Supporting the Co-Evolution of Metamodels and Constraints through Incremental Constraint Management

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. Design models must abide by constraints that can come from diverse sources, like metamodels, requirements, or the problem domain. Modelers intent to live by these constraints and thus desire automated mechanism that provide instant feedback on constraint violations. However, typical approaches assume that constraints do not evolve over time, which, unfortunately, is becoming increasingly unrealistic. For example, the co-evolution of metamodels and models requires corresponding constraints to be co-evolved continuously. This demands efficient constraint adaptation mechanisms to ensure that validated constraints are up-to-date. This paper presents an approach based on constraint templates that tackles this evolution scenario by automatically updating constraints. We developed the Cross-layer Modeler (XLM) approach which relies on incremental consistency-checking. As a case study, we performed evolutions of the UML-metamodel and 21 design models. Our approach is sound and the empirical evaluation shows that it is near instant and scales with increasing model sizes.
Keywords: Co-evolution, metamodeling, consistency-checking

1 Introduction

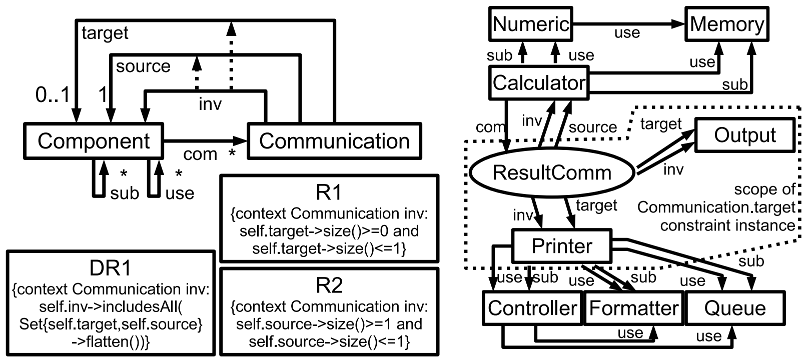
In *Model-Driven Development (MDD)* [1], metamodels play a key role as they reflect real-world domains and define the language of models as well as the constraints these models must satisfy. Over the past years, a trend has emerged that calls for design tools with adaptable metamodels – to customize the tool to a particular discipline, domain, or even application under development. Indeed, those metamodels must evolve continuously; for example, to reflect changes of a domain or to meet new business needs. Refactorings that improve a metamodel's structure and usability are also common. Nowadays, a range of "flexible" design tools with adaptable metamodels are available to support such scenarios (e.g., [2,3]).

Co-evolution of models denotes the process of concurrently evolving metamodels and their models – a process that is non trivial since inconsistent co-evolution may cause models and metamodels to drift apart. Several incremental approaches have been proposed to support this process (e.g., [4]).

However, metamodels also impose constraints onto models. When the metamodels evolve, so must the constraints – a scenario that has been largely overlooked so far. For example, the *Unified Modeling Language (UML)* [5] is supported by hundreds of well-formedness rules and the community augmented these with even more consistency rules. Moreover, *UML Profiles*, which may also include consistency rules, are commonly used to extend the UML and adapt it to specific domains [6]. Modifying the UML metamodel thus impacts these constraints. Previously semantically and syntactically correct constraints may become incorrect after structural or semantic metamodel changes; or new constraints may appear. It is crucial to extend the notion of co-evolution to include the continuous maintenance of constraints such that only correct constraints are enforced on design models. Of course, it is also crucial to have available a consistency checker that is not only able to react to design model changes but also to metamodel/constraint changes. Generating and adapting constraints incrementally as well as checking them incrementally are thus pre-requisites to ensure that designers are always given instant and reliable feedback on the validity of their modeling work.

State-of-the-art consistency checkers are commonly employed to validate constraints and determine whether a model is consistent with respect to its metamodel. Most consistency checkers rely on an existing set of constraints for performing the validation [7, 8]. It is common to write these constraints manually, typically in a standardized language such as the *Object Constraint Language (OCL)* [9]. Often, constraints are also "hard-coded" into modeling tools. Although the automatic co-evolution of metamodels and models has become an active field of research, the issue of co-evolving constraints is not well addressed. Incremental consistency checkers typically do not support the live updating of constraints and little support for updating outdated constraints is available.

This paper describes an approach for the co-evolution of metamodels and their constraints that uses constraint templates and a template engine to automatically and incrementally manage constraints – it is an extension of a previously published idea-paper [10]. New contributions include the in-depth illustration and discussion of the approach and a prototype implementation (the *Cross-Layer Modeler (XLM)* [11]) that leverages from our previous work on the Model/Analyzer [8], an efficient incremental consistency checker. Moreover, we evaluated our approach by using the XLM for automating the generation of constraints that ensure the structural integrity of UML models and by performing sample evolutions of the UML metamodel. Tests were performed on 21 large industrial UML models of up to 36,205 model elements. While the UML is not the primary motivation for our approach (it changes occasionally only), it is like any modeling language in that it must adhere to a metamodel and imposes constraints. UML metamodel changes thus impose the same kind of challenges. The fact that the UML language is far from trivial and we have available large-scale, industrial models thus make it a very suitable environment to test the scalability of our approach and the XLM tool. The results show that our approach is correct and works efficiently even as model sizes increase.



```
R1
{context Communication inv:
self.target->size()>=0 and
self.target->size()<=1}

DR1
{context Communication inv:
self.inv->includesAll(
Set{self.target,self.source}
->flatten())}

R2
{context Communication inv:
self.source->size()>=1 and
self.source->size()<=1}
```

(a) Metamodel (b) Model

Fig. 1: Metamodel and model of component-based system with constraints.

2 Example and Motivation

We use an excerpt of a simple metamodel, shown in Fig. 1a, to illustrate our work. The metamodel consists of two elements: `Component` and `Communication`. Every `Component` can include an arbitrary number of `sub`-components and can directly `use` an undefined number of other components. A `Communication` expresses a data exchange from a `source` to at most one `target` component. Components can have an arbitrary number of open communications (`com`).

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

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

I: Metamodel directly. First, we use intuitive constraints that check the cardinality of references. For each reference, we create a constraint (e.g., $R1$ or $R2$ in Fig. 1a) that ensures that every instance of the owning element is connected to the specified number of elements in a model (e.g., every instance of `Communication` must be connected to exactly one `Component` instance through a connection named `source`). We use the term *connected* in models to avoid ambiguity with *references* in the metamodel. Connections are depicted as named arrows in model diagrams. Constraints for references with unrestricted cardinalities (e.g., `com`) are not shown in Fig. 1a for readability reasons. Note that common modeling tools that use the *Eclipse Modeling Framework (EMF)* [12] for example either do not derive such constraints or have them “hard-coded”, meaning that changes cannot lead to constraint updates which effectively disables automated co-evolution.

II: Metamodel semantics. Next, we create a constraint for the derived reference (e.g., $DR1$ in Fig. 1a) to ensure that instances of the owning element are connected to all the elements that are reached through the aggregated references (e.g., for every instance of `Communication`, all elements that are connected to it via `source` and `target` must also be connected via `inv`). Note that our constraints make use of OCL collection iterations even though they are invoked on single objects. The issues arising because of the distinction between single and multi-object values in OCL have been discussed and identified in literature as a problem especially during evolution [13]. For the sake of generality, we use a consistency checker with an OCL interpreter that allows collection operations being used with single objects by performing the necessary conversions automatically.

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

As depicted in Fig. 1b, the metamodel from Fig. 1a is used to create a small model of a calculator system. The `Calculator` component has two subcomponents that are used directly: `Memory` and `Numeric`. The `Numeric` component also uses the component `Memory`. A `Printer` has three sub-components: `Formatter`, `Queue`, and `Controller`. It uses the `Queue` to store print jobs and informs the `Controller`, which retrieves data from the `Queue` and runs the `Formatter` before printing. Finally, there is an `Output` component to display information to the user. The `Calculator` uses a `Communication` element called `ResultComm` to send its results to the `Printer` and the `Output` components.

As indicated by the encircled area in Fig. 1b, the two `target` connections of `ResultComm` are causing an inconsistency because only one `target` is allowed according to the metamodel. Note that any consistency checking approach could detect inconsistencies in the model according to the constraints we defined above.

2.1 Incremental Consistency Checking

As the model size increases, so does the effort to check its consistency. Checking consistency in an entire model can easily become a time consuming task. Incremental consistency checking addresses this limitation by looking only at a subset of an entire model, namely the elements that change as a model evolves [14]. This set of elements can be either directly observed or calculated from differences between model versions [8,15]. The existing approach automatically defines *constraint instances* that validate whether specific model elements violate a given constraint [14]. The change impact *scope* of a constraint instance is the set of model elements that are used for calculating the constraint instance's validation result which are also computed automatically. For example, Fig. 1b shows

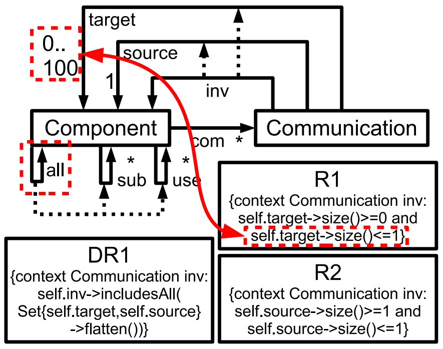

target
source
0..
100
1
inv
Component
com *
Communication
all
*
sub
*
use
R1
{context Communication inv:
self.target->size()>=0 and
self.target->size()<=1}
DR1
{context Communication inv:
self.inv->includesAll(
Set{self.target,self.source}
->flatten())}
R2
{context Communication inv:
self.source->size()>=1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

Fig. 2: The evolved metamodel.

a *constraint instance* of the `Communication` metaclass constraint $R1$ that requires communications to have exactly one `target`. The scope of this constraint instance consists of the two elements that are reached through the `target` reference to `Printer` and `Output`. Changes falling within scope of a constraint instance, like removing a `target`, would lead to a re-validation of the constraint instance. The Model/Analyzer automatically creates, re-evaluates, and destroys constraint instances according to changes in the model in Fig. 1b. However, if the metamodel were to change, consistency checkers would continue to validate the now-potentially-outdated design rules.

2.2 Co-Evolution Examples

Let us consider what happens when a metamodel changes. For instance, if the number of maximum targets of a `Communication` rises from `1` to `100` because new technologies allow multicasting of messages between components. Additionally, a new derived reference `all` is introduced to combine the `sub` and `use` references of a `Component`. These two changes are encircled with dashed lines in Fig. 2. These changes have the following consequences:

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

In the first case, $R1$ must be adapted by replacing the upper limit value `1` with literal `100`. Without this adaptation, the corresponding constraint instance, circled in Fig. 1b, would still incorrectly try to enforce an upper bound of `1`. In the second case, the inconsistency that neither `Calculator` nor `Printer` have the required connection `all` in our model is missed. To address this problem, a constraint that checks the derived reference `all` needs to be added.

A common way of dealing with co-evolution is to manually re-write the constraints after performing a metamodel modification. Although this approach can work in our example because of its small size and simple constraints, manually

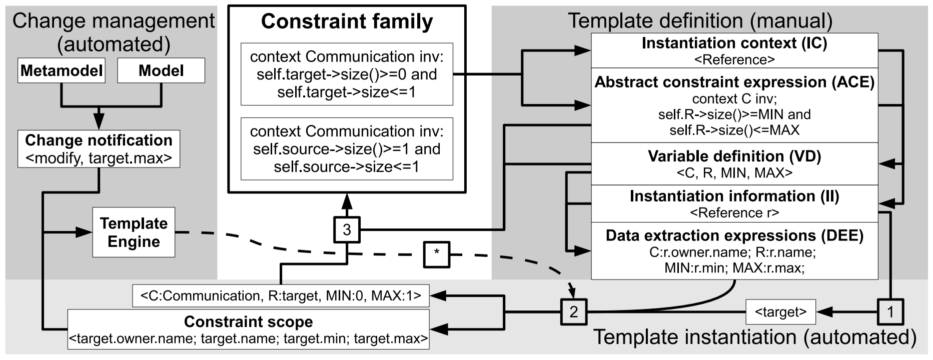

Change management (automated)
Metamodel
Model
Change notification
<modify, target.max>
Template Engine
Constraint family
context Communication inv: self.target->size()>=0 and self.target->size<=1
context Communication inv: self.source->size()>=1 and self.source->size<=1
Template definition (manual)
Instantiation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

Fig. 3: Example of steps performed during template definition, instantiation, and change management.

identifying and adapting affected constraints in more complex models is both time consuming and error-prone.

3 Constraint Templates and Template Engine

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

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

3.1 Template Definition

Templates are written manually by metamodel authors who are also in charge of maintaining and evolving metamodels. Before discussing the authoring process in detail, we discuss the structure of a template, as shown in Table 1, and the information it requires. The *instantiation context (IC)* defines for which elements, or combinations thereof, a template should be instantiated. The *abstract constraint expression (ACE)* is used to define the *family of constraints* generated from the template. A constraint family consists of constraints that share some static aspects (e.g., the structure) and have some variable parts that differ for each constraint. Thus, the ACE captures the static parts of the constraint family and also identifies the locations of variability which are also defined explicitly in the *variable definition (VD)*. The VD declares which parts of the ACE are interpreted as variables. To bind specific values to these variables, data has to

Table 1: Template structure

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

be read from specific elements that are available when the template is instantiated. These elements are specified in the *instantiation information (II)*. How the values for the variables are extracted from the elements is declared in *data extraction expressions (DEE)*. Let us now show how we can write a template $T1$ for the constraint family of $R1$ and $R2$.

Template for cardinalities. The top-right section "Template definition" in Fig. 3 illustrates the steps we perform next. The remainder of the figure depicts template instantiation and change management processes we discuss later. Template $T1$, shown in Table 2, creates a constraint for every instance of `Reference`, for example when the reference `target` is added to the class `Communication` during the initial modeling of our sample metamodel. Therefore, we define the IC of our template to be `<Reference>`. This means that we provide an instance of `Reference` to the template in order to create a new constraint. Note that templates are reusable for other metamodels that conform to the same metametamodel. We define the ACE by using the desired expression of one sample constraint of the constraint family (e.g., an OCL statement) and replacing all concrete values that are specific for a single instance with variables. In our example, we take the expression from the constraint $R1$ for the reference `Communication.target` in Fig. 1a:

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

Table 2: Definition of template T1

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

Table 3: Definition of template T2

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

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

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

as defined in Table 2 with the variable parts (VD) being `<C, R, MIN, MAX>`. As shown in Fig. 3, the instantiation information of $T1$ is `<Reference r>`.

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

Template for derived references. We use the same process to write template $T2$, as shown in Table 3, based on the constraint $DR1$ as an example for the constraint family that checks derived references.

As a simplification, we replaced the set of references (`Set{self.source, self.target}->flatten()`) from $DR1$ in Fig. 1a with a construct (`collect(x|self.{x})`) that allows us to aggregate the results of different references – based on a set of reference names – dynamically. When the template is instantiated for the derived reference `Communication.inv`, the resulting constraint is:

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

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

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

3.2 Template Instantiation

To enable a template, it is passed to the template engine that observes a specific model and handles template instantiation and updating. We will now discuss how the template $T1$ for checking reference cardinalities is instantiated when it is applied to the metamodel in Fig. 1a.

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

After applying our templates $T1$ and $T2$ to the initial version of our example metamodel from Fig. 1a, template $T1$ was instantiated once for every reference (i.e., five times in total), template $T2$ was instantiated once to generate the constraint for the only derived reference `inv` in the metamodel.

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

3.3 Change Management

In Section 2 we discussed the effects of two metamodel evolutions on the correctness of constraints. We will now present how such metamodel evolutions are handled automatically by the template engine.

Metamodel evolution. After every modification of the metamodel, the template engine is notified, as shown in the top-left box "Change management" in Fig. 3. The change notification includes information about the changed metamodel elements which the engine uses to determine the actions that are required to adapt the set of current constraints to the new version of the metamodel.

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

As an example, consider the metamodel version shown in Fig. 2. We first replaced the upper bound value `1` of the constraint $R1$ with the value `100`. The change notification that is passed to the engine indicates that the metamodel element `target.max` was modified. Since the scope of the constraint $R1$ contains the modified element, as discussed above, the engine detects that this constraint is affected by the modification. Because there are no other constraints that include the modified model element in their scope, $R1$ is identified as the only constraint that needs to be updated.

The update is performed by executing the data extraction expressions that added the modified metamodel element to the constraint's scope, as depicted by step (*) in Fig. 3, and replacing the outdated values in the constraint expression with the newly retrieved ones. In our example, `target.max` now returns the value `100`. Replacing the old value results in the new constraint expression

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

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

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

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

Model evolution. As we have discussed in Section 2.1, changes of a model typically lead to a re-validation of affected constraint instances. With our approach, such changes can affect the scopes of generated constraint instances. For example, imagine the addition of a new component as a `target` of `ResultComm` in Fig. 1b. Indeed, this may affect the consistency status of a constraint instance of $R1$. However, since such changes are handled entirely by the employed consistency checker, we omit a detailed discussion here and refer to [8].

4 Evaluation and Analysis

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

4.1 Prototype Implementation

For the evaluation, we developed the *Cross-Layer Modeler (XLM)* [11]. This tool allows working with models and their metamodels at the same time, which means that manipulations of the metamodel have immediate effects on the conformance of the model. The XLM leveraged from our previous work on the Model/Analyzer [8, 14] which supports efficient and scalable incremental consistency checking of arbitrary design constraints.

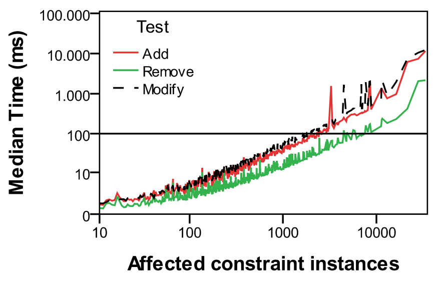
We extended the Model/Analyzer by adding an incremental template engine and the corresponding infrastructure to support the incremental creation, deletion and modification of constraints (based on meta model changes) which the Model/Analyzer then incrementally validates against model changes. Ten sample templates from different domains are available at the tool website [11].

4.2 Case Study: UML

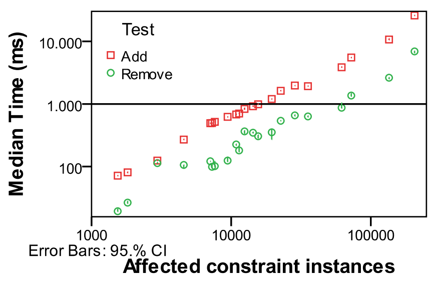
As our case study, we used templates and the Cross-Layer Modeler tool to automate constraint generation and updates for the UML. We chose UML as the subject because it is a well known and commonly used language for modeling software systems. We argue that its size and high level of complexity make it ideal for our purposes because the sample evolutions we performed simulate typical evolutions of metamodels in general. Additionally, numerous industrial software models are available [16]. We ran tests with 21 models with sizes from 3,077 to 36,205 model elements (i.e., instances of UML elements) and with different characteristics for our experiments. Every test was performed 100 times on an Intel Core i5-650 machine with 8GB of memory running Windows 7 Professional. The median and average values were used for analysis.

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

Category I. Metamodel evolution. Different metamodel modifications and common refactorings have been discussed in literature [4, 17–21]. During most common metamodel evolutions, references or attributes are added, removed, or are modified (e.g., the cardinality of an attribute is changed or an attribute is moved to another class). Therefore, we performed these kinds of



(a) Metamodel evolutions.



(b) Template addition/removal.

Fig. 4: Evaluation results.

evolutions with the UML metamodel. From this point on we will use the term *property* for references and attributes alike.

Scenario 1. Add new property. In the first scenario, a new property was added to every single element of the UML metamodel, which required the generation of a new constraint (as we discussed in Section 2.2 where we added a new derived reference to our sample metamodel). We investigated the total time required for performing the metamodel change, the required co-evolutions and the validation of the model with the new constraint. Note that for our statistics we only considered those changes that created constraints that could actually be validated with at least one model element (e.g., we ignored the addition of a new reference to `UseCase` if the model did not include any use cases). Fig. 4a shows the required processing times for changes that affected different numbers of constraint instances. 99% of all modifications took less than 166ms to finish and only 0.15% of all performed changes took more than 500ms. On average, changes took 12.5ms and the generated constraint was validated with 201 constraint instances in the model. For the addition of elements in this test we observed a Pearson correlation coefficient of 0.845 between the required time and the number of required validations. The correlation between T and the model size S, $P(T, S)$ was 0.099, which indicates that the processing time strongly depends on the validation effort needed for the new constraint and that it is independent from the model size.

Scenario 2. Remove existing property. In the second scenario, each test run started with the unmodified UML metamodel and exactly one property was removed, meaning that exactly one constraint became obsolete and was removed from the consistency checker. Again, only changes of metamodel elements that were actually used in the model were captured. 99% of all modification took less than 38.5ms. Only 0.1% of the modifications took longer than 250ms. On average, element removal took 4.5ms and 202 constraint instances were removed with the obsolete constraint. Fig. 4a shows that property removal is always faster than addition because there is no need for validating any constraint instances.

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

Category II. Model evolution. The incremental consistency checker that is used by the Cross-Layer Modeler, the Model/Analyzer, is highly scalable [16]. We previously evaluated the approach on 34 models with model sizes of up to 162,237 model elements and 24 types of consistency rules (constraints). Empirical evaluation showed that the consistency checking part requires only 1.4ms to re-evaluate the consistency of the model after a change for typical UML consistency and well-formedness constraints [22]. The data indicates that the additional change processing infrastructure does not impose a significant performance penalty.

Category III. Template addition and removal. Even though adding, removing, or changing a template is a task performed less often than metamodel evolutions, we still investigated this aspect. Since the addition of a new template requires a full scan of the metamodel to create all possible constraints and a complete initial validation of the model we expected this task to be more time consuming than processing changes incrementally. The processing times for the addition and removal of the templates we used in Category I to the UML metamodel that caused the generation or removal of different numbers of constraint instances are shown in Fig. 4b. Adding a template took less than 5,700ms in 90% of our tests, in only 8% of the tests it took more than 10s. On average, the addition of a template took 2,818ms and created constraints that were validated 31,936 times. Removing a template does not require validations of constraints, thus this task is performed in less than 1,600ms in 90% of our tests. Only 5% of template removals take more than 3s.

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

4.3 Applicability

In the presented examples, we have illustrated how our approach performs co-evolution of model constraints when metamodel changes occur. However, our approach is not limited to metamodels as the source of constraints. Quite the contrary, any model can be used to trigger template instantiation and the generated constraints may restrict any kind of model – even metamodels [23]. To date, various sample templates for different metamodels and models are available [11], thus we are confident that the approach is generally applicable.

Note that evolving constraints also enables repair technologies that fix detected inconsistencies (e.g., [24–26]). Therefore, our approach provides a foundation for providing guided or even automatic co-evolution of metamodels and models based on evolved constraints.

5 Related Work

There has been an extensive research activity in models and their evolution. Here we focused on those closest to our work and grouped them in three themes.
Metamodel and model (co-)evolution. The efficient, and ideally automated, (tool-)support for metamodel evolution and the corresponding co-evolution of conforming models was identified by Mens et al. in 2005 as one of the major challenges in software evolution [27]. Since then, various approaches have been proposed to deal with this challenge. Wachsmuth addresses the issue of metamodel changes by describing them as transformational adaptations that are performed stepwise instead of big, manually performed ad hoc changes [21]. Changes to the metamodel become traceable and can be qualified according to semantics- or instance-preservation. He further proposes the use of transformation patterns that are instantiated with metamodel transformations to create co-transformations for models. Cicchetti et al. classify possible metamodel changes and decompose differences between model versions into sets of changes of the same modification-class [28]. They identify possible dependencies that can occur between different kinds of modifications and provide an approach to handle these dependencies and to automate model co-evolution.

Herrmannsdoerfer et al. also classified coupled metamodel changes and investigated how far different adaptations are automatable [29]. One aspect that these approaches have in common is that they are based on decomposing evolution steps into atomic modification for deriving co-adaptations. Our approach is also based on atomic modifications that are handled individually to perform necessary adaptations incrementally. However, we do not try to automate co-evolution of metamodels and models in the first place. Instead, the co-evolution of metamodels and constraints enables tool users to perform adaptations of a model with guidance based on specific constraints and their own domain knowledge.

Wimmer et al. follow a different approach by merging two versions of a metamodel to a *unified metamodel* and then applying co-evolution rules to the models [30]. They instantiate new metaclasses and remove existing elements that are no longer needed. At first, they encountered problems regarding typecasts and instantiation so they had to change some co-evolution rules. XLM can handle the instantiation of created metaclasses as well as arbitrary typecasts of instances.

In terms of constraint co-evolution, Büttner et al. discuss various metamodel modifications and how they affected constraints [13]. They describe how OCL expressions can be transformed to reflect metamodel evolution. We encountered some of the issues they identified during the evolution of our running example,

for example the transition from single-object to collection values and vice versa because of multiplicity changes which is handled automatically in XLM.
Flexible and multilevel modeling. Atkinson and Kühne identified several issues in the field of multilevel (meta-)modeling, namely the so-called *shallow instantiation* of the UML [31] that forced us to use a graph-oriented model in XLM. They discussed different approaches to overcome these issues like the concept of *deep instantiation* where instances can be types at the same time; an approach we used in our tool's graph model. Ossher et al. lately presented the *BITKit* tool [3] that allows domain-agnostic modeling and on-the-fly assignment of visual notations to dynamically defined domain types. This approach is also implemented in our tool where the type of a model element can be changed at any time.

6 Conclusions and Future Work

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

For future work, we plan to investigate the possible benefits of using the approach not only for metamodel-dependent constraints but also for constraints that primarily rely on domain-knowledge. Moreover, we want to expand the approach so that not only constraints but also new templates can be generated through template instantiation.

7 Acknowledgments

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

References

1. D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
2. E.-J. Manders, G. Biswas, N. Mahadevan, and G. Karsai, "Component-oriented modeling of hybrid dynamic systems using the generic modeling environment," in *MBD/MOMPES*, 2006, pp. 159–168.

3. H. Ossher, R. K. E. Bellamy, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, M. Desmond, J. de Vries, A. Fisher, and S. Krasikov, "Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges," in *OOPSLA*. ACM, 2010, pp. 848–864.
4. M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "COPE - automating coupled evolution of metamodels and models," in *ECOOP*, 2009, pp. 52–76.
5. Object Management Group. Unified Modeling Language (UML). http://www.uml.org/.
6. J. Pardillo, "A systematic review on the definition of uml profiles," in *MoDELS (1)*, 2010, pp. 407–422.
7. M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider, "Flexible and scalable consistency checking on product line variability models," in *ASE*. ACM, 2010, pp. 63–72.
8. A. Reder and A. Egyed, "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *ASE*. ACM, 2010, pp. 347–348.
9. Object Management Group. Object Constraint Language (OCL). http://www.omg.org/spec/OCL/.
10. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Automatically generating and adapting model constraints to support co-evolution of design models," in *ASE*, 2012, pp. 302–305.
11. ——, "Cross-layer modeler: A tool for flexible multilevel modeling with consistency checking," in *ESEC/SIGSOFT FSE*, 2011, pp. 452–455. [Online]. Available: http://www.sea.jku.at/tools/xlm
12. Eclipse Foundation. Eclipse Modeling Framework (EMF). http://eclipse.org/modeling/emf/.
13. F. Büttner, H. Bauerdick, and M. Gogolla, "Towards transformation of integrity constraints and database states," in *DEXA Workshops*, 2005, pp. 823–828.
14. A. Egyed, "Instant consistency checking for the UML," in *ICSE*, 2006, pp. 381–390.
15. X. Blanc, A. Mougenot, I. Mounier, and T. Mens, "Incremental detection of model inconsistencies based on model operations," in *CAiSE*, 2009, pp. 32–46.
16. A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.
17. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *EDOC*, sept. 2008, pp. 222 –231.
18. K. Hassam, S. Sadou, V. L. Gloahec, and R. Fleurquin, "Assistance system for OCL constraints adaptation during metamodel evolution," in *CSMR*, 2011, pp. 151–160.
19. S. Markovic and T. Baar, "Refactoring OCL annotated UML class diagrams," in *MoDELS*, 2005, pp. 280–294.
20. G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel, "Refactoring UML models," in *UML*, 2001, pp. 134–148.
21. G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, 2007, pp. 600–624.
22. I. Groher, A. Reder, and A. Egyed, "Incremental consistency checking of dynamic constraints," in *FASE*, 2010, pp. 203–217.
23. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Constraint-driven modeling through transformation," in *ICMT*, 2012, pp. 248–263.
24. A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *ASE*, 2008, pp. 99–108.
25. A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, 2012, pp. 220–229.

26. C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE*, 2003, pp. 455–464.
27. T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *IWPSE*, 2005, pp. 13–22.
28. A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "Managing dependent changes in coupled evolution," in *ICMT*, 2009, pp. 35–51.
29. M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "Automatability of coupled evolution of metamodels and models in practice," in *MoDELS*, 2008, pp. 645–659.
30. M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel, "On using inplace transformations for model co-evolution," in *MtATL*. INRIA & Ecole des Mines de Nantes, 2010.
31. C. Atkinson and T. Kühne, "The essence of multilevel metamodeling," in *UML*. Springer, 2001, pp. 19–33.