Automatically Generating and Adapting Model Constraints to Support Co-evolution of Design Models

Andreas Demuth
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
andreas.demuth@jku.at

Roberto E. Lopez-Herrejon
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
roberto.lopez@jku.at

Alexander Egyed
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Design models must abide by constraints that can come from diverse sources, like their metamodels, requirements, or the problem domain. Software modelers expect these constraints to be enforced on their models and receive instant error feedback if they fail. This works well when constraints are stable. However, constraints may evolve much like their models do. This evolution demands efficient constraint adaptation mechanisms to ensure that models are always validated against the correct constraints. In this paper, we present an idea based on constraint templates that tackles this evolution scenario by automatically generating and updating constraints.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Languages, Reliability, Performance

Keywords

Co-evolution, metamodeling, consistency checking

1. INTRODUCTION

In *Model-Driven Development (MDD)* [1], metamodels play a key role as they define the language of models and the constraints these models must satisfy. Over the past years, a trend has emerged that calls for metamodels to be adaptable – to customize the design to a particular discipline, domain, or even application under development. Nowadays, a range of "flexible" design tools are available (e.g., [2–4]) and their metamodels are about as changeable as any other development artifact. Indeed, metamodels are allowed to evolve continuously; for example, to reflect changes in the domain, to meet new business needs or to improve the structure of the metamodel. *Co-evolution* of models denotes the process of evolving models and their metamodels concurrently – a process that is non trivial since inconsistent co-evolution may cause models and metamodels to drift apart. Several approaches that perform this process incrementally have been proposed (e.g., [5]).

However, what has been largely overlooked is the fact that metamodels also impose constraints onto models and while evolving metamodels one also ought to evolve their constraints. For example, the *Unified Modeling Language (UML)* [6] is supported by hundreds of well-formedness rules and the community augmented these with even more consistency rules. Modifying the UML metamodel thus impacts these constraints. Constraints that previously were semantically and syntactically correct can become incorrect as a consequence of structural or semantic changes of the metamodel. It is crucial to extend the notion of co-evolution to include the continous 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 its metamodel. Most consistency checkers perform the validation by presuming the existence of constraints [7, 8]. It is common to write constraints manually, often in a standardized language such as the *Object Constraint Language (OCL)* [9], or to "hard-code" them into the modeling tools. Standard consistency checkers also typically presume that the constraints remain stable throughout model evolution. 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. State-of-the-art incremental consistency checkers typically do not support the live updating of constraints.

In this paper, we discuss an approach for the co-evolution of models and their constraints that uses constraint templates and a template engine to automatically and incrementally generate and update constraints.

ASE '12, September 3–7, 2012, Essen, Germany

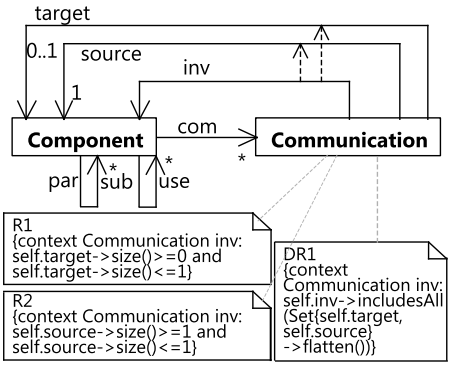

Figure 1: A metamodel for component-based systems with constraints.

In a world where little is stable, the fast and incremental co-evolution of models is becoming a fundamental best practice for software development. The main benefit of our approach in this regard is that model and metamodels can be evolved concurrently while still benefiting from fast, incremental error feedback.

2. EXAMPLE AND MOTIVATION

We use an excerpt of a simple metamodel, shown in Fig. 1, 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`).

Since producing only valid models that conform to this metamodel is crucial for using the MDD approach, constraints that stem from different sources may be added.

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. 1) 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. 1 for readability reasons. Note that common modeling tools that use *EMF* [10] for example either do not derive such constraints or have them "hard-coded", meaning that changes cannot lead to constraint updates which counters the idea of co-evolution.

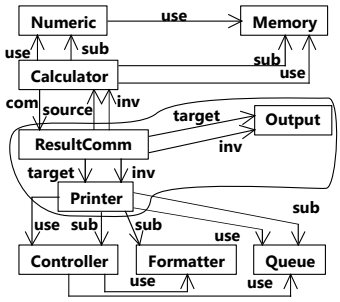

Figure 2: A model of a calculation system.

II: Metamodel semantics. Next, we create a constraint for the derived reference (e.g., $DR1$ in Fig. 1) 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 [11]. 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 hierachies. We omit a detailed description of such a constraint because of space restrictions.

As depicted in Fig. 2, the metamodel from Fig. 1 is used to create a small model of a calculator system. The `Calculator` component has two sub-components that are used directly: `Memory` and `Numeric`. The `Numeric` component also uses the component `Memory`. A `Printer` has three sub-components: `Formatter`, `Queue`, and `Controller`. It uses the `Queue` to store print jobs and informs the `Controller`, which retrieves the data from the `Queue` and runs the `Formatter` before actually 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. 2, 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 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 dotted lines in Fig. 3.

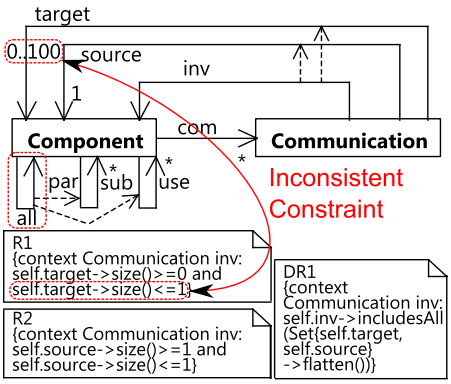

Figure 3: The evolved metamodel.

These changes have the following consequences:

- Constraint $R1$ becomes incorrect. The upper bound checked by the constraint, value (`1`), is no longer equal to the actual upper bound value of the reference (`100`).
- An additional constraint is now needed for the newly created 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. 2, 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 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.

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 and the information it requires. The *instantiation context (IC)* defines for which elements, or combinations thereof, a template should be instantiated. The *abstract constraint expression (ACE)* is used to define the *family of constraints* generated from the template. A constraint family consists of constraints that share some static aspects (e.g., the structure) and have some variable parts that differ for each constraint. Thus, the ACE captures the static parts of the constraint family and also identifies the locations of variability which are also defined explicitly in the *variable definition (VD)*. The VD declares which parts of the ACE are interpreted as variables. To bind specific values to these variables, data has to be read from specific elements that are available when the template is instantiated. These elements are specified in the *instantiation information (II)*. How the values for the variables are extracted from the elements is declared in *data extraction expressions (DEE)*.

Let us now show how we can write a template $T1$ for the constraint family of $R1$ and $R2$. Template $T1$, shown in Table 1, creates a constraint every time an instance of `Reference` is connected to an instance of `Class`, 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 `<Class, Reference>`. This means that we provide an instance of `Class` and 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 constraint $R1$ and replace the two values 0 and 1 with `MIN` and `MAX`, the context `Communication` with `C`, and the two occurrences of `target` with `R`. The result is the abstract constraint expression as defined in Table 1 with the variable parts (VD) being `<C, R, MIN, MAX>`. The instantiation information of $T1$ is `<Class c, Reference r>`. The data extraction expressions can be written as `r.min`, `r.max`, `r.name` and `c.name`. Note that – for convenience – we use the n-th expression in the DEEs to extract the value for the n-th variable in the VD in the paper. However, the connection between expressions in the DEE and the corresponding variables in the VD could also be defined explicitly.

3.2 Template Instantiation and Change Management

For performing automatic instantiation of templates (i.e., generation of constraints) as well as automatic updates of already instantiated ones, we propose the use of a template engine. This engine observes the metamodel for changes. Such changes may be the addition or removal of metamodel elements as well as metamodel element modifications. For the addition, the engine looks for templates with an instantiation context that matches the added element. For each match, the engine instantiates the template by executing

Table 1: Definition of template T1

IC:	`<Class, Reference>`

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

VD:	`<C, R, MIN, MAX>`
II:	`<Class c, Reference r>`
DEE:	`<c.name, r.name, r.min, r.max>`

the data extraction expressions (DEEs) and replacing the defined variables with the retrieved data, creating a constraint that is then passed to the consistency checker. During execution of the DEEs, the engine captures which metamodel elements were accessed, building the scope of the generated constraint. If the engine observes a metamodel element modification, it looks for generated constraints whose scope includes the changed metamodel element. These constraints are potentially outdated and require updating. The engine then re-instantiates the template by re-executing the DEEs that retrieve the updated values, leading to an updated constraint that replaces the outdated one. The removal of metamodel elements indeed leads to the removal of constraints that rely on the removed data.

4. RELATED WORK

There has been an extensive research activity in models and their evolution. Here we focused on those closest to our work. 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 [12]. 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 [13]. 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 [14]. 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 [15]. 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 both tool guidance based on specific constraints and their own domain knowledge.

In [16], Kim and Czarnecki deal with the evolution of *feature models* and their *specializations* in software product lines. One of the feature model evolutions they perform is the restriction of cardinalities in the feature model. With our approach, such changes are handled automatically and specializations that are inconsistent with the evolved feature model are detected immediately.

5. CONCLUSIONS

This paper introduced the idea of constraint templates and an automated template engine to address the issue of co-evolution of models and constraints. We illustrated how constraint templates can be written and how constraints can be generated from them. Moreover, we discussed how automatic co-evolution of constraints can be achieved.

6. 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.

7. 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] J. C. Grundy, J. G. Hosking, J. Huh, and K. N.-L. Li, "Marama: an eclipse meta-toolset for generating multi-view environments," in *ICSE*, 2008, pp. 819–822.

[4] 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.

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

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

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

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

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

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

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

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

[16] C. H. P. Kim and K. Czarnecki, "Synchronizing cardinality-based feature models and their specializations," in *ECMDA-FA*, 2005, pp. 331–348.