Cross-Layer Modeler – A Tool for Flexible Multilevel Modeling with Consistency Checking

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

Model-driven engineering has become a popular methodology in software engineering. Most available modeling tools support the creation of models based on a fixed metamodel. Typically, tool users cannot change the metamodel to reflect domain changes or newly emerged requirements. As a consequence, an updated version of the tool with an evolved metamodel must be developed and models as well as constraints that ensure model consistency have to be co-evolved, often manually, to conform to the new metamodel. Both, tool evolution and the necessary co-evolutions, are time consuming and error prone tasks. Furthermore, common tools often restrict the number of metalevels that can be modeled and force modelers to use workarounds to express certain facts. To overcome these issues we present the *Cross-Layer Modeler (XLM)*, a modeling tool that supports multilevel modeling and allows co-evolution of metamodels and models. The XLM automatically performs co-evolution of constraints and gives instant feedback about model consistency. We illustrate the novel modeling approach of our tool and discuss its main capabilities.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Design, Experimentation

Keywords

Consistency checking, metamodeling, multilevel modeling

1. INTRODUCTION

Model-driven engineering (MDE) has become a popular practice to reduce development effort and to improve product quality by automatically generating artifacts from models [8]. Even though most common modeling tools use sophisticated metamodels (e.g., the *Unified Modeling Language (UML)* [7]) that allow the modeler to describe various aspects of systems in different notations, there can still arise the need for special language constructs that are not provided by the metamodel. However, modeling tools typically do not support metamodel modifications because they internally use a fixed implementation of the metamodel and rely on an immutable set of classes and their properties.

Metamodeling tools can be used to easily create domain-specific modeling tools with a metamodel that reflects a single domain. Although this simplifies updating the metamodel, there are several issues that arise with the updated version of the modeling tool. Namely, old models are likely to no longer conform to the new metamodel and require co-evolution.

Changes of metamodels do not only require tools and existing models to be updated, they can also mean that existing constraints can become syntactically incorrect or that they are checking wrong properties because of modified semantics.

In this paper, we present the *Cross-Layer Modeler (XLM)* that addresses these issues by allowing the user to modify metamodels and models at the same time. Our tool provides consistency checking and automatically updates constraints to keep them valid after metamodel changes.

In the next section, we illustrate the problems that can arise with the use of conventional modeling tools. Then, we show how the use of our tool can avoid these problems in Section 3. We give a detailed description of how our tool can be used to model an arbitrary number of metalevels at the same time, explain how we add constraints and discuss different aspects of the tool's architecture.

2. MOTIVATION

As a motivating example, we use a simple feature model of a car product line inspired by [2].

Fig. 1(a) depicts the metamodel of the used feature model. There are `Feature` elements that can be linked by `Association` elements. Every `Association` connects a `source` feature to at least one `target` feature. Features can be the source of an arbitrary number of associations. There are different kinds of associations: `Xor`, `Or`, `Mandatory`, and `Optional`. `Xor` and `Or` associations are respectively used when exactly one or at least one of several `target` sub-features

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09 ...$10.00.

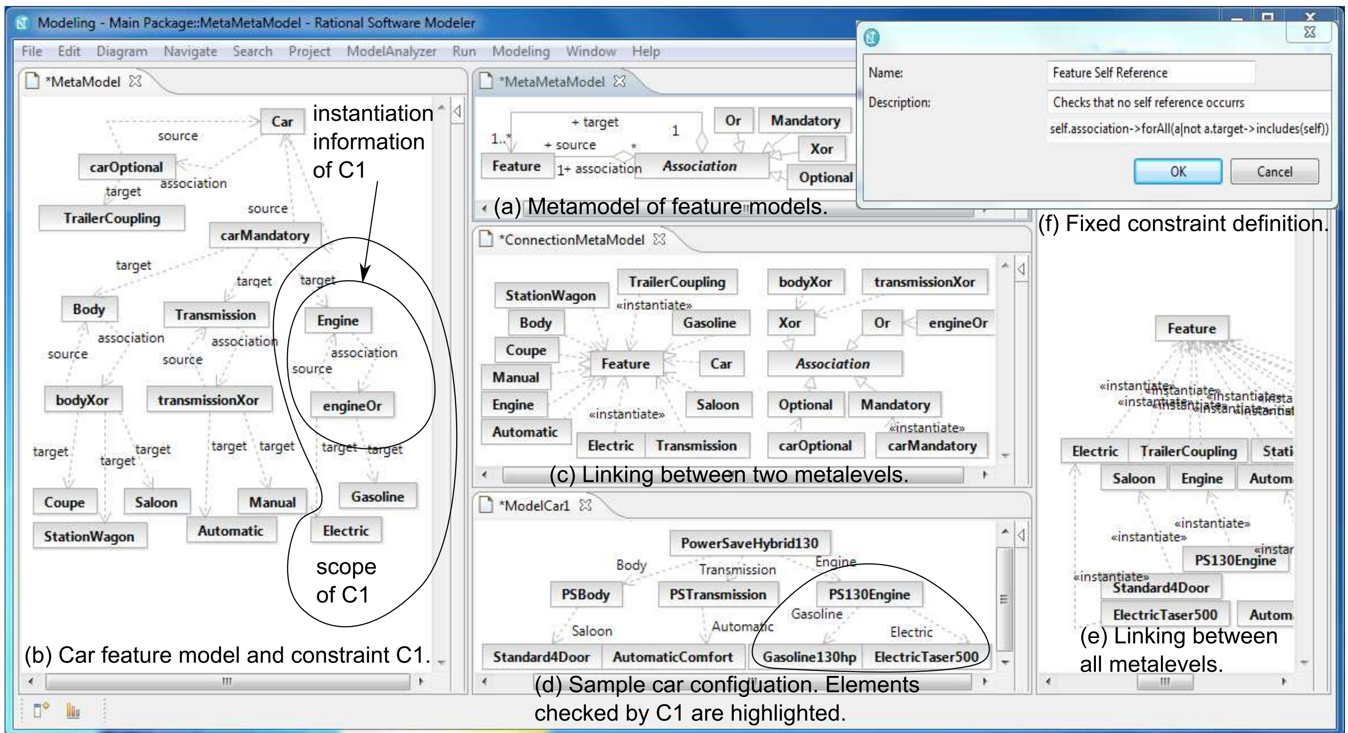

Figure 1: Screenshot of the Cross-Layer Modeler.

must be selected. The `target` of `Mandatory` associations must be selected and `target` features of `Optional` associations can be selected.

Based on this metamodel, we create the feature model of our car product line as depicted by the diagram in Fig. 1(b). A `Car` needs a `Body`, a `Transmission`, an `Engine` and can have a `TrailerCoupling`. The `Transmission` can be either `Automatic` or `Manual`. The `Engine` can be `Electric` or `Gasoline` powered, hybrid models can have both. One of three different `Body` types must be used: `Convertible`, `Saloon`, or `Coupe`. Note that for readability reasons we displayed only references from `Association` elements to features and omitted reference names.

Now that we have created a metamodel and a feature model for our small product line, we want to model a product configuration that consists of concrete realizations of the defined features. At this point, we face a challenge: modeling more than two metalevels (e.g., the metamodel and the feature model).

When we create an instance of a metamodel element (e.g., `Feature`) to define a feature (e.g., `Body`), the actual class of the created objects is the UML type `InstanceSpecification`, which means that we cannot create instances of `Body`, for example to describe a concrete feature realization.

The typical workaround is to define a class `FeatureRealization` and create a reference to the realized `Feature`. However, this approach leads to a counterintuitive situation where features and their realizations are situated at the same metalevel, namely they are all defined and also instantiated at the `M1` (User Model) level directly underneath the `M2` (UML) level (as discussed in [1]). Next, we explain how we addressed this issue.

3. CROSS-LAYER MODELER

The Cross-Layer Modeler allows multilevel modeling of arbitrary numbers of modeling layers. Every model element, regardless of its metalevel, can be instantiated to create an element at the next metalevel underneath. It should be noted that our tool does not impose any restrictions in terms of instantiations or connections between model elements at different levels. In contrast to the UML where different elements have different meanings and properties (e.g., `Usage` and `Generalization` connections), in XLM model elements and connections do not have a meaning unless it is assigned by the addition of type information (e.g., `Body` is an instance of `Feature`) or constraints (e.g., every `Feature` instance can have an arbitrary number of `Association` instances assigned to it).

Most parts of modeling, including instantiations, are performed visually. The user can decide whether the modeling of various aspects of the system is done in a single or in multiple diagrams, the latter allows a clear visual separation of metalevels or other concerns. Instances of selected model elements as well as all of its attributes are displayed in a separate view to provide a quick overview even when the information is not explicitly shown in the diagram. We will illustrate the modeling by extending our sample product line by a product configuration next.

3.1 Building models

As it can be seen in Fig. 1(b), we use UML `Class` elements instead of `InstanceSpecification` elements to define various features such as `Body` or `Engine`. Since we want to keep model and metamodel in different diagrams to achieve a clear visual separation, we use a separate diagram for instantiations between the two levels. Fig. 1(c) depicts how

we use UML `Usage` elements (dashed connections) with an `instantiate` stereotype to model instantiations of features (left side) and associations (right side) (most stereotype labels are omitted in the figure for readability reasons).

Now we create a concrete configuration that is assembled from several artifacts that implement different features, this configuration is shown in Fig. 1(d). The `PowerSaveHybrid130` has a body (`PSBody`), a transmission (`PSTransmission`) and an engine (`PS130Engine`) designed especially for the PowerSave series. The body is a `Standard4Door` version, the `PSTransmission` is based on the `AutomaticComfort` transmission and the `PS130Engine` is a combination of the `Gasoline130hp` and the `ElectricTaser500` engines. In this configuration, the car does not provide a trailer coupling. In the same way we did it in Fig. 1(c), we can use a separate diagram to define that the used model elements are realizations of features. We use the same kind of connection to indicate that a model element follows the concepts defined by another element, like instantiations of classes or realizations of features, as shown in Fig. 1(e).

For references between model elements we use UML `Dependency` connections. The source element of the connection owns a reference with the connection's name to the target. For example, `PS130Engine.Gasoline` refers to `Gasoline130hp`.

Note that the modeling we have done so far, except for the instantiations, does not add any semantics to model elements and that there are no rules that restrict the addition of connections between arbitrary elements. Next, we describe how we add semantics and restrict the freedom of modeling to only allow correct models.

3.2 Adding constraints

In our tool, there are two ways of adding constraints to models: i) add a *constraint template* or ii) add a *fixed constraint*. Constraint templates are automatically instantiated to create constraints that check properties that are not known in advance. For example, we can define a template `T1` to ensure that for every `Or` association and its source `Feature` we check that realizations of the `Feature` have references to realizations of the association's `target` features. This template generates a specific constraint for every `Or` association and its `source` in the model.

A template consists of the *instantiation context*, the types of the elements that are needed for an instantiation (e.g., `Feature` and `Or` in case of `T1`). We specify the desired *abstract constraint* with a rule language statement (e.g., in OCL [9]), parts of the constraint that are different for every instantiation of the template are expressed with placeholders. For example, we define the abstract constraint of `T1` as:

```
context FEATURE inv:  not self.allReferences->
excludesAll(TARGETS).
```

Then, we define how concrete values for the placeholders are retrieved. We can use the elements required for instantiation - called the *instantiation information* - to retrieve the values `Feature.name` for `FEATURE` and `Or.target->collect(name)` for `TARGETS`. The resulting *concrete constraint* is:

```
context Engine inv:  not self.allReferences->
excludesAll({'Gasoline', 'Electric'}).
```

Fig. 1(b) shows the instantiation information (`Engine` and `engineOr`) and the elements that are additionally accessed (`Gasoline` and `Electric`) when `T1` is instantiated to create `C1`. As it can be seen highlighted in Fig. 1(d), our sample configuration is consistent with `C1` because `PS130Engine` has a reference named `Gasoline` to `Gasoline130hp` and a reference named `Electric` to `ElectricTaser500`.

Fixed constraints are used to check properties that are equal for all instances of a type, for example a constraint could check that every `FeatureModel` does not contain two different features with the same name.

Our tool provides several wizards for the definition of templates and fixed constraints, as seen in Fig. 1(f). There are also various views that inform the user about constraints defined on selected model elements, all defined constraints, model inconsistencies and more. Because we perform the consistency checking incrementally, feedback about model consistency is provided instantly after every model change.

In addition to the creation of different metalevels at the same time and consistency checking, another key aspect of our tool is the flexibility in terms of model evolution.

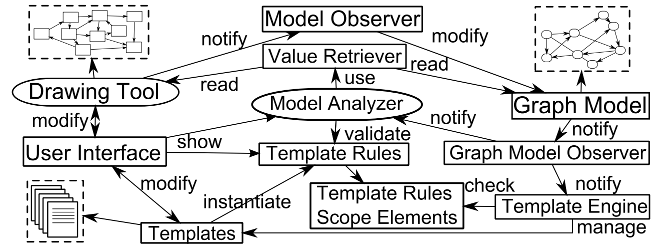

Figure 2: Architecture of Cross-Layer Modeler.

3.3 Evolving models

The XLM does not only allow the user to perform typical modifications like adding and removing elements from the model, but it also supports changes of the types of model elements at runtime. As an example, lets assume that because of new laws our `Standard4Door` feature realization is now legally treated as a station wagon. We can easily adapt to this new situation by changing the type from `Saloon` to `StationWagon` without the need of creating a new element or performing any other changes (note that in standard modeling tools, changing the type of an instance is typically not possible). However, changing the type means that our sample configuration in Fig. 1(d) is no longer valid because `PSBody` must refer to `Standard4Door` via `StationWagon` now.

Constraints derived from constraint templates are automatically updated if model elements that provided specific information for the constraint are modified. For example, changing the name of the element `Gasoline` to `Gas` in Fig. 1(b) triggers an update of the constraint `C1` because the modified element is part of the constraint's *scope*, the set of elements that are accessed during template instantiation (illustrated by the highlighted area in Fig. 1(b)). The updated constraint then requires a reference named `Gas` instead of `Gasoline`.

Now that we have presented the most important features of the XLM, we will have a look at its architecture in the next section.

3.4 Tool Architecture

Fig. 2 illustrates the overall architecture of the Cross-Layer Modeler. Encircled we indicate the connections of

our tool with other existing components, dashed boxes indicate artifacts (e.g., models and template files). We employed the *IBM Rational Software Modeler (RSM)* as a front-end drawing tool. We addressed the issues previously highlighted in Section 2 by using a graph-oriented model (`GraphModel`) that consists of `Node` and `Association` elements. Every model element, independent from the type it models or the metalevel it resides on, is an instance of `Node`. Instantiation of modeled types is emulated by a reference `type` between two nodes. This means that we can treat every element at every metalevel equally without having to deal with different classes and their properties. The graph model is kept up to date by the `Model Observer`, which is notified about changes of the drawing tool and performs the required steps to reflect these changes in the graph model.

The templates that describe the desired behavior of models are managed by the `TemplateEngine` component. The engine handles the creation of `Template Rules` (i.e., rules that were created through template instantiation) and is also responsible for updates of outdated rules. Rules are considered as outdated by the template engine when parts of their scopes (i.e., the model elements that were used during instantiation to derive concrete values to replace the variables in the abstract specification of a template) are changed.

The `Model Analyzer` [5] component is responsible for validating rules and checking the model's consistency. Note that the component also handles rules that were not derived from templates but directly added by the user, which is omitted in Fig. 2. We use a `Value Retriever` component to extract values required during rule validation.

The `Graph Model` informs the `Graph Model Observer` about changes. This observer generates change notifications and notifies both the `Template Engine` and the `Model Analyzer` so that they can start a re-validation of affected rules or trigger the creation or an update of `Template Rules`.

3.5 Preliminary Evaluation

We performed several tests with our tool and compared its consistency checking performance to the reference implementation of the *Model Analyzer* [5] consistency checker. The results show that the higher complexity of our implementation – based on the template mechanism and more complex type handling – does not impose noticeable performance drawbacks as feedback about model consistency is provided within milliseconds after a model change occurred. For the future, we plan to do extensive analysis of this impact with large-scale industrial models.

4. RELATED WORK

Even though there are various approaches that address the issue of model and metamodel co-evolution (e.g., *Epsilon Flock* by Rose et al. [6]), less work has focused on the co-evolution of (meta)models and constraints.

In terms of flexible modeling, the *Generic Modeling Environment (GME)* is a well known configurable environment that allows the modeler to specify metamodels and generate domain-specific modeling environments [3]. They use a sophisticated, yet fixed, set of concepts to describe these metamodels, therefore there are still possible restrictions.

Another tool that focuses on flexibility is the *Business Insight Toolkit (BITKit)* developed by IBM [4]. It allows the user to create diagrams and add semantics to graphical elements afterwards. This approach is also used in our tool where we can create elements and change their type anytime, although we do not provide such a visual experience yet.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have shown that the Cross-Layer Modeler supports the modeling of multiple metalevels and gives the user the freedom to change types of model elements as well as semantics at any time. The simple notation and the clear design principles enable even novice users to work with our tool. Currently our tool is in an early prototype stage, nevertheless it provides full functionality in terms of modeling multiple metalevels and checking the conformance of models to their metamodels.

For future releases we plan to enrich the visual experience during modeling and to add more flexibility and guidance regarding the definition of constraints, for example dynamic code-completion during constraint or template authoring.

Even though we focused on a small product line example, we believe that our tool is suitable for modeling various domains where it is necessary to use multiple metalevels or where aspects of the domain are likely to change.

6. ACKNOWLEDGMENTS

We would like to gratefully acknowledge the Austrian Science Fund (FWF) through grant P21321-N15 and the EU Marie Curie Actions – Intra European Fellowship (IEF) through project number 254965.

7. REFERENCES

[1] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *UML*, pages 19–33. Springer, 2001.

[2] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming. In *ESEC / SIGSOFT FSE*, pages 2–19. Springer, 1999.

[3] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.

[4] Harold Ossher, Rachel K. E. Bellamy, Ian Simmonds, David Amid, Ateret Anaby-Tavor, Matthew Callery, Michael Desmond, Jacqueline de Vries, Amit Fisher, and Sophia Krasikov. Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges. In *OOPSLA*, pages 848–864. ACM, 2010.

[5] Alexander Reder and Alexander Egyed. Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML. In *ASE*, pages 347–348. ACM, 2010.

[6] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In *ICMT*, pages 184–198. Springer, 2010.

[7] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual (2nd Edition)*. Pearson Higher Education, 2004.

[8] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[9] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman, Inc., 1999.