Constraint-driven Modeling through Transformation

Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed

Institute for Systems Engineering and Automation
Johannes Kepler University (JKU)
Linz, Austria
{andreas.demuth|roberto.lopez|alexander.egyed}@jku.at

Abstract. In model-driven software engineering, model transformations play a key role since they are used to automatically generate and update models from existing information. However, defining concrete transformation rules is a complex task because the designer has to cope with incompleteness, ambiguity, bidirectionality, and rule dependencies. In this paper, we propose a vision of Constraint-driven Modeling in which transformation is used to automate the generation of model constraints instead of generating entire models. Three illustrative scenarios show how this approach addresses common transformation issues and how designers can benefit from using model constraints and guidance. We developed a proof-of-concept implementation that covers an important part of this vision and thus demonstrates its feasibility. The implementation also suggests that a constraint-driven transformation is efficient and scales even with increasing numbers of involved models.

1 Introduction

With the increasing 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]) and rich transformation languages are available, such as ATL [5] or QVT [6], which define *transformation rules* that are executed by a *transformation engine* to generate models. Since the source models of transformations are likely to be manually edited during development, re-transformations are necessary to update the corresponding generated models. However, such a re-transformation of non-trivial models can be time expensive and may affect the modeler's normal workflow [7, 8]. Hence, incrementality is required to allow partial model updates without complete re-transformations in order to achieve acceptable performance when working with large, non-static models [8]. To date, various sophisticated transformation techniques exist that produce excellent results as long as the generated models are static and there are no uncertainties.

However, problems arise when these requirements are not fulfilled. For example, a common issue with re-transformations – even when performed incrementally – is the inevitable loss of manual changes to the generated models. The issue is similar with *bidirectional transformations* [9, 10], which are often used to synchronize models or to keep them consistent, when both involved models are edited concurrently. Furthermore, there are situations where it cannot be ensured that traditional approaches will generate the desired models because of ambiguity, uncertainties, and the fact that certain information neither is available at the time the transformation rules are written nor can be derived from the involved models when those rules are executed.

In this paper, we propose *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 with re-transformations, uncertainties, and bidirectionality. CDM relies on incremental model transformations to generate *constraints* from existing models that represent the invariants that the generated models should meet. Such constraints, written in a *constraint language* (e.g., the *Object Constraint Language (OCL)* [11]), are validated by a *consistency checker* on a given model. The provided *guidance*, which is derived from the generated constraints and existing inconsistencies, helps designers to stepwise transform the initially generated model to a version that matches the desired characteristics by pointing out inconsistencies (i.e., aspects of the model that do not satisfy invariants). Such guidance can be either the information which elements are causing inconsistencies, or suggestions of model changes (*options*) that can be performed to restore consistency. To obtain an initial, yet incomplete version of the desired model to start working with, a traditional batch model transformation with unambiguous rules can be used to generate a skeleton. Thus, CDM can be seen as a complement to traditional model transformation.

We evaluated our approach and showed its feasibility by implementing a prototype that generates constraints, enforces them incrementally, and informs the user about existing inconsistencies. Performance tests with large industrial models of up to 162,237 model elements previously showed the scalability of constraint validation [12]; our tests with these models show that the median times for incremental transformation and constraint generation are under .07 milliseconds. Thus, the approach scales and provides instant user feedback when involved models are edited.

2 Running Example

To illustrate our work, we first present two incremental changes that are challenging for common model transformation approaches.

Let us consider the sequence and class diagrams shown in Fig. 1(a) and Fig. 1(b) respectively. In Fig. 1(a), 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 `LightSwitch` provides a method also named `activate`. At first glance, it looks like a simple transformation can be used to automatically add the method `activate` to class `LightSwitch` in Fig. 1(b) whenever a message is added to a sequence diagram

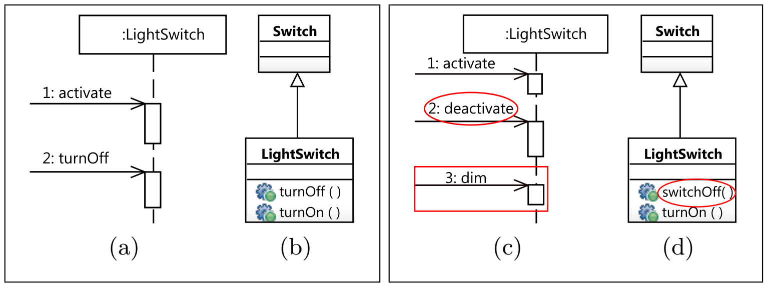


Fig. 1. Two UML models (a) and (b), and evolved versions (c) and (d).

whose name does not match any method in the class. An example for such a transformation rule written in ATL-like syntax is shown in Listing 1.

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 cannot be answered automatically. The only possibility would be to make an assumption (e.g., always add the method to the specified class to be on the safe side), which leads to the generation of potentially unintended models where methods are not declared in the desired place or where methods are unnecessarily overridden.

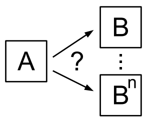
3 Constraint-driven Modeling

Common transformation languages usually describe the steps that have to be performed to generate new models from existing ones. The previous section illustrated that it can be difficult or even impossible to writing transformation rules that automate complex decisions or always lead to desired results.

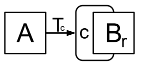
Intuitively, and in contrast with standard model transformations, we propose to generate constraints on a model (to guide designers) rather than generating the model itself whenever precise transformation results cannot be derived. For example, the added message `activate` on the source model should 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 further

```
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.



(a) Model transformation.



(b) Constraint transformation.

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

actions are required to deal with this problem – actions that must either come from a human or be derivable from other transformations.

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

$$A \xrightarrow{T_m} B_g \tag{1}$$

where A is called the source model, consisting of an arbitrary number of model elements. T_m is the transformation model, consisting of transformation rules, that is used to transform A to the generated model B_g.

We expanded this notation and define our approach as:

$$A \xrightarrow{T_c} C \rightsquigarrow B_r \tag{2}$$

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 of T suggest, this set of rules no longer generates a model (i.e., B_g), but instead it contains transformation rules that are applied to A in order to generate constraints (i.e., the constraint model C). This constraint model consists of a set of constraints that are enforced by an incremental consistency checker on the model B_r, as indicated by the curvy arrow from C to B_r. The model B_r is no longer the generated model but is now called the restricted model, as indicated by the subscript r, that is either consistent or inconsistent with the constraint model C, and therefore a valid or invalid solution of the modeling problem.

Note that an initial version of B_r may be generated through a traditional transformation (analogous to B_g) 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 Section 2 where we illustrated that choosing the right class for a required method cannot be fully automated. The traditional approach shown in Fig. 2(a) automatically generates one of several possible models and we could at most use heuristics for deciding on which

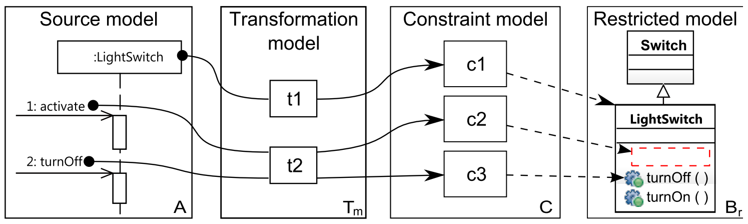


Fig. 3. Application of approach to models from Fig. 1(a) and Fig. 1(b).

transformation to use (which never guarantees correctness). However, while the knowledge contained in Fig. 1(a) is insufficient to generated a correct update to the class diagram, it is sufficient to generate a correct constraint on said diagram. Such constraints can be generated by transformation rules that are 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, we provide 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.

```
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.

Note that, even though we use ATL-like syntax for this example, our approach can be used with any transformation language. After applying these rules to the motivating example from Section 2 as illustrated in Fig. 3 and according to (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 and we have existing tool support for incrementally validating OCL constraints. Nonetheless, in principle any constraint language and consistency checker may be used.

Fig. 2(b) illustrates the basic concept of the constraint-driven modeling approach. It is noteworthy that the approach does not modify the restricted model. It simply restricts it. The generated restriction – depicted as partial frame with rounded corners around the restricted model B_r – may be light in that there are various options on how to change the restricted model. In such as case, the designer has the freedom to decide which of the options is the desired one (e.g., add `activate` to `LightSwitch` or `Switch`) with the knowledge that the approach notifies/prevents options that are invalid. In the most extreme case, the restrictions may be severe enough to allow for one option only. In such a case, the approach could automatically select this option with the knowledge that it is the one and only right option (e.g., if `LightSwitch` had no parent class then there is a single option only).

3.2 Incremental Constraint Model Management

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

Source model update. The transformation approach we use supports incrementality to allow updates of the constraint model without performing a complete re-transformation of the source model. When A is updated to A', we can write this as

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

where ΔA is a sequence of modifications done to elements in A (e.g., add a new model element). ΔA is used as input for the transformation model to generate the set ΔC, as shown in (4).

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

ΔC includes pairs of constraints and actions (i.e., $\{add, remove\}$) 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:

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

Let us consider the evolution of the models shown in Fig. 1(a) and Fig.1(b) to the versions shown in Fig. 1(c) and Fig. 1(d) 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 $\langle\langle Message2, update\rangle, \langle Message3, new\rangle\rangle$.

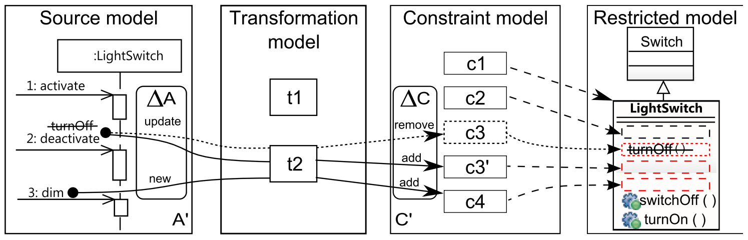


Fig. 4. Update of constraint model after changes in source model.

To build ΔC, the transformation engine executes the applicable transformation rules for the elements in ΔA (i.e., message #2 and message #3) to generate the corresponding constraints, as defined in (4) and shown in Fig. 4. 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 $\langle Message3, new \rangle$, 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|m.name='dim')
```

At this point, ΔC is $\{\langle c3, remove \rangle, \langle c3', add \rangle, \langle c4, add \rangle\}$. When these changes are applied to $C = \{c1, c2, c3\}$ as defined in (5) and shown in Fig. 4, 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. 4 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`.

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

$$C' \rightsquigarrow B_r \tag{6}$$

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 space of a modeling problem to initially include all possible instances of a metamodel (there are likely infinite). When a constraint is validated, it determines whether a model meets those characteristics. Therefore, applying a constraint decreases the size of the solution space and the validation result shows whether a specific model is part of the solution space.

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 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:

$$val_T(B_r, C) = \bigwedge_{1 \leq i \leq |C|} val(B_r, c_i) \tag{7}$$

If at least one constraint validation $val(B_r, c_i)$ returns $false$, the overall status of B_r is also $false$ and therefore 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.

Since constraints are composed of expressions that are evaluated on only the restricted model, direct dependencies among constraints typically do not exist and are not considered here. The addition of a new constraint thus does not affect the validity of existing constraints. This leads us to the conclusion that **constraints are independent of each other**. 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.

3.4 Providing Guidance

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 is violating which constraint). Based on data captured during constraint validation, the consistency checker can determine which model elements are actually causing the inconsistency. Hence, it can inform the designer about the locations of error-causing elements.

Constraint-driven modeling may appear inferior to traditional transformation in that it never generated model elements in the restricted model. However, there is currently considerable progress in automatically suggesting repairs to inconsistencies in design models. Based on a specific constraint 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 user as a list of options. If the restrictions are unambiguous, only a single option remains and it could be applied automatically (much like transformation). For example, the action `<add method "dim" to`

`class "LightSwitch">` is an option for removing the inconsistency caused by the absence of the method `dim` in the `LightSwitch` class and the constraint $c4$. Thus, using constraints does not only expose inconsistencies but it also enables user guidance to help understanding and solving them. Note that incorporating source model data makes a constraint much more specific and expressive when presented to the user than a manually written, generic constraint that relies on metamodel data and functions.

Nevertheless, dependencies between constraints in terms of required model characteristics and corresponding model elements can occur (e.g., $c1$ requires a class `LightSwitch` and $c2 - c3$ require specific methods in this class). Creating additional inconsistencies can therefore be necessary to achieve overall model consistency. More research in automated fixing of design models based on constraint violations is needed to automate this. However, we believe that this problem is solvable and the focus of our future work.

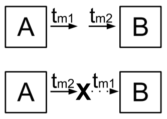
Guidance is however not limited to inconsistencies. For each constraint, its source as well as the locations where it is validated are available and can be presented to the user. When the source model is edited during development, the constraints that are affected by those changes can also be highlighted. 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 Additional Benefits of Constraint-driven Modeling

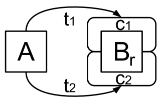
Now, we want to show several additional scenarios that benefit from constraint-driven modeling in context of rule-scheduling, model merging, and bidirectionality.

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. 1(a) contains an instance of the class `LightSwitch`. Therefore, let us assume that transformation rule t_{m1} generates a corresponding class if no such class exists in the diagram in Fig. 1(b). As we have discussed in Section 2, the sequence diagram requires the class `LightSwitch` to provide a method `activate`. Let transformation rule t_{m2} generate this method in `LightSwitch`[1]. 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. 5(a) where the bottom transformation

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



(a) Model transformation.



(b) Constraint transformation.

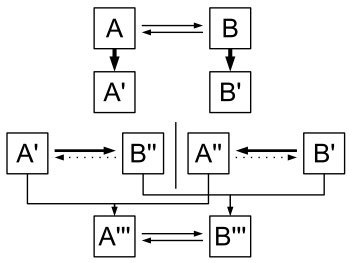
Fig. 5. From dependent transformation rules (a) to independent ones (b).

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].

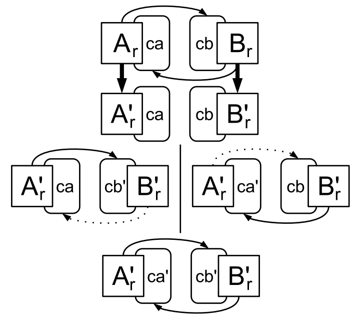
The constraining approach, shown in Fig. 5(b), is free of scheduling issues because constraints cannot directly depend on other constraints and the order of transformation is not relevant for the transformation results, as discussed in Section 3.3. Hence, the rules t_1 and t_2 we have previously defined can be applied in any order. 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.

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. 1(c), the name of the highlighted message has been changed from `turnOff` (see Fig. 1(a)) to `deactivate`. Concurrently, the corresponding method in the class diagram was changed from `turnOff()` (see Fig. 1(b)) to `switchOff()`, as highlighted in Fig. 1(d). 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 afterwards). A possible solution would be the concurrent execution of the transformations followed by a merge of the updated models (A' and B') and the resulting generated models (A'' and B''), as illustrated in Fig. 6(a), that generated A''' and B'''. However, this requires a complex merging strategy and is likely to produce models that still require manual adaptation.

The solution of the constraint transformation approach is shown in Fig. 6(b). We can see that our approach still has to decide which change to process first. However, because only constraint models are updated, the restricted models A'_r and B'_r are not changed and can therefore still be processed to perform constraint



(a) Model transformation.



(b) Constraint transformation.

Fig. 6. From bidirectionality (a) to unidirectional constraint transformation (b).

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. 1(c) should be named `switchOff` (from Fig. 1(d)) and ii) the name of the method `switchOff` in Fig. 1(d) should be changed to `deactivate` (from Fig. 1(c)). The designer can then decide which of the elements should be renamed.

5 Validation

In this section we first discuss various aspects regarding the correctness of our approach and its results. Then, we present the results of a performance evaluation and finally discuss possible threats to validity.

5.1 Correctness

Based on the presented scenarios and the properties of constraints we showed that common transformation issues like rule-scheduling, race conditions, and model merging do not arise when constraint models are generated through transformation and that those models can be updated easily.

Nevertheless, the correctness of the applied constraints and the provided user guidance is determined by the correctness of the manually written transformation rules, the used source models, and the transformation language implementation – as with traditional approaches.

Errors in both the source models and the applied rules lead to errors in the generated model. As with traditional approaches, such errors also affect the generated model (i.e., the constraints) in our approach. However, designers can inspect arbitrary constraints and decide whether the constraints are correct. By using a transformation mechanism that creates traceability links between source model elements, transformation rules and the generated constraints automatically – as we did in our prototype implementation – designers can use faulty constraints to detect errors in the source model or the transformation rules and fix them. Moreover, faulty constraints can simply be ignored or deactivated, which means that contradicting constraints do not prevent designers from constructing the desired model. Generally, existing errors are never incorporated in the restricted model automatically when our approach is used.

5.2 Implementation

To support the vision proposed in this work, we implemented a proof-of-concept prototype tool and applied it to several domains. The prototype, based on the *Cross-Layer Modeler (XLM)* [13], investigates the constraint generation/validation and the presentation of consistency information and constraints, but not the inconsistency repair. The latter is future work. The tool employs the *Model/-Analyzer* [14] consistency checker to validate constraints that are automatically generated through incremental transformation from templates and are managed and updated incrementally. We added components to the XLM to support multiple different source, constraint and restricted models simultaneously, which requires the management of multiple, parallel running consistency checkers.

5.3 Performance Evaluation

Basically, our approach has two phases: i) generating constraints, and ii) validating them. For the latter, the performance of the employed consistency checker was thoroughly evaluated with 34 large-scale industrial models of up to 162,237 model elements and complex constraints in [12, 15]; it was shown that most changes in restricted models are processed in less than one millisecond. To show that also the former is fast and scalable, two different test setups were used:

Test I Replacing ambiguous transformations – as discussed in Section 3
Test II Replacing merges – different sources restrict a single model

Test I simulated simple unidirectional transformations in the scenario we described in Section 3 and was performed with 20 of the industrial models we previously used in [12, 15]. Test II determined 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 gradually increased and more models become involved. For Test II, we generated random models and constraints with similar characteristics as those we used for constraint validation testing because the scenario required multiple, related source models and our industrial models were designed as independent models.

For all tests, single model elements were added to or removed from a source model, which forced an incremental constraint model update (i.e., the addition or

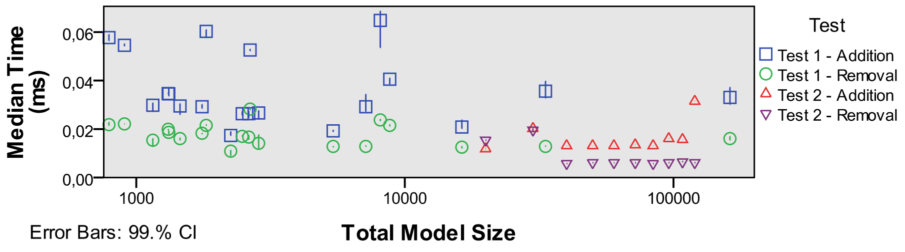


Fig. 7. Median processing times for constraint updates.

removal of exactly one constraint). Between 1 and 10 source models were used for Test II, the required time for processing the source model change and performing constraint model updates was measured. For the evaluation we used an Intel Core i5-650 machine with 8GB of memory running Windows 7 Professional. In Fig. 7 the median times for 1,000 runs per test with a 99% confidence interval are shown. Note that the increasing number of source models in Test II does not affect the processing time significantly and that the median times for the addition of elements are between .01 and .07 milliseconds in both tests. Element removal takes between .006 and .03 milliseconds and is indeed faster than element addition because no transformations are required. These numbers show that our approach can update constraint models instantly and does scale for increasing numbers of source models. The similar results for Test I and Test II suggest that our random models for Test II were a valid choice for testing scalability. Note that bidirectional transformations are split into unidirectional ones in our approach, thus there was no need for testing bidirectional transformations explicitly.

5.4 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. Another threat to the validity of our vision is the automated derivation and execution of options to remove existing inconsistencies. Even though basic traceability information – which is always available – provides a certain amount of guidance, a key aspect of constraint-driven modeling is the automated suggestion of valid options to remove inconsistencies. However, this is still an open research question that we want to address in future work. Finally, we have yet to develop an efficient strategy for finding contradictions between constraints and fixing them automatically.

6 Related Work

Model Transformation is a very active field of research and several topics related to our work have been discussed. Regarding bidirectionality, Sasano et al. [16] developed a system to perform bidirectional transformations with ATL, and Stevens [17] focused on bidirectionality for QVT. Cicchetti et al. [18] developed the bidirectional transformation language *JTL* that supports the specification of non-bijective transformations so that one model can be mapped to a set of other models. We tackle the complexity of bidirectional transformations by using unidirectional transformations and constraints.

In terms of incrementality and execution speed, Jouault and Tisi [19] 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 [14]. In [20], 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.

Regarding automated design error fixes, the generation of fixing actions was discussed by Xiong et al. [21]. They developed a language called *Beanbag* that allows the definition of constraints and fixing behavior at the same time. With our approach, such Beanbag programs can be generated automatically. Saxena and Karsai [22] 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.

7 Conclusions and Future Work

In this paper we presented an incremental and generic approach that uses model transformation to automatically generate constraint models. We showed that constraints are independent, and constraint validation does not require a fixed order of execution. We then discussed how model transformation issues like ambiguity, rule-scheduling, model merging, and bidirectionality are addressed and how the approach enables user guidance and encourages the use of domain-knowledge to solve specific modeling problems. 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. 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 implement the automated derivation of options for removing inconsistencies and the checking for contradicting constraints.

Acknowledgments

The research was funded by the Austrian Science Fund (FWF): P21321-N15 and the EU Marie Curie Actions – Intra European Fellowship (IEF) through project number 254965.

References

1. D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
2. S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.
3. K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–646, 2006.
4. T. Mens and P. V. Gorp, "A taxonomy of model transformation," *Electr. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, 2006.
5. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008.
6. Object Management Group, "Query/View/Transformation (QVT)." http://www.omg.org/spec/QVT/.
7. M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider, "Flexible and scalable consistency checking on product line variability models," in *ASE*, pp. 63–72, ACM, 2010.
8. M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires, "Performance in model transformations: Experiments with ATL and QVT," in *ICMT*, pp. 198–212, 2011.
9. P. Stevens, "A landscape of bidirectional model transformations," in *GTTSE*, pp. 408–424, 2007.
10. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *ICMT*, pp. 260–283, 2009.
11. Object Management Group, "Object Constraint Language (OCL)." http://www.omg.org/spec/OCL/.
12. A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.
13. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Cross-layer modeler: A tool for flexible multilevel modeling with consistency checking," in *ESEC/SIGSOFT FSE*, pp. 452–455, 2011.
14. A. Reder and A. Egyed, "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *ASE*, pp. 347–348, ACM, 2010.
15. I. Groher, A. Reder, and A. Egyed, "Incremental consistency checking of dynamic constraints," in *FASE*, pp. 203–217, 2010.
16. I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano, "Toward bidirectionalization of ATL with GRoundTram," in *ICMT*, pp. 138–151, 2011.
17. P. Stevens, "Bidirectional model transformations in QVT: Semantic issues and open questions," in *MoDELS*, pp. 1–15, 2007.
18. A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "JTL: A bidirectional and change propagating transformation language," in *SLE*, pp. 183–202, 2010.

19. F. Jouault and M. Tisi, “Towards incremental execution of ATL transformations,” in *ICMT*, pp. 123–137, 2010.
20. M. Tisi, S. M. Perez, F. Jouault, and J. Cabot, “Lazy execution of model-to-model transformations,” in *MoDELS*, pp. 32–46, 2011.
21. Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, “Supporting automatic model inconsistency fixing,” in *ESEC/SIGSOFT FSE*, pp. 315–324, 2009.
22. T. Saxena and G. Karsai, “MDE-based approach for generalizing design space exploration,” in *MoDELS*, pp. 46–60, 2010.