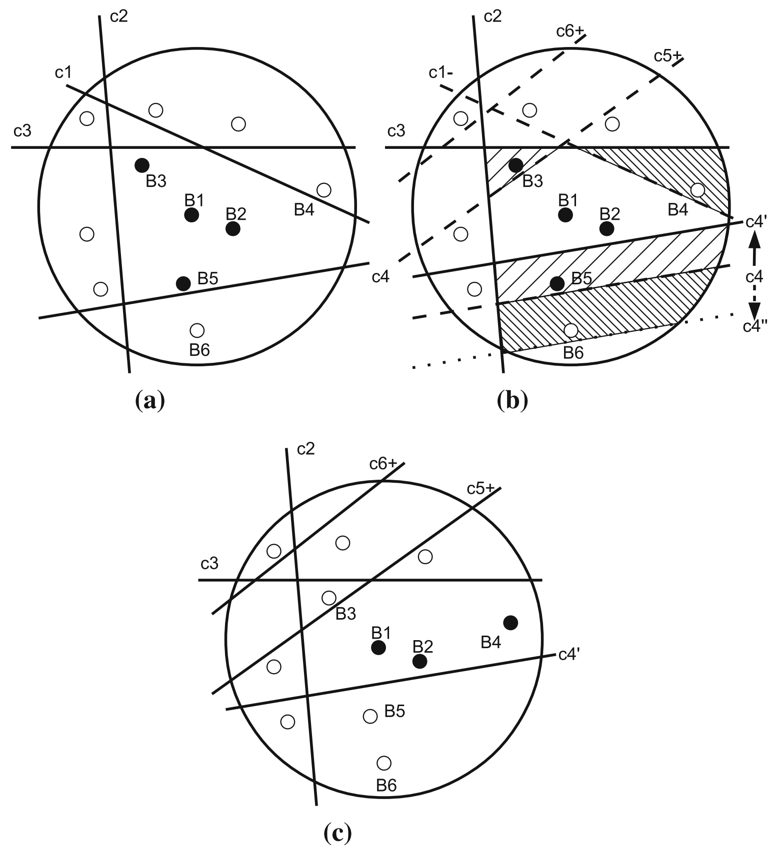


Fig. 8 Effects of constraint changes on solution space. **a** Before, **b** changes, **c** after

Note that this simplified view is only used to make the illustrations readable. Moreover, we assume that there is only a limited number of possible models for the illustration—in practice, the number of valid models may remain infinite.

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

$$val_T(B_r, C) = \bigwedge_{1 \leq i \leq |C|} val(B_r, c_i) \tag{8}$$

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

Constraints are composed of expressions that belong to exactly one constraint (i.e., information is not shared between constraints) and that are evaluated on the restricted model only. Therefore, *structural dependencies* between constraints do not exist and are not considered here. The addition of a new constraint thus does not affect the validity of existing constraints as they are not connected structurally. This leads us to the conclusion that **constraints are structurally independent of each other**.[3] Furthermore, the used transformation rules do only access the source model to construct constraints and add the constraint to the constraint model without accessing other constraint model elements, thus the **transformation rules for generating constraints are independent** and dependencies between them that require a certain order of execution cannot occur. These observations have interesting benefits to model transformation discussed next.

When the constraint model is changed, so need to be the restrictions imposed by it. There are three possible constraint model changes, which we will show next: (i) addition, (ii) removal, and (iii) update of constraints.[4]

3.3.1 Constraint addition

When new constraints are added to the consistency checker, the solution space either stays unmodified or is narrowed,

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

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

المنارة للاستشارات
www.manaraa.com

as illustrated in Fig. 8. The addition of $c5$ in Sect. 3.2.2, which requires an element of type `Class` and with the name `LightSwitch` must be owned by a `Package`, does reduce the remaining solution space (indicated by the thinly shaded area around $B3$ in Fig. 8b) so that the previously valid model $B3$ is removed from the solution space because the class `LightSwitch` did not provide such a method, as shown in Fig. 8c. The manual addition of a new constraint $c6$, which requires a UML `Package` to own at least one element, does not change the remaining solution space because there are other constraints (e.g., $c5$) that impose stronger restrictions than $c6$.

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

As Eq. (9) shows, we can easily split the complete evaluation of the new constraint model C' into the validation of the old constraint model C and only those parts of C' that were actually added (i.e., $C' \backslash C$).

$$\bigwedge_{h=1}^{|C'|} val(B_x, c'_h) = \bigwedge_{i=1}^{|C|} val(B_x, c_i) \wedge \bigwedge_{j=1}^{|C' \backslash C|} val(B_x, (C' \backslash C)_j) \quad (9)$$

Assuming that the validation result for C is still available, we can see that only the last part of Eq. (9) has to be evaluated to determine whether a specific model B_x is part of the updated solution space (i.e., a valid model). Obviously, this validation is not even relevant if the old validation result was already $true$.

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

3.3.2 Constraint removal

The removal of constraints can increase the size of the solution space. Figure 8 shows the removal of constraint $c1$, as discussed in Sect. 3.2.2. This change means that the model $B4$ becomes part of the solution space (as shown in Fig. 8c), which was increased by the densely shaded area around $B4$ in Fig. 8b. Intuitively, it is not possible that the removal of a constraint narrows the solution space. After constraints are removed, the restricted model is validated with the remaining constraints. As we have seen, the order of the constraints has no effect on the final validation result.

In contrast to the addition of constraints, the basic consistency checking mechanism we used in Eq. (8) cannot be split into parts to reduce calculation effort since we do not know the result of the remaining constraints. However, if we capture the constraints that were inconsistent during the validation of C, we can define a function $ic(B_x, C) = \{c | c \in C \wedge val(B_x, c) = false\}$ that returns exactly those constraints. After removing constraints from C to generate C', $C \backslash C'$ is the set of constraints that are removed from the constraint model. If $ic(B_x, C) \subseteq C \backslash C'$ holds, the model B_x is a valid solution after the constraint model updates since all constraints that previously caused an inconsistency are no longer part of the constraint model. Whether the function val_T or the set comparison is used to determine the new consistency status of a model, either way the order of removal does not affect the final result.

3.3.3 Constraint update

We have defined that a constraint update consists of the removal of old and the addition of new constraints. Therefore, such an update can either increase or decrease the size of the solution space, depending on the specific update. The constraint $c4$ is based on a message in a sequence diagram, and it requires the `LightSwitch` class to provide a `Method` with the name `dim`. If the constraint is updated so that it requires an instance of a more specific class, for example an instance of `AsyncMethod` instead of `Method`, $c4$ is updated to $c4'$, the solution space size is decreased (indicated by the thinly shaded area around $B5$ in Fig. 8b), and $B5$ becomes inconsistent because the class only provides a `Method` object. However, Fig. 8b also illustrates a possible update to $c4''$ (indicated by the dotted line) where the required object is allowed to be an instance of a more general class (e.g., `NamedElement`). This update would increase the solution space by the densely shaded area and make $B6$ consistent.

After performing the addition of $c5$ and $c6$, the removal of $c1$, and the update of $c4$ to $c4'$, the updated solution space consists of $B1$, $B2$, and $B4$ as shown in Fig. 8c.

4 Providing guidance and executing fixes automatically

When an inconsistency is detected, the minimum amount of guidance provided to the designer is a notification about the inconsistency's occurrence and its location (i.e., which model element violates which constraint). Based on data captured during constraint validation, consistency checkers can determine which model elements are actually causing the inconsistency [21]. Hence, the designer can be informed about the locations of error-causing elements.

Constraint-driven modeling may appear inferior to traditional transformation in that it does not generate or update model elements in the restricted model. However, there is cur-

المنارة للاستشارات
www.manaraa.com

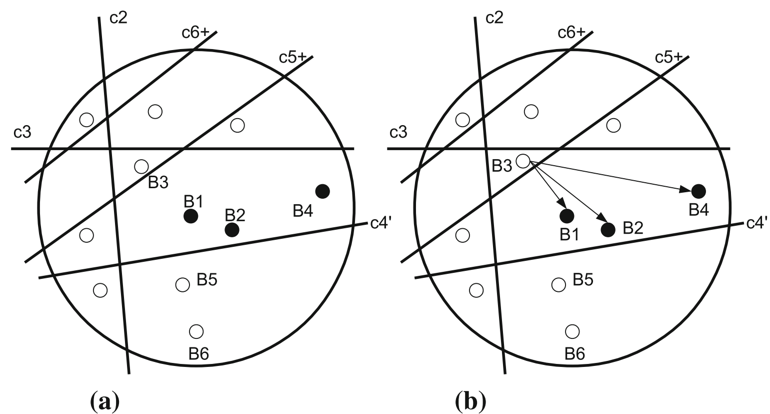


Fig. 9 Adding guidance to transform inconsistent models to consistent ones. **a** Without guidance, **b** with guidance for *B*3

rently considerable progress in suggesting repairs to individual inconsistencies in design models (e.g., [20,21,25–29]). Based on a specific constraint (or a set of constraints) and the inconsistent parts, it is thus possible to derive modifications—like specialized transformations—that lead to a consistent model. If such modifications can be derived, they are proposed to the designer as a list of options, or a suitable solution may be chosen automatically by applying heuristics. In Fig. 9, the general idea of fixing is illustrated for the inconsistent model version *B*3. The solution space for the modeling problem is depicted in Fig. 9a. In Fig. 9b, the derived fixes for the inconsistency (i.e., the possible transformations that may be executed to change the current model version to one within the allowed solution space) are depicted with solid arrows from *B*3 to *B*4, *B*5, and *B*6. If the restrictions are unambiguous, only a single option remains and it may be applied automatically (much like transformation). For example, the action ⟨`addmethod"dim"toclass"LightSwitch"`⟩ is an option for removing the inconsistency caused by the absence of the method `dim` in the `LightSwitch` class and the constraint *c*4. Thus, using constraints does not only expose inconsistencies, but it also enables user guidance to help understanding and solving them [30]. However, models may contain a large number of inconsistencies that makes manually fixing them one by one—even with guidance—practically impossible. In such cases, the use of fully automated approaches is inevitable. Thus, deciding which fixing strategy to use (i.e., guided manual fixing or automated fixing) strongly depends on the specific situation (i.e., the present inconsistencies). Let us now discuss how both strategies support our approach and for which typical scenarios they are most suitable.

4.1 Guided fixing

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

We discussed above that structural dependencies between constraints do not occur and that their validation is independent. However, *logical dependencies* between constraints in terms of required model characteristics and corresponding model elements may occur (e.g., *c*1 requires a class `LightSwitch` and *c*2–*c*3 require specific methods in this class). Without a `LightSwitch` class, the model cannot be consistent. However, the constraints *c*2–*c*3 are consistent if there is no such class at all or if the specific class provides the methods. This means that choosing an option that removes the inconsistency of *c*1 (i.e., the addition of a new and empty class named `LightSwitch`) leads to the constraints *c*2–*c*3 being inconsistent until they are addressed by adding the required methods to the new class. Creating additional inconsistencies can therefore be necessary to achieve overall model consistency. Recent research has shown that such problem can be solved, for example using search-based transformation in combination with constraints [31]. Other recent studies suggest that such dependencies can be leveraged to decrease the number of possible actions to fix inconsistencies dramatically [32].

However, even without considering logical dependencies between constraints, it has been shown that the average number of fixes per inconsistency for common UML well-formedness constraints in industrial models stays below 10 [21]. Thus, a newly introduced single inconsistency can be easily handled through guided fixing. Moreover, the time for calculating possible fixes typically remains under 0.1 seconds with common approaches such as [33] or [21]. Therefore, chaining the incremental constraint generation and management of CDM with existing semi-automatic fixing approaches produces a modeling environment in which designers are informed about both inconsistencies and possible options for resolving them live during modeling. Note that a variety of such approaches is available for use with

المنارة للاستشارات
www.manaraa.com

CDM. For example, there are search-based approaches that find possible fixes by exploring the problem's solution space (e.g., [34] or [33]). Moreover, there are also incremental techniques that compute possible fixes directly based on specific inconsistencies (e.g., [21]).

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

4.2 Automated fixing

Although guided fixing is suitable for removing small numbers of inconsistencies, there may be situations in which larger number of inconsistencies occur that are not manageable through fixing inconsistencies one by one. As we have briefly mentioned above, constraints may come from diverse sources. For example, they may be derived from a metamodel and may be used for validating the structural integrity of a specific metamodel instance (i.e., model) in a flexible modeling tool (e.g., [35–37]). In such a case, a generic constraint derived from the metamodel could, for example, check that all modeled classes provide exactly one name. Another constraint may check that no class has more than one parent class. Note that such generic constraints have to be enforced for every single class present in the restricted model. Indeed, if a new concept is then introduced in the metamodel (e.g., a new single-valued attribute for classes called "description"), this would result in the generation of a new generic constraints that required all classes to provide such a field. For every single class in the restricted model, an individual inconsistency is detected. Indeed, the number of inconsistencies that occurs due to such a change is often not manageable manually.

There has been significant progression in the field of automated model healing and fixing of inconsistencies. For instance, Anastasakis et al. [27] presented an approach for the transformation of UML models and OCL constraints to Alloy that allows for sophisticated reasoning over UML models. After using Alloy for analyzing the model and extending it automatically to establish consistency, the extended model can be transformed from Alloy back to UML using the approach presented by Shah et al. [28]. Kuhlmann and Gogolla [29] use a similar approach for transforming UML models and OCL constraints to relational logic for doing SAT-based reasoning with *Kodkod* [38] and transforming the resulting model back to UML.

A different approach for fixing inconsistencies was presented by Xiong et al. [39]. They developed a language called *Beanbag* that allows the definition of constraints and fixing behavior at the same time. When a constraint becomes inconsistent, its associated fixing behavior is used to derive a set of model changes that are executed to restore consistency. The Beanbag language is quite similar to standard OCL, but provides several additional language constructs that allow constraint authors to also specify how issues may be solved. Our approach can be used to generate such Beanbag programs instead of pure OCL constraints. Note that with our approach, the fixing-related parts may also be generated using specific source model data, potentially allowing for more precise or more efficient calculation of fixes.

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

4.3 User-centric approach

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

As discussed in Sect. 4.1, changes in the source model will often lead to a small number of new inconsistencies that can be fixed manually because of typically small numbers of available options. However, as discussed in Sect. 4.2, there may also be scenarios in which multiple inconsistencies—which are, however, based on a single kind of constraint—are introduced by a single source model modification. In that scenario, the use of fully automated approaches will often be the best solution. Note that using an automatic approach based on sophisticated reasoning is always an option for getting a valid model quickly.

5 Additional benefits of constraint-driven modeling

Above, we described the main benefits of CDM on which we also focused in three case studies we used for evaluation (see below). There are, however, additional benefits that are beyond the scope of this paper. Next, we introduce additional scenarios in context of rule-scheduling, model merging, and bidirectionality.

المنارة للاستشارات
www.manaraa.com

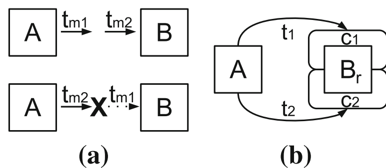


Fig. 10 From dependent (**a**) to independent (**b**) transformation rules

5.1 Rule-scheduling and race conditions

Now, let us consider an example where two transformation rules t_{m1} and t_{m2} are working with the same generated model and the order of rule execution is important. For example, the sequence diagram in Fig. 1a contains an instance of the class `LightSwitch`. Therefore, let us assume that transformation rule t_{m1} generates a corresponding class if no such class exists in the diagram in Fig. 1b. As we have discussed in Sect. 2, the sequence diagram requires the class `LightSwitch` to provide a method `activate`. Let transformation rule t_{m2} generates this method in `LightSwitch`.[5] When the transformations are performed, it is crucial that t_{m1} is executed before t_{m2} to ensure that the class `LightSwitch` exists before the method `activate` is added. This issue is illustrated in Fig. 10a where the bottom transformation encounters an error after the execution of t_{m2}. If the rule t_{m1} is still executed, the resulting model B will contain an empty `LightSwitch` class because only t_{m1} was executed successfully. If the execution of rules is stopped after the error, no model is generated at all. Defining the order of rule execution manually is tedious and a constant source of error. Moreover, support for defining an execution order is not a standard feature of all transformation languages or systems [3].

The constraining approach, shown in Fig. 10b, is free of scheduling issues because constraints cannot structurally depend on other constraints, and the order of transformation is not relevant for the transformation results, as discussed in Sect. 3.3. Hence, the rules t_1 and t_2 we have previously defined can be applied in any order and produce the distinct constraints $c1$ and $c2$ in Fig. 10b. If a model does not provide the required information for constraint validation (e.g., the class that should be checked is not present), the validation fails and an inconsistency is detected.

5.2 Bidirectionality and model merging

When models should be synchronized automatically, transformations are often used to propagate changes from one model to the other and perform the corresponding changes. Let us assume that we have established transformation rules that keeps message names and method names synchronized and that a link between messages and corresponding methods exists. In Fig. 5a, the name of the highlighted message has been changed from `turnOff` (see Fig. 1a) to `deactivate`. Concurrently, the corresponding method in the class diagram was changed from `turnOff()` (see Fig. 1b) to `switchOff()`, as highlighted in Fig. 5b. Since both synchronized model elements were changed (indicated by the bold arrows), there is no way to determine in which direction the required synchronization should be performed. Performing a synchronization in this situation will always lead to the loss of the changes in the generated model (i.e., either B'' overrides changes in B' or A'' overrides changes in A' that cannot be used for a transformation in the opposite direction afterward). A possible solution would be the concurrent execution of the transformations followed by a merge of the updated models (A' and B') and the resulting generated models (A'' and B''), as illustrated in Fig. 11a that generated A''' and B'''. However, this requires a complex merging strategy and is likely to produce models that still require manual adaptation.

The solution of the constraint transformation approach is shown in Fig. 11b. We can see that our approach still has to decide which change to process first. However, because only constraint models are updated, the restricted models A'_r and B'_r are not changed and can therefore still be

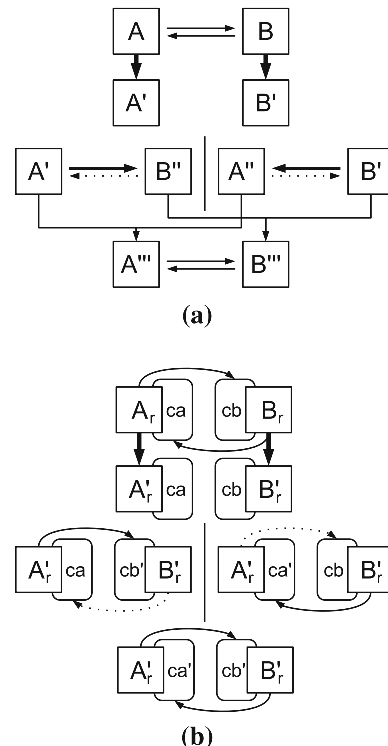


Fig. 11 From bidirectionality (**a**) to unidirectional (**b**) constraint transformation

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

للاستشارات المنارة
www.manaraa.com

processed to perform constraint updates in the opposite direction, leading both constraint models *ca* and *cb* being updated. With our approach, no immediate merging (either automated or manual) is required when restricted models are edited and following source model changes lead to constraint updates.

After the constraint model updating took place in the example, there are two new constraints: (i) message number 2 in Fig. 5a should be named `switchOff` (from Fig. 5b), and (ii) the name of the method `switchOff` in Fig. 5b should be changed to `deactivate` (from Fig. 5a). The designer can then decide which of the elements should be renamed.

6 Case studies

To assess the feasibility and the applicability of CDM, we conducted three case studies with two goals: (i) demonstrate the applicability of CDM to various domains, and (ii) assess the efficiency and performance of our constraint management approach (i.e., the incremental transformation of constraints). For the former goal, the three case studies are large-scale industrial-grade models and span across the domains of object-oriented software modeling, metamodeling, and software product lines. CDM was used to generate various kinds of domain-specific constraints. For the latter, we performed various source model changes to trigger changes in the set of enforced constraints (e.g., constraint addition). In Fig. 12, the different sources of change (Δ) in our approach are depicted. Note again that the application of CDM consists of two core phases: (i) constraint management, and (ii) constraint validation. The first phase is triggered by changes of the source model (ΔA) and includes an update of the applied constraints (ΔC). The second phase—validation—is triggered by changes of either the constraints (ΔC) or the restricted model (ΔB).

In principle, we need to demonstrate performance for both phases—including the validation triggered by changes of the restricted model (ΔB). However, the latter can be omitted for two reasons. First, this work uses an existing consistency checker that was already thoroughly evaluated on 34 large-scale industrial models of up to 162,237 model elements and complex constraints in [17,40]. It was shown that most changes to a restricted model are processed in less than one millisecond and that the validation is faster than typical batch-validation performed by conventional consistency checkers. Second, this paper does not prescribe any consistency checker in particular. In principle, any technology may be used. The one used in this paper is merely a proof that

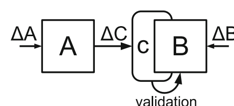


Fig. 12 Possible changes in CDM

current state of the art is sufficiently progressed to support the second phase. Thus, the following focused on assessing the first phase (i.e., processing of ΔA and ΔC) in our performance tests and captured the time needed for handling the source model change and generate, remove, or update constraints. However, the time for constraint validation triggered by the resulting changes of constraints (ΔC) was also measured, and in our analysis, we provide values for both constraint management only and constraint management with constraint validation. The observed times for the validation of constraints were compared with the previously observed validation times for manually written constraints to assess how using generated constraints affects the validation performance of existing consistency checking technologies.

6.1 Prototype implementation

To evaluate the three case studies, we implemented a prototype tool. The core component of the tool is a constraint transformation engine that supports arbitrary EMF-based source models and generates OCL constraints. For efficient constraint validation, we employ the *Model/Analyzer* [16] consistency checking framework which we slightly adapted to support EMF-based models in general instead of only UML models. Note that the constraint validation may be done by any consistency checker that supports EMF models and OCL constraints. However, we chose the Model/Analyzer because it was evaluated with design models that were used in two of the case studies presented below (i.e., Case Study I and Case Study II). Thus, observed validation times may be compared to previously observed data. Change trackers are used to observe models and inform the transformation engine or the consistency checker, respectively, about changes. The transformation engine is capable of handling arbitrary numbers of source and restricted models concurrently.

6.2 Case Study I: model to model constraint

For our first case study, we apply CDM to typical object-oriented software models and generate constraints from different diagrams, as we have already illustrated in Sects. 2 and 3. In particular, we use individual parts (i.e., diagrams) of large-scale UML models from which we generate constraints that restricts other parts of the same model.

6.2.1 Case study models

For this case study, we employed 22 of the industrial models we previously used in [17,40]. Note that those 22 models were selected because they include different types of diagrams from which constraints that restrict other diagrams can be generated (i.e., we did not use design models that include diagrams of only one type).

المنارة للاستشارات
www.manaraa.com

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

6.2.2 Transformation rules

We used the transformation rule $t2$ shown in Listing 2 to generate constraints from `Message` instances in sequence diagrams, as discussed in Sect. 3.

6.2.3 Performance evaluation

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

Test I Replacing ambiguous transformations—as discussed in Sect. 3
Test II Replacing merges—different sources restrict a single model
Test III Restricting multiple models—one source restricts different models

Test I simulated simple unidirectional transformations in the scenario we described in Sect. 3. Note that UML models usually consist of one single model that includes all data and that different diagrams only represent different views on that data. Therefore, we use the same model as source and restricted model in Test I (this being another benefit of our approach). The test was executed for all available models that included elements matching the transformation rule context.

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

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

Performed source model changes. For all tests, randomly selected single model elements were removed from a source model and then added back to the source model, which forced an incremental constraint model update. That is, the update of exactly one constraint for Test I and Test II and the update of at least one constraint for Test III. For Test II, the source model on which a change was performed was also chosen randomly for every performed change. Note that we only performed changes that updated constraints which were actually validated on at least one restricted model element. This ensures that all changes impact source models used during transformation. Note also that complex model changes, such as the removal of a whole package, can be expressed as a sequence of atomic changes. Therefore, the results we obtain for atomic changes are sufficient to predict the effects of complex model changes.

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

Results We now discuss the observations made for the individual tests.

Test I In Fig. 13a, the median times for adding a source model element are depicted. For the transformation time (i.e., the time required for constraint generation), the observed processing times are steady at about 1 ms and slightly increase for large models of over 27,000 model elements to about 10 ms. For the total processing time including both transformation and constraint validation, we observed a correlation with the model size. This is based on the fact

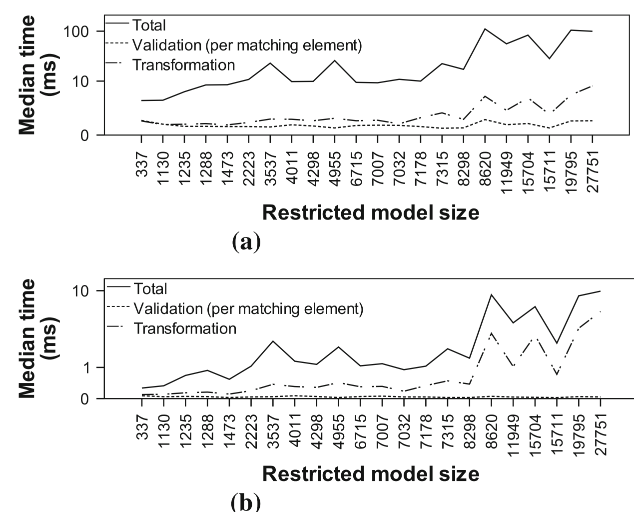


Fig. 13 Results for Test I. **a** Addition processing times, **b** removal processing times

المنارة للاستشارات
www.manaraa.com

that we generated generic constraints that were validated for all instances of a specific type. Therefore, the total time required for validation increases with the number of elements matching the constraint context. However, the dashed line in Fig. 13a shows that the validation time per constraint context matching element was steady at under 1 ms. These observations are consistent with those made in [17] and indicate that the constraints generated with CDM allow for the same quick validation as manually written constraints. Note that the total processing times even for large models reach a maximum of about 100 ms.

The median times for removing a source model element are depicted in Fig. 13b. Note that the median total processing time has a maximum of about 10 ms even for the largest models. The time required for constraint validation is negligible because there is no validation required when removing constraints. We observed a correlation between the number of constraints generated from the source model and the time required for finding and removing the constraints based on the removed source model element in the test. The spikes in the graph for models between 8,000 and 20,000 model elements support this assumption and indicate that the model characteristics have a stronger effect than the model size.

Test II In Fig. 14a, the processing times for the addition of an element to one out of five concurrent source models are shown. Note that there is no correlation between the total source model size and the median processing times.

In Fig. 14b, the processing times for ten concurrent source models are depicted. Again, there is no correlation between the total source model size and the processing time. Additionally, note that the processing times for both five and ten source models are typically within 10 and 100 ms—similar to the results of Test I in Fig. 13a—indicating that the additional infrastructure required for handling concurrent source models does not lead to reduced efficiency.

In Fig. 14c, the median required processing time for the addition of a source model element as a function of the number of restricted model elements matching the affected constraint's context (i.e., the number of constraint validations triggered by the change) is depicted. As for Test I, both the validation time per constraint matching element and the transformation time are steady at about 1 ms and reach a maximum for large models of 5 ms. The total processing time of course increases with the number of required validations. For the removal of a source model element, both the processing times for validations per matching element and for the transformation remain at under 1 ms and the maximum total processing time stays under 100 ms (figures are omitted).

Test III The observed processing times for the addition of an element to the source model are depicted in Fig. 15a. The results for tests with both five and ten concurrent restricted models are combined, and the times are drawn as a function of

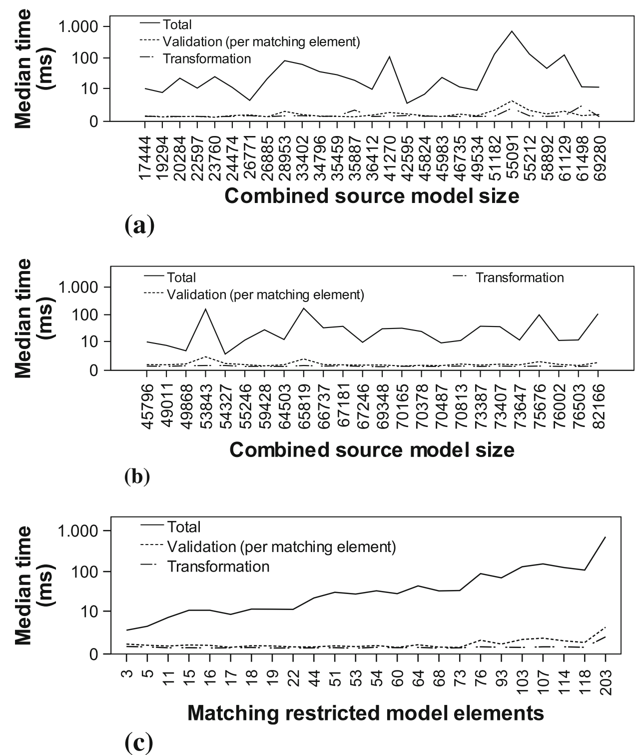


Fig. 14 Results for Test II. **a** Addition processing times (5 source models), **b** addition processing times (10 source models), **c** addition processing times for different effects on restricted model

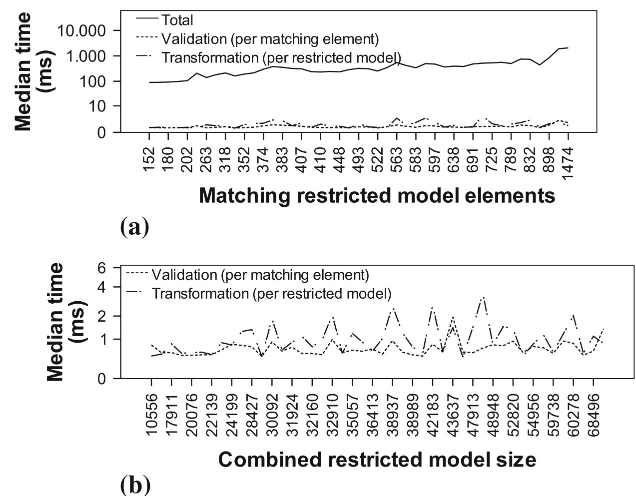


Fig. 15 Results for Test III. **a** Addition processing times for different change effects, **b** addition processing times for different restricted model sizes

the number of constraint validations that become necessary after the change. Note that the times for both transformation per restricted model and constraint validation per matching element are nearly static at about 1 ms. As in Test II, the total processing time is determined primarily by the number of required constraint validations.

المنارة للاستشارات
www.manaraa.com

In Fig. 15b, the processing times for constraint validation and transformation are depicted as a function of the combined restricted model sizes.

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

6.3 Case Study II: metamodel to model constraints

Model transformations are commonly used in metamodeling when *co-evolution* of metamodels and models is required [41] (i.e., when models must be adapted after their metamodel was changed). As a second case study, we applied CDM to metamodeling and use it for addressing the important issue of co-evolving metamodels and model constraints [42]. While there is much research done about co-evolution of metamodels and models, for instance [41,43], less work was done to co-evolve constraints. This case study demonstrates how CDM can be used to address the issue of metamodel-knowledge-based constraints that may become incorrect after metamodel changes.

In this scenario, our approach complements existing metamodel and model co-evolution approaches. Note that required model updates often cannot be performed automatically because of various uncertainties like arbitrary naming of elements or cardinalities. For example, adding a mandatory attribute `supertype` to all `Element` objects adds the constraint ⟨`contextElementinv : self.supertype <> null`⟩. Even though a common supertype object can be inserted automatically to all `Element` instances, the correct solution still depends on the specific element and the desired object hierarchy. Thus, trying to automate such model updates will inevitably make the intervention of the designer necessary. By using CDM, constraints that are based on metamodel knowledge are also updated automatically when the metamodel evolves. For instance, if the cardinality of the attribute `supertype` is changed from `[1..1]` to `[1..*]` to allow multiple inheritance, the constraint is updated to ⟨`contextElementinv : self.supertype -> size()`⟩ = 1. Thus, CDM keeps metamodel-based constraints correct which provides valuable information for performing co-evolution of design models.

6.3.1 Case study models

For this case study, we used the UML metamodel and 28 industrial UML models from [17]. On the first glance, the UML meta model may not appear to be an ideal choice because it is rarely used for co-evolution. Nonetheless, for measuring the performance of our approach, the UML is as suited as any meta model. The UML metamodel is clearly a sophisticated and complex metamodel for defining various types of diagrams related to object-oriented software engineering artifacts. And it is also very large (i.e., the UML metamodel was given as Ecore model which contained 6583 model elements). If our approach scales for UML model/meta model evolution, then this should be convincing.

6.3.2 Transformation rule

The UML metamodel is defined with the *Meta-Object Facility (MOF)* [44]. The elements available in UML are modeled as instances of the *Ecore* type `EClass`. Thus, it is possible to define a single transformation that uses `EClass` instances to generate constraints for the corresponding UML elements. The rule is shown in Listing 3 (lines 1–9). For a given `EClass`, which must not be abstract or an interface, it generates a constraint that is validated for all instances of the defined UML metamodel element (e.g., an `EClass` instance with the name `Message` is transformed to a constraint that checks all instances of `Message` in a UML model). Generating the invariants for the model elements is done in two separate helper functions called `generateAttributeInvariants` and `generate ReferenceInvariants` for the `EAttribute` and `EReference` instances, respectively, belonging to the `EClass` either directly or through inheritance.

Function `generateAttributeInvariants` is shown in Listing 3 (lines 11–17). It takes as input an ordered set of attributes and recursively generates the invariant string by calling the helper `generateAttributeInvariant` with the first available attribute and itself with the remaining set of attributes.

For individual attributes, the helper `generate AttributeInvariant` only creates a statement for checking cardinality by calling the helper `cardinality Statement`, as shown in Listing 3 (line 19).

The helper `cardinalityStatement` generates invariants based on an associations lower and upper bounds, as shown in Listing 3 (lines 21–41). For unbounded associations (i.e., [0..1] for single-valued and [0..*] for multi-valued associations), the helper simply returns "true", and for bounded associations, the helper returns statements to check that an element provides the correct number of results for the given association.

Function `generateReferenceInvariants`, as shown in Listing 3 (lines 43–49), is a recursive helper similar to that for attributes, and thus, we omit a detailed discussion for space reasons.

To generate the invariants for an individual reference, the helper `generateReferenceInvariant`, shown in Listing 3 (lines 51–52), not only generates a cardinality

المنارة للاستشارات
www.manaraa.com

```
1  rule EC
2     from
3        s : MetaModel!EClass (
4           not s.isAbstract and not s.isInterface)
5     to
6        t : ConstraintModel!Constraint (
7           context <- s.name;
8           inv <- generateAttributeInvariants(s.eAllAttributes->
               asOrderedSet()) + " and " +
               generateReferenceInvariants(e.allReferences->
               asOrderedSet());
9     )
10
11 helper def generateAttributeInvariants(s:OrderedSet(EAttribute)):
      String =
12    if s->size()=0
13    then
14       "true"
15    else
16       let x=s->first() in generateAttributeInvariant(x) + " and "
            + generateAttributeInvariants(s-x)
17    endif
18
19 helper def generateAttributeInvariant(s:EAttribute):String =
      cardinalityStatement(s.upperBound,s.lowerBound,s.name)
20
21 helper def cardinalityStatement(l:Integer,u:Integer,n:String):
      String =
22    if (l=0 and u=-1) or (l=0 and u=1)
23    then
24       "true"
25    else
26       if l=1 and u=1
27       then
28          "self."+n+"<>null"
29       else
30          if l=0
31          then
32             "self."+n+"->size()<="+u
33          else
34             if u=-1
35                "self."+n+"->size()>="+l
36             else
37                "self."+n+"->size()>="+l+"and self."+n+"->size
                     ()<="+u
38             endif
39          endif
40       endif
41    endif
42
43 helper def generateReferenceInvariants(s:OrderedSet(EReference)):
      String =
44    if s->size()=0
45    then
46       "true"
47    else
48       let x=s->first() in generateReferenceInvariant(x) + " and "
            + generateReferenceInvariants(s-x)
49    endif
50
51 helper def generateReferenceInvariant(s:EReference):String =
52    cardinalityStatement(s.upperBound,s.lowerBound,s.name) + " and
         " + oppositeStatement(s,s.opposite)
```

Listing 3 Transformation rule and helper functions to generate constraints for UML metamodel types

statement, as it is done for attributes, but it also generates semantic invariants that check the correct implementation of opposite references. The latter is done through the helper `oppositeStatement`, which is shown in Listing 4.

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

```
1  helper def oppositeStatement(x:EReference,y:EReference):String =
2     if y=null
3     then
4        "true"
5     else
6        if x.upperBound=1
7        then
8           if y.upperBound=1
9           then
10             "self."+x.name+"."+y.name+"=self"
11          else
12             "self."+x.name+"."+y.name"+"->includes(self)"
13          endif
14       else
15          if y.upperBound=1
16          then
17             "self."+x.name+"->forAll(z|z."+y.name+"=self)"
18          else
19             "self."+x.name+"->forAll(z|z."+y.name+"->includes(
                   self))"
20          endif
21       endif
22    endif
```

Listing 4 Helper function `oppositeStatement`

6.3.3 Performance evaluation

For our performance evaluation, we use a basic setup with a single source and a single restricted model: the UML metamodel is used as source model that restricts a single UML model. Each of the industrial UML models from Case Study I was used as restricted model.

Performed source model changes. For the performance test, we removed and added existing attributes and references from/to a UML metamodel element (e.g., `Class`, `Message`, `InstanceSpecification`). This was done for 100 randomly selected elements and attributes/references in the UML metamodel. Note that for our statistics, we ensured that all changed metamodel elements were actually instantiated in the restricted model.

Results The observed results for this case study are depicted in Fig. 16. Not surprisingly, the time required for constraint management only (i.e., transformation of constraints without constraint validation, or the removal of existing constraints) is steady for all restricted models, regardless of their size.

For constraint validation, on the other hand, we observed that the required time increases with the restricted model size. Note that this is based on the fact that the generated constraints have to be validated not only one time, but once for every restricted model element that matches the constraint's context (e.g., all `Class` instances), and larger restricted models typically have higher numbers of those matching elements.

However, the median total validation time for a restricted model of just under 10,000 model elements was still below 500 ms for a source model change that required the constraint to be validated for an average of 200 different elements. During the transformation, data from in average 107 different

Springer
www.manaraa.com

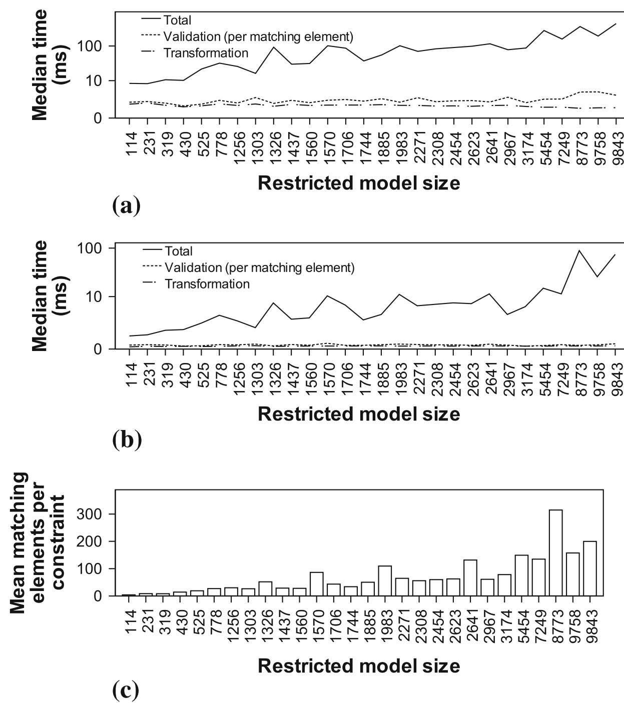


Fig. 16 Results for Case Study II. **a** Metamodel element addition processing times, **b** metamodel element removal processing times, **c** average number of matching restricted model elements per constraint

metamodel elements were processed on average (i.e., implying that the transformations were far from trivial).

The spikes in the solid lines showing the total processing times in Fig. 16a, b are consistent with the average numbers of constraint matching elements in the respective restricted models shown in Fig. 16c. Note that the validation time per constrained target model element (e.g., the time for the validation of a single `Class` instance), shown by the dashed line in Fig. 16a, b, remains nearly unchanged for different restricted model sizes.

6.4 Case Study III: software product line to product constraints

Software Product Lines (SPLs) [45] have become popular as they foster systematic software reuse and support simple configuration of products. In this case study, we apply CDM to SPLs by: i) generating constraints that ensure the correctness of configured products, and ii) updating those constraints during SPL evolution.

Hierarchical *feature models* are commonly used in SPLs to express the different *features* (i.e., *functionalities and capabilities*) products may have and how these features are related [46]. Different kinds of associations between features have been discussed in literature that define how features are related and how they may be combined in a specific product (e.g., [45–50]). For example, the selection of a feature $F1$ in a product requires a feature $F2$ to be also selected (*mandatory* relation), or that the selection of $F1$ requires at least / exactly one of the features $F3$ and $F4$ to be selected in a product (*or* / *xor* relation). Moreover, there are *cross-tree constraints* that relate features and may break the hierarchical structure. Examples of such cross-tree constraints are *requires* and *excludes*, which require their target features to be selected or not selected in a product, respectively. A *product configuration* consists of a set of features that are combined to build a specific product (i.e., $\{F1, F2, F4\}$), and the configuration is valid if none of the relationships between features defined in the feature model are violated. We used our prototype to generate constraints that restricted product configurations from a feature model.

Updates of those constraints become necessary when the feature model changes as a consequence of technological innovation or new business strategies. Existing product configurations must then be checked for consistency with constraints based on the updated features and relations. Defining and maintaining the required constraints manually for a large number of features is practically impossible and thus not an option.

Changing a configuration that has become inconsistent with the updated feature model is a task that requires domain knowledge and also intuition for valuable business decisions. Performing such an update automatically to generate a new and consistent version of the product would lead to a correct but also potentially impractical and thus non-saleable product. Hence, automating this update task is not a valid solution to the problem. However, by using the guidance provided by the consistency checker, it is easy for product configurators to detect the erroneous parts and find a valid and practical fixing strategy that results in the intended product.

6.4.1 Case study system

For assessing the performance of CDM in the domain of SPLs, we use a large-scale benchmark product line: the *Barbados Crash Management System Product Line (bCMSPL)* [51], which is publicly available in ReMoDD [52]. The case study SPL is modeled using a standard feature model that consists of 66 features and denotes a total of 440,640 valid products.

6.4.2 Transformation rules

We used transformation rules to generate typical product constraints from a feature model. We focused on commonly accepted product constraints as described, for example, in [50]. The required constraints were generated from seven transformation rules.

المنارة للاستشارات
www.manaraa.com

```
1   rule tMandatory
2       from
3           s : FeatureModel!Mandatory
4       to
5           t : ConstraintModel!Constraint (
6               context <- Product,
7               inv <- "self.features->includes("+s.sourceFeature+")
                    implies self.features->includesAll("+s.
                    targetFeatures+")")
8
9   rule tOr
10      from
11          s : FeatureModel!Or
12      to
13          t : ConstraintModel!Constraint (
14              context <- Product,
15              inv <- "self.features->includes("+s.sourceFeature+")
                    implies self.features->select(x|"+s.
                    targetFeatures+"->includes(x))->size()>0")
16
17  rule tXor
18      from
19          s : FeatureModel!Xor
20      to
21          t : ConstraintModel!Constraint (
22              context <- Product,
23              inv <- "self.features->includes("+s.sourceFeature+")
                    implies self.features->select(x|"+s.
                    targetFeatures+"->includes(x))->size()=1")
24
25  rule tRequires
26      from
27          s : FeatureModel!Requires
28      to
29          t : ConstraintModel!Constraint (
30              context <- Product,
31              inv <- "self.features->includes("+s.sourceFeature+")
                    implies self.features->includesAll("+s.
                    targetFeatures+")")
32
33  rule tExcludes
34      from
35          s : FeatureModel!Excludes
36      to
37          t : ConstraintModel!Constraint (
38              context <- Product,
39              inv <- "self.features->includes("+s.sourceFeature+")
                    implies self.features->select(x|"+s.
                    targetFeatures+"->includes(x))->size()=0")
40
41  rule tParent
42      from
43          s : FeatureModel!Feature (
44              s.parent<>null)
45      to
46          t : ConstraintModel!Constraint (
47              context <- Product,
48              inv <- "self.features->includes("+s+") implies self.
                    features->includes("+s.parent+")")
49
50  rule tRoot
51      from
52          s : FeatureModel!FeatureModel
53      to
54          t : ConstraintModel!Constraint (
55              context <- Product,
56              inv <- "self.features->includes("+s.root+")")
```

Listing 5 Transformation rules for product lines

The first rule, shown in Listing 5 (lines 1–7), is executed for all mandatory associations. Such mandatory associations mean that every target feature must be selected in a product if the source feature is selected. The second rule we defined, shown in Listing 5 (lines 9–15), is used for or-associations that require at least one of the target features to be selected in a product if the source is selected. The third rule for xor-associations, shown in Listing 5 (lines 17–23), is similar to the second rule. Note that for xor-associations, exactly one instead of at least one target feature must be selected if the source is selected. The fourth and the fifth rule are used to transform, requires, and excludes cross-tree constraints, as shown in Listing 5 (lines 25–31) and Listing 5 (lines 33–39), respectively. These two rules are very similar to $tMandatory$ (Listing 5, lines 1–7) and $tXor$ (Listing 5, lines 17–23) relations. For $tRequires$, only the source element type was changed to `Requires`. For $tExcludes$, the source element type was changed to `Excludes` and the size of the result must be equal to 0.

The sixth rule, shown in Listing 5 (lines 41–48), generates constraints for the generic product invariant that a feature (that is not the root feature) may only be selected in a product if its parent feature is also selected. The rule is guarded so that it is only executed for features that actually have a parent (i.e., all features but the root feature). The seventh rule, shown in Listing 5 (lines 50–56), generates a constraint that requires the root feature of the feature model to be selected in a product.

6.4.3 Performance evaluation

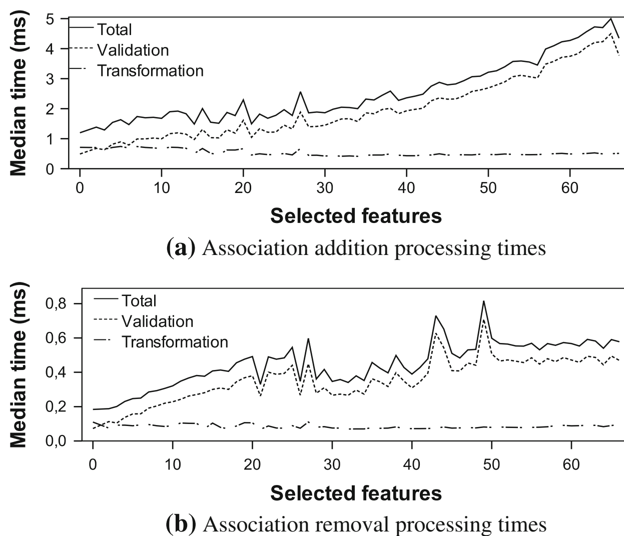
For assessing the performance of our CDM implementation in the domain of product lines, we used a setup with a feature model as source model and a product as target model.

Performed source model changes. For this case study, we measured the time required for processing the addition/removal of an association or cross-tree constraint to the feature model. For each of the existing 42 associations and cross-tree constraints in the bCMSPL feature model, we performed the removal and the addition of the element. For our tests, we used products that were composed of randomly selected features. The used products included the empty product (i.e., zero features selected) and the fully configured product (i.e., all 66 features selected).

Results The results for our tests are shown in Fig. 17. For the addition of a new association or cross-tree constraint, Fig. 17a shows that the total time for processing in all tests stayed below 5 ms. Note that the time required for the transformation (i.e., the creation of the constraint) remains nearly unchanged at under 1 ms. The time required for constraint validation increases with the number of selected features because the constraints we generate contain expressions that require an iteration over all selected features.

For the removal of associations or cross-tree constraints from the feature model, Fig. 17b shows that the total processing time stays under 1 ms for all performed changes. Again, note the constant time of about 0.1 ms required for the transformation (i.e., removal of the existing constraint that

المنارة للاستشارات
www.manaraa.com



(a) Association addition processing times

(b) Association removal processing times

Fig. 17 Results for Case Study III. **a** Association addition processing times, **b** association removal processing times

becomes obsolete after the change). Because there is no actual validation required for the removal of a constraint, the slope of the line indicating the validation is less than for the addition. However, the slope is still positive because the consistency checker must discard data that were captured during previous validation of the removed constraint and the amount of that data correlates with the number of selected features.

7 Discussion

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

7.1 Key challenges addressed

Let us briefly revisit the three key challenges we identified in Sect. 1 and discuss how they are addressed in CDM.

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

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

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

7.2 Scalability

We demonstrated the feasibility of CDM by implementing a prototype tool. The applicability was demonstrated by applying the approach to three different domains through our case studies. Each case study evaluated a range of small to large systems.

The observed transformation times remained nearly constant for all three case studies even when model sizes increased significantly, which indicates that the core aspect of our approach can be performed efficiently. The observed overall processing times (including constraint validation) show that even for large models, our approach works efficiently. The results demonstrate in a proof-of-concept manner that there exists at least one consistency technology that supports CDM in a scalable manner.[6]

Case Study I and Case Study II were conducted with the same models that were previously used for evaluating the employed consistency checker. The observed constraint validation times indicate that times required for validating generated constraints are comparable to those required for validating manually written constraints. Thus, generated constraints do not necessarily impose additional complexity or require higher validation effort.

7.3 Correctness aspects

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

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

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

[6] Note that the validation is not meant to demonstrate superior performance of the employed consistency checker but only to demonstrate feasibility of CDM.

المنارة للاستشارات
www.manaraa.com

to detect and report or fix errors in the source model or the transformation rules [21].

7.4 Threats to validity

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

In Sect. 3, we illustrated the application of CDM and assumed that transformation rules are executed incrementally. Even though there is significant progress in terms of incremental execution of transformations for common languages such as ATL or QVT, there are transformation languages for which such support is not available. However, we have discussed in Sect. 3.3 that individual constraints are structurally independent. This simplifies the implementation of incremental rule execution dramatically as it allows tools to execute a transformation rule with a given source model element in a sandbox-like environment (i.e., execute the transformation rule with a temporary and empty target model) and then add the resulting constraint to the set of applied constraints. Note that the order of rule execution does not matter. Moreover, if transformation rules are executed individually, then this also simplifies the backtracking later (i.e., if it is believed that a constraint is incorrect).

8 Related work

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

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

The major difference between CDM and approaches such as, for example, the ones presented by Büttner et al. [22] or Cabot et al. [23] is that the goals of these approaches are different from those of CDM. While CDM helps avoiding uncertainties and premature decisions during transformation rule writing in order to avoid the generation of unintended target models, other approaches typically generate constraints in order to do verification or validation of transformation rules through sophisticated reasoning over those transformation rules and target models (e.g., derive an optimal order of execution for a set of transformation rules). They derive target model conditions from the transformation rules directly through a translation. Thus, they are typically not capable of dealing with domain-specific semantics except for what is already expressed in the transformation rules. For the transformation in Listing 1, for example, those approaches may generate a constraint that requires a `Method` with a specific name and a specific owner to be present in the target model after the transformation was executed with a given `Message`. This constraint ensures that the transformation rule behaves as expected. The constraint may then be used in combination with constraints derived from other transformation rules, for example to find logical contradictions within the transformation rule, to find contradictions between rules, or to determine the required order of execution. Note that when those constraints are used as input for reasoning engines or fixing approaches, the constraints are already based on an assumption that was made during rule authoring to overcome an uncertainty (e.g., the desired location of the method) and that was incorporated in the transformation rule. With CDM, on the other hand, a constraint that a method must be *provided* by the class would be generated (i.e., a different transformation rule would be used), expressing the actually desired condition without making assumptions. Thus, existing approaches that generate constraints from rules are designed for different problems (e.g., ensuring syntactical correctness of target models or checking validity of transformation rules). However, those approaches are of course a valid choice for verifying and validating unambiguous transformations and may be used for verifying the transformations used for CDM (e.g., to find out whether the applied transformations lead to contradictory constraints).

Another approach related to CDM is proposed by Vallecillo et al. [53]. They define *Tracts*, contracts between source and target models which include constraints. Using those tracts, model transformation rules can be derived or existing transformation rules can be tested. While this approach is similar to CDM in that it focuses on the intentions of the designer and allows for domain-specific knowledge, it does not address the issue of designers being forced to make premature decisions when writing transformation rules. For example, it would identify the transformation rule presented in Listing 1 as valid with respect to the target model constraints—the solution it produces will always be valid as the required method is present in the target model. However, the transformation rule still relies on assumptions and it may produce unintended target models.

Another benefit of our approach is that it is generic. It can be used with arbitrary models, transformation languages, constraint languages, and consistency checkers. Existing approaches interpret rules and generate constraints for a given transformation language and are also strongly tailored

www.manaraa.com

to a constraint language, a consistency checker, or a reasoning engine (e.g., [22,23]).

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

Model transformation and OCL constraints. Büttner et al. [54] discussed various endogenous transformations of OCL constraints that are necessary when either the constraint or the model it is based on change. We apply transformations to automatically generate such constraints from a source model and, instead of transforming the existing constraints, use incremental re-transformation to reflect model changes. Simplifications of OCL constraints through transformations were also discussed by Giese and Larsson [55]. They defined transformation rules that could be used with our approach to simplify the constraints that have been automatically created and possibly make them faster to check and also easier to read. In [56], Bajwa and Lee propose an approach to automatically generate OCL constraints by transforming business rules specified in Semantics of Business Vocabulary and Business Rules (SBVR) [57]. Our approach of course also supports business models as constraint source. However, we have shown that the generation of constraints is a valid option to reduce ambiguities and provide guidance in different domains.

Bidirectional transformation. Giese and Wagner [58] as well as Xiong et al. [59] performed extensive research on bidirectional transformations and synchronization in general. Interestingly, Xiong et al. define an "undo" operation after model updates (i.e., reverting performed changes) to be an unwanted option for fixing introduced inconsistencies because it would ignore the latest decisions made by the designer [39]. Indeed, for automated fixing without human intervention, making such an assumption is necessary as otherwise a simple "undo" of the latest change would typically be the best solution that removes all inconsistencies without any side effects (i.e., introducing new inconsistencies through fixing existing ones). However, a designer's decisions can be wrong, and therefore, we believe that presenting the "undo" option in addition to other possible fixes is necessary to address this possibility. Regarding language support for bidirectional transformations, Sasano et al. [60] developed a system to perform bidirectional transformations with ATL, and Stevens [8] focused on bidirectionality for QVT. In general, we tackle the complexity of bidirectional transformations by using unidirectional transformations that generate constraints—without tailoring our approach to a specific transformation language or specialized transformation engines, and without deriving constraints from target model generating transformation rules. Updating one of the involved models, then changes its own consistency status and potentially lead to updates of constraints on the other model. The involved models are synchronized if they are both free of inconsistencies. As we have shown, the risk of overriding changes because of the change processing order is eliminated as we only update constraints automatically and the designer ultimately decides how the models should be adapted. Cicchetti et al. [61] developed the bidirectional transformation language *JTL* that supports non-bijective transformations and change propagation. JTL generates constraints by translating user-defined transformation rule and it uses *answer set programming (ASP)* to find models that match those constraints. The translation of transformation rules to ASP constraints can be interpreted as a transformation of those rules. Thus, JTL could be seen as an application of CDM chained with automatic model finding where the user-defined transformation rules are used as source model to generate constraints through the execution of other transformation rules (i.e., rules that capture the translation semantics).

Incrementality and execution speed of transformations. Jouault and Tisi [62] proposed an approach to make ATL transformations incremental. They achieve incrementality by using scopes built during OCL expression execution to determine which rules have to be re-executed after source model changes. We make use of automatically created scopes in the same way to determine which constraints have to be re-created in our prototype and also for finding constraints that have to be re-validated by the consistency checker [16]. In [63], Tisi et al. propose the lazy execution of transformations, which eliminates the need for an initial transformation of the entire source model to speed up the process for large source models, which is also the performance bottleneck of our prototype.

Product line constraints. The effect of feature model changes on the set of possible product configurations has been investigated by Thüm et al. [64]. While they used SAT solvers to find out how the set of all possible configurations is affected, we focused on existing products and used our approach to check whether they are still valid after feature model changes and to provide guidance for fixing possible inconsistencies.

We derived constraints based on standard rules as described in [50]. However, our approach of course also supports the generation of constraints for product line approaches that use additional feature model concepts such as feature group cardinalities, as described by Czarnecki et al. in [65].

المنارة للاستشارات
www.manaraa.com

Design space effects of constraints. Saxena and Karsai [66] published a MDE-based approach for design space exploration in which constraints are used to describe invariants of valid models. Our approach is ideal to generate constraints for design space exploration algorithms. The source of these constraints may be the metamodel to which the generated model must conform or also an already existing partial model that is either provided by the designer as input for the exploration algorithm or generated by the algorithm itself. Our approach also reduces the solution space and provides guidance to transform an invalid solution to one that is within the remaining solution space. Horváth and Varró [67] presented an approach for design space exploration using a CSP-solver and *dynamic constraints* that may change over time. Combined with CDM to generate and manage these constraints automatically, their approach is perfectly suitable for finding out whether there actually are restricted models for which all constraints are satisfied. Moreover, their approach supports *flexible constraints*. That is, it can providing solutions even if contradictory constraints that cannot be satisfied at the same time are used.

Finding domain design errors. Queralt and Teniente [68] presented an approach for finding conceptual errors in UML schemas and OCL constraints (e.g., a schema from which some classes may never be instantiated without causing inconsistencies). They transform UML class diagrams and OCL constraints to logic formulas such that standard reasoning engines can be employed. The reasoning engine may then be used to find a sample instantiation of the schema which is consistent. In contrast to the automatic inconsistency fixing approaches discussed in Sect. 4.2, the generated solution is not a UML class diagram, but an object diagram which conforms to the class diagram (i.e., they use the schema defined with UML as the metamodel for the reasoning). As with the approaches discussed in Sect. 4.2, CDM can be used for generating and updating the constraints which are used for reasoning (i.e., it is an enabling technology).

9 Conclusions and future work

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

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

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

For future work, we plan to further investigate the usability of the approach and to integrate approaches for automatic inconsistency fixing in our prototype. Moreover, we plan to do further research on finding contradictory constraints and the effects of CDM when used with bidirectional transformations.

Acknowledgments The research was funded by the Austrian Science Fund (FWF): P21321-N15 and P23115-N23, the EU Marie Curie Actions—Intra European Fellowship (IEF) through project number 254965, and FWF Lise-Meitner Fellowship M1421-N15.

References

1. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. IEEE Comput **39**(2), 25–31 (2006)
2. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
3. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–646 (2006)
4. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
5. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
6. Object Management Group, Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/
7. Ruscio, D.D., Eramo, R., Pierantonio, A.: Model transformations. In: SFM, pp. 91–136 (2012)
8. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Softw. Syst. Model. **9**(1), 7–20 (2010)
9. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: IWPSE, pp. 13–22 (2005)

المنارة للاستشارات
www.manaraa.com

10. Stevens, P.: A landscape of bidirectional model transformations. In: GTTSE, pp. 408–424 (2007)
11. Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W.: Flexible and scalable consistency checking on product line variability models. In: ASE, pp. 63–72, ACM (2010)
12. van Amstel, M., Bosems, S., Kurtev, I., Pires, L.F.: Performance in model transformations: experiments with ATL and QVT. In: ICMT, pp. 198–212 (2011)
13. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: ICMT, pp. 260–283 (2009)
14. Object Management Group: Object Constraint Language (OCL). http://www.omg.org/spec/OCL/
15. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Constraint-driven modeling through transformation. In: ICMT, pp. 248–263 (2012)
16. Reder, A., Egyed, A.: Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML. In: ASE, pp. 347–348, ACM (2010)
17. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Softw. Eng. **37**(2), 188–204 (2011)
18. Object Management Group: Unified Modeling Language (UML) superstructure. http://www.omg.org/spec/UML/2.4.1/Superstructure (2012)
19. Micskei, Z., Waeselynck, H.: The many meanings of uml 2 sequence diagrams: a survey. Softw. Syst. Model. **10**(4), 489–514 (2011)
20. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: ICSE, pp. 455–464 (2003)
21. Reder, A., Egyed, A.: Computing repair trees for resolving inconsistencies in design models. In: ASE, pp. 220–229 (2012)
22. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: ICFEM, pp. 198–213 (2012)
23. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. J. Syst. Softw. **83**(2), 283–302 (2010)
24. Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E.: Flexible consistency checking. ACM Trans. Softw. Eng. Methodol. **12**(1), 28–63 (2003)
25. da Silva, M.A.A., Mougenot, A., Blanc, X., Bendraou, R.: Towards automated inconsistency handling in design models. In: CAiSE, pp. 348–362 (2010)
26. Reder, A., Egyed, A.: Incremental consistency checking for complex design rules and larger model changes. In: MoDELS, pp. 202–218 (2012)
27. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model. **9**(1), 69–86 (2010)
28. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: MoDELS Workshops, pp. 158–171 (2009)
29. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: MoDELS, pp. 415–431 (2012)
30. Nöhrer, A., Egyed, A.: C2o configurator: a tool for guided decision-making. Autom. Softw. Eng. **20**(2), 265–296 (2013)
31. Kessentini, M., Sahraoui, H.A., Boukadoum, M., Omar, O.B.: Search-based model transformation by example. Softw. Syst. Model. **11**(2), 209–226 (2012)
32. Nöhrer, A., Reder, A., Egyed, A.: Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track. In: ICSE, pp. 864–867 (2011)
33. Puissant, J.P., Straeten, R.V.D., Mens, T.: Badger: A regression planner to resolve design model inconsistencies. In: ECMFA, pp. 146–161 (2012)
34. Hegedüs, Á., Horváth, Á., Ráth, I., Branco, M.C., Varró, D.: Quick fix generation for DSMLs. In: VL/HCC, pp. 17–24 (2011)
35. Manders, E.-J., Biswas, G., Mahadevan, N., Karsai, G.: Component-oriented modeling of hybrid dynamic systems using the generic modeling environment. In: MBD/MOMPES, pp. 159–168 (2006)
36. Ossher, H., Bellamy, R.K.E., Simmonds, I., Amid, D., Anaby-Tavor, A., Callery, M., Desmond, M., de Vries, J., Fisher, A., Krasikov, S.: Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges. In: OOPSLA, pp. 848–864. ACM (2010)
37. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Cross-layer modeler: A tool for flexible multilevel modeling with consistency checking. In: ESEC/SIGSOFT FSE, pp. 452–455 (2011)
38. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS, pp. 632–647 (2007)
39. Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H.: Supporting automatic model inconsistency fixing. In: ESEC/SIGSOFT FSE, pp. 315–324 (2009)
40. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: FASE, pp. 203–217 (2010)
41. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC, pp. 222–231. Sept 2008
42. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Automatically generating and adapting model constraints to support co-evolution of design models. In: ASE, pp. 302–305 (2012)
43. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: COPE—automating coupled evolution of metamodels and models. In: ECOOP, pp. 52–76 (2009)
44. Object Management Group: Meta-Object Facility (MOF). http://www.omg.org/mof/
45. Pohl, K., Böckle, G., van der Linden, F.: Software Product line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
46. Thaker, S., Batory, D.S., Kitchin, D., Cook, W.R.: Safe composition of product lines. In: GPCE, pp. 95–104 (2007)
47. Sun, J., Zhang, H., Li, Y.-F., Wang, H.H.: Formal semantics and verification for feature modeling. In: ICECCS 2005, pp. 303–312 (2005)
48. Mendonca, M., Wasowski, A., Czarnecki, K.: SAT-based analysis of feature models is easy. In: SPLC, pp. 231–240 (2009)
49. Mazo, R., Lopez-Herrejon, R.E., Salinesi, C., Diaz, D., Egyed, A.: Conformance checking with constraint logic programming: the case of feature models. In: COMPSAC, pp. 456–465 (2011)
50. Lopez-Herrejon, R.E., Egyed, A.: Detecting inconsistencies in multi-view models with variability. In: ECMFA, pp. 217–232 (2010)
51. Capozucca, A., Cheng, B.H., Guelfi, N., Istoan, P.: Barbados crash management system. http://www.cs.colostate.edu/remodd/v1/content/bcms-spl-case-study-proposition-based-cloud-component-approach, 2011. [Accessed 1-August-2012]
52. ReMoDD Team: Repository for model driven development (ReMoDD). http://www.cs.colostate.edu/remodd/v1/ (2011)
53. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: SFM, pp. 399–437 (2012)
54. Büttner, F., Bauerdick, H., Gogolla, M.: Towards transformation of integrity constraints and database states. In: DEXA Workshops, pp. 823–828 (2005)
55. Giese, M., Larsson, D.: Simplifying transformations of OCL constraints. In: MoDELS, pp. 309–323 (2005)
56. Bajwa, I.S., Lee, M.G.: Transformation rules for translating business rules to OCL constraints. In: ECMFA, pp. 132–143 (2011)
57. Object Management Group: Semantics of Business Vocabulary and Rules (SBVR). http://www.omg.org/spec/SBVR/

المنارة للاستشارات
www.manaraa.com

58. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. **8**(1), pp. 21–43 (2009)
59. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. In: ICMT, pp. 213–228 (2009)
60. Sasano, I., Hu, Z., Hidaka, S., Inaba, K., Kato, H., Nakano, K.: Toward bidirectionalization of ATL with GRoundTram. In: ICMT, pp. 138–151 (2011)
61. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: SLE, pp. 183–202 (2010)
62. Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In: ICMT, pp. 123–137 (2010)
63. Tisi M., Perez S.M., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: MoDELS, pp. 32–46 (2011)
64. Thüm, T., Batory, D.S., Kästner, C.: Reasoning about edits to feature models. In: ICSE, pp. 254–264 (2009)
65. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration using feature models. In: SPLC, pp. 266–283 (2004)
66. Saxena, T., Karsai, G.: MDE-based approach for generalizing design space exploration. In: MoDELS, pp. 46–60 (2010)
67. Horváth, Á., Varró, D.: Dynamic constraint satisfaction problems over models. Softw. Syst. Model. **11**(3), 385–408 (2012)
68. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM Trans. Softw. Eng. Methodol. **21**(2), 13:1–13:41 (2012)

Author Biographies

Andreas Demuth received his Bachelor's degree in Computer Science and his Master's degree in Software Engineering from the Johannes Kepler University Linz (JKU), Austria. Currently, he is working as a researcher, funded by the Austrian Science Fund (FWF) project P23115-N23, at JKU's Institute for Systems Engineering and Automation. His main research interests include model-driven engineering, especially flexible and multilevel modeling as well as model transformations, incremental consistency checking, and evolving software product lines.

Roberto Erick Lopez-Herrejon is a Lise Meitner Fellow (2012–2014) sponsored by the Austrian Science Fund (FWF). From 2010–2012, he held an FP7 Intra-European Marie Curie Fellowship (2010–2012) on a project for consistency and composition of variable systems with multiple view models. He obtained his PhD from the University of Texas at Austin in 2006, funded in part by a Fulbright Fellowship sponsored by the U.S. State Department. From 2005 to 2008, he was a Career Development Fellow at the Software Engineering Centre of the University of Oxford sponsored by Higher Education Founding Council of England (HEFCE). His expertise is software product lines, variability management, feature oriented software development, model driven software engineering, and consistency checking.

Alexander Egyed is a Professor at the Johannes Kepler University (JKU), Austria. He received his Doctorate degree from the University of Southern California, USA and previously worked for Teknowledge Corporation, USA (2000–2007) and the University College London, UK (2007–2008). He is most recognized for his work on software and systems modeling—particularly on consistency and traceability of models. Dr. Egyed's work has been published at over a hundred refereed scientific books, journals, conferences, and workshops, with over 3000 citations to date. He was recognized as the 10th best scholar in software engineering in Communications of the ACM and was named an IBM Research Faculty Fellow in recognition to his contributions to consistency checking, received a Recognition of Service Award from the ACM, a Best Paper Award from COMPSAC, and an Outstanding Achievement Award from the USC. He has given many invited talks including four keynotes, served on scientific panels and countless program committees, and has served as program (co-) chair, steering committee member, and editorial board member of SoSyM and other journals.

المنارة للاستشارات
www.manaraa.com

Copyright of Software & Systems Modeling is the property of Springer Science & Business Media B.V. and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.

المنارة للاستشارات

www.manaraa.com