Approaches to Identify Object Correspondences Between Source Models and Their View Models

Csaba Debreceni[1,2], Dániel Varró[1,2,3]
[1]Budapest University of Technologies and Economics, Department of Measurement and Information Systems, Hungary
[2]MTA-BME Lendület Research Group on Cyber-Physical Systems, Hungary
[3]McGill University of Montreal, Department of Electrical and Computer Engineering, Canada
Email: {debreceni,varro}@mit.bme.hu

***Abstract*—Model-based collaborative development of embedded, complex and safety critical systems has increased in the last few years. Several subcontractors, vendors and development teams integrate their models and components to develop complex systems. Thus, the protection of confidentiality and integrity of design artifacts is required.**

In practice, each collaborator obtains a filtered local copy of the source model (called view model) containing only those model elements which they are allowed to read. Write access control policies are checked upon submitting model changes back to the source model. In this context, it is a crucial task to properly identify that which element in the view model is associated to which element in the source model.

In this paper, we overview the approaches to identify correspondences between objects in the filtered views and source models. We collect pros and cons against each approach. Finally, we illustrate the approaches on a case-study extracted from the MONDO EU project.

I. Introduction

Model-based systems engineering has become an increasingly popular approach [1] followed by many system integrators like airframers or car manufacturers to simultaneously enhance quality and productivity. An emerging industrial practice of system integrators is to outsource the development of various components to subcontractors in an architecture-driven supply chain. Collaboration between distributed teams of different stakeholders (system integrators, software engineers of component providers/suppliers, hardware engineers, specialists, certification authorities, etc.) is intensified via the use of models.

In an *offline collaboration* scenario, collaborators check out an artifact from a version control system (VCS) and commit local changes to the repository in an asynchronous long transaction. Several collaborative modeling frameworks exist (CDO [2], EMFStore [3]), but security management is unfortunately still in a preliminary phase. Traditional VCSs (Git [4], Subversion [5]) try to address secure access control by splitting the system model into multiple fragments, but it results in inflexible model fragmentation which becomes a scalability and usability bottleneck (e.g. over 1000 model fragments for automotive models).

This paper is partially supported by the EU Commission with project MONDO (FP7-ICT-2013-10), no. 611125. and the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems.

In our previous works [6], we introduced a novel approach to define fine-grained access control policies for models using graph queries. A bidirectional graph transformation so called *lens* is responsible for access control management. The forward transformation derives consistent *view models* by eliminating undesirable elements from the *source model* according to the access control rules that restrict the visibility of the objects, references and attributes. In contrast, the backward transformation propagate the changes executed on the view models back to the source model. It also enforces the write permission defined in the policies by rejecting all the changes if any of them violates an access control rule.

At commit time, the executed operations and their orders are not available in the most cases, only the deltas are sent to the VCS. It is an urgent task to correctly identify the modified elements of the model to recognize whether an access control rule is violated. Thus object correspondences need to be built between the element of the source and the modified views.

In this paper, first we motivate the need of identification of object correspondences using a case study from MONDO EU FP7 project. Then we overview the possible approaches and discuss their advantages and disadvantages that can be applied onto our existing lens-based approach.

II. Preliminaries

A. Instance Models and Modeling Languages

A metamodel describes the abstract syntax of a modeling language. It can be represented by a type graph. Nodes of the type graph are called classes. A class may have attributes that define some kind of properties of the specific class. Associations define references between classes. Attributes and references altogether are called features.

The instance model (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called objects and links, respectively. Objects and links are the instances of modeling language level classes and associations, respectively. Attributes in the metamodel appear as slots in the instance model.

B. Enforce Access Control Policies by Graph Transformation

In the literature of bidirectional transformations [7], a lens (or view-update) is defined as an asymmetric bidirectional transformations relationship where a source knowledge base completely determines a derived (view) knowledge base, while the latter may not contain all information contained in the former, but can still be updated directly.

The kind of the relationship we find between a source model (containing all facts) and a view model (containing a filtered view) fits the definition of a lens. After executing the transformation rules, model objects of the two models reside at different memory addresses, so the transformation must set up a one-to-one mapping called *object correspondence*, that can be used to translate model facts when propagating changes.

We assume that the forward transformation of the lens builds correspondences between objects of source and view models. But these correspondence relation cannot be guaranteed when the derived view model is reloaded as a new model because the new objects will not share the same memory addresses.

Rebuilding correspondence mapping between the source model and the modified view model is cumbersome, where the view may hide most of the sensitive information. Instead, correspondences are easier to build between the unmodified and modified view model as it is depicted in Fig. 1, then the originally achieved mapping can be used.

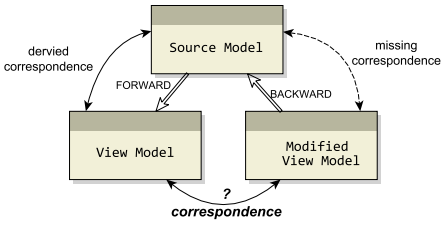

Fig. 1. Request to build correspondence between view models

C. Model Comparison

Building correspondences between two version of the same model is common problem in model versioning called *model comparison*. Model comparison process is responsible for identifying differences of two model and translate them into elementary model operations such as *create*, *update* and *delete*. A common issue in this context is to recognize whether an object is moved to another place or an existing object is deleted and a completely new one is created in the model with the same attribute values. Thus it is required to build correspondences between the two model to properly identify the differences.

III. Motivating Example

Several concepts will be illustrated using a simplified version of a modeling language (metamodel) for system integrators of offshore wind turbine controllers, which is one of the case studies [8] of the MONDO EU FP7 project. The metamodel, depicted by Fig. 2, describes how the system is modeled as modules providing and consuming signals. Modules are organized in a containment hierarchy of composite modules, ultimately containing control unit modules responsible for a given type of physical device (such as pumps, heaters or fans). Composite modules may be shipped by external vendors and may express protected intellectual property (IP).

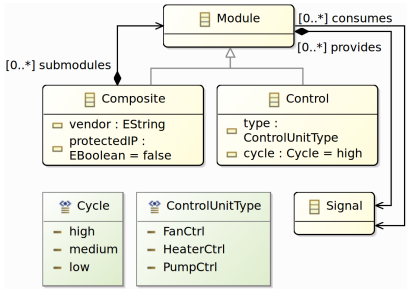

Fig. 2. Simplified wind turbine metamodel

A sample instance model containing a hierarchy of 2 `Composite` modules and a `Control` units, providing a `Signal` altogether, is shown on the top left side of Fig. 3 called *source model*. Boxes represent objects (with attribute values as entries within the box), while arrows represent containment edges and cross-references.

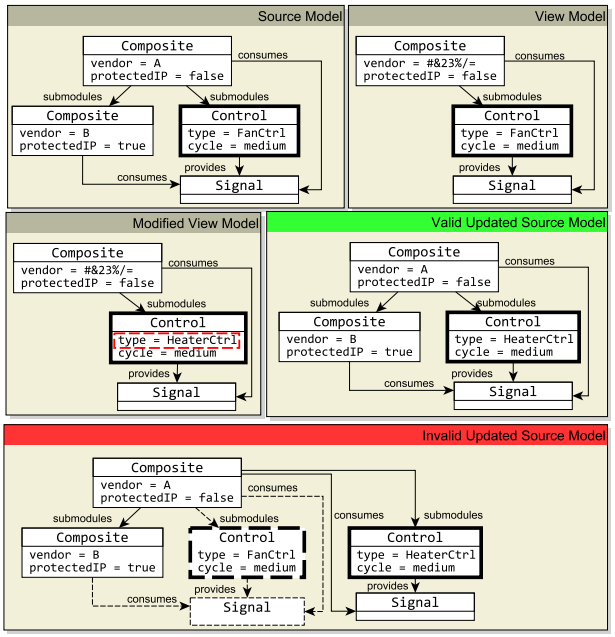

Fig. 3. Example instance model

Access Control. Specialists are engineers responsible for maintaining the model of control unit modules and have specific access to the models, described in the following:

R1. Intellectual properties have to be hidden from specialists.
R2. Objects cannot be created or deleted in the system model.
R3. Vendor attribute of visible composites must be obfuscated.

R4. Control units and their attributes can be modified.

According to the aforementioned access control rules, view models depicted in the top middle of Fig. 3 are derived for specialists where the protected IP objects are not visible and the vendor attributes are obfuscated. Only the control unit (marked with bold border) is allowed to be modified by specialists.

Scenario. At a given point, a specialist changes the type of the control unit from `FanCtrl` to `HeaterCtrl` represented on the top right side of Fig. 3 and propagate the modifications from view model back to the source model. It is need to be decided whether the change was allowed or not. Two cases can arise: (i) the VCS realizes that only the type attribute was modified; or (ii) the VCS interprets the change as the deletion of the original control unit and an addition of a new control unit. The former case will be accepted (*valid updated source model* on Fig. 3) while the latter one need to be rejected ((*invalid updated source model* on Fig. 3)) as it removes the `control` unit and its `signal` with the related references (marked with dashed borders and edges). Thus, the VCS has to identify which object has changed to be able to make a proper decision.

IV. Overview of the Approaches

In this section, we categorize the possible approaches to based on comparison techniques collected in [9]. For each approach we discuss its advantages and disadvantages and provide their application onto our running example.

A. Static Identifiers

Several modeling environments automatically provide unique identifiers for each object. The requirements against the identifiers are the following:

SI1. Identifiers need to assign to all objects.
SI2. Recycling of identifiers are not allowed.
SI3. Identifiers cannot be changed after serialization.
SI4. After deserialization, the identifiers need to remain.

For instance, the Industry Foundation Classes [10] (IFC) standard, intended to describe building and construction industry data, assigns unique number at element creation time. At the beginning it assigns 0 for the first object and then it increases the previous assigned identifier with 1. In case of the Eclipse Modeling Framework [11] (EMF), unique identifiers are assigned at serialization time if the serialization format supports this features (e.g. *XMI* format supports, but *Binary* not). In practical, a universally unique identifier (UUID) is generated for each object that still does not have any.

Advantages. Static identifiers require no user specific configuration. Always provides a perfect match for correspondences.

Disadvantages. Modeling environments or serialization format need to be changed. Moreover, it is possible, that the modeling tools do not support these formats.

Example. For our running example, static identifiers can be achieved using a proper serialization format that provide unique identifiers.

B. Custom Identifiers

In practice, domain language developers usually prepare their languages to support identifiers by adding a common ancestor for all classes that provides an identifier attribute. During the development phase, engineers need to manually set the identifiers for each object where the uniqueness cannot be guaranteed. Moreover, access control rules must make that attribute visible (at least in an obfuscated form) in views.

Advantages. There is no need to change modeling environment or model serialization.

Disadvantages. Existing languages need to be modified which may lead to inconsistencies. Uniqueness is questionable.

Example. Fig. 4 shows a possible extension of the aforementioned metamodel with a `NamedElement` interface. All the classes inherit the `id` attributes.

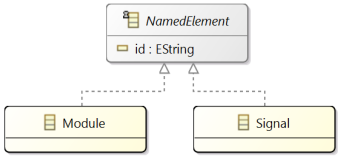

Fig. 4. Identifier introduced in a common ancestor

C. Signature-based Matching

Signature-based matching does not rely on unique identifiers, instead it calculates signatures of the objects. A signature is a user-defined function described as a model query. This approach is introduced in [12], whereas Hegedus et al. [13] described a similar approach so called *soft traceability links* between models. All the references and attributes involved in the calculation of a signature need to be visible (at least in and obfuscated form).

Advantages. There is no need to change modeling environment or serialization format, thus the modeling tools will still support the models.

Disadvantages. Users need to specify how to calculate the signature, which might lead to several false positive results.

Example. For our running example, a simple signature query defined in ViatraQuery language [14] is represented in Listing 1, which makes two control units identically equal if their `container` modules, the value of their `cycle` attributes and their provided `signal` objects are identically equals. This query successfully identifies changes introduced in the running example. However, it cannot recognize the deletion and addition of two different control units at the same position.

```
1 pattern sign(ctrl,cycle,sig,container) {
2   Control.eContainer(ctrl,container);
3   Control.cycle(ctrl,cycle);
4   Control.provides(ctrl,sig);
5 }
```

Listing 1. Example Signature Query

D. Similarity-based Matching

Similarity-based matching tries to measure the similarity between objects based on the *similarity value*. In contrast,

identifiers and signatures directly decide whether a correspondence exists between two objects. Similarity is calculated by the values of each features. For each feature, users need to specify a weight that define how important is it in the identification. Using these weights, meta-model independent algorithms derive the correspondences between the objects.

For instance, EMF Compare [15] is comparison tools to compare EMF models, and use similarity based-matching. Its calculation includes analyzing the name, content, type, and relations of the elements, but it also filters out element data that comes from default values etc.

Advantages. The identification is based on general heuristics and algorithms, where the users do not need to provide complex description on how to identify an object.

Disadvantages. Users need to specify weight for the features to fine-tune the similarity algorithms.

Example. A possible list of weights is defined in Listing 2, where the references of the aforementioned metamodel have more influence on the similarity than the attributes. In this case, our example modifications will be successfully recognized. However, Listing 3 describes a context, where the attributes are more important than the others. Thus, if we change the value of an attribute, it will be recognized as a deletion of an object and the creation of a new one.

```
1 wieghts
2 * container: 2
3 * provides:  2
4 * type: 0
5 * cycle: 0
6 * vendor: 0
7 * consumes: 0
8 * submodules: 0
9 * protectedIP: 1
```

Listing 2.
Weights with environment pressure

```
1 wieghts
2 * container: 0
3 * provides:  0
4 * type: 5
5 * cycle: 2
6 * vendor: 2
7 * consumes: 0
8 * submodules: 0
9 * protectedIP: 1
```

Listing 3.
Weights with attribute pressure

E. Language-specific Algorithms

Language-specific algorithms are designed to a given modeling language. Thus these approaches can take the semantics of the languages into account to provide more accurate identification of objects. For instance, a UML-specific algorithm can use the fact that two classes with the same name mean a match and it does not matter where they were moved in the model. UmlDiff [16] tool uses similar approach for differencing UML models. To ease the development of such a matching algorithms, the Epsilon Comparison Language (ECL) [17] can automate the trivial parts of the process, where developers only need to concentrate on the logical part.

Advantages. Semantics of the language are used and there is no need to any modification in the model or modeling tools.

Disadvantages. Users need to specify a complete matching algorithm for a given language which can be challenging.

Example. A simple matching rule defined with ECL is presented in Listing 4. It matches a `s` control unit with Fig.t control unit (declared in `match-with` part) if their container and the provided signals are equal (declared in `compare` part).

```
1 rule MatchControls
2   match s : Control
3   with t : Control {
4     compare {
5       return s.container = t.container
6       and s.provides = t.provides
7     }
8 }
```

Listing 4. Example Rule in Epsilon Comparison Language

V. Conclusion and Future Work

In this paper, we aimed to overview the approaches to identify correspondences between an original model and its filtered and modified version. We categorized these approaches into 5 groups - using *static identifiers* or *custom identifiers*, calculating *signature-based matches*, aggregating values of features using *similarity-based matching* and providing *language specific algorithms*. We introduced their application on a case study extracted from MONDO EU project and discussed their pros and cons.

As future work, we plan to integrate these approaches into our query-based access control approach [6] and evaluate them from the aspects of usability and scalability.

References

[1] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79 – 85, 2014.

[2] The Eclipse Foundation, "CDO," http://www.eclipse.org/cdo.

[3] ——, "EMFStore," http://www.eclipse.org/emfstore.

[4] Git, "Git," https://git-scm.com/.

[5] Apache, "Subversion," https://subversion.apache.org/.

[6] G. Bergmann, C. Debreceni, I. Ráth, and D. Varró, "Query-based Access Control for Secure Collaborative Modeling using Bidirectional Transformations," in *ACM/IEEE 19th Int. Conf. on MODELS*, 2016.

[7] Z. Diskin, "Algebraic models for bidirectional model synchronization," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 21–36.

[8] A. Bagnato, E. Brosse, A. Sadovykh, P. Maló, S. Trujillo, X. Mendialdua, and X. De Carlos, "Flexible and scalable modelling in the mondo project: Industrial case studies." in *XM@ MoDELS*, 2014, pp. 42–51.

[9] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. IEEE Computer Society, 2009, pp. 1–6.

[10] M. Laakso and A. Kiviniemi, "The IFC standard: A review of history, development, and standardization, information technology," *ITcon*, vol. 17, no. 9, pp. 134–161, 2012.

[11] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[12] R. Reddy, R. France, S. Ghosh, F. Fleurey, and B. Baudry, "Model composition-a signature-based approach," in *Aspect Oriented Modeling (AOM) Workshop*, 2005.

[13] Á. Hegedüs, Á. Horváth, I. Ráth, R. R. Starr, and D. Varró, "Query-driven soft traceability links for models," *Software & Systems Modeling*, pp. 1–24, 2014.

[14] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for emf models," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2011, pp. 167–182.

[15] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *UPGRADE, The European Journal for the Informatics Professional*, vol. 9, no. 2, pp. 29–34, 2008.

[16] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM Int. Conf. on Automated Soft. Eng.* ACM, 2005, pp. 54–65.

[17] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Model comparison: a foundation for model composition and model transformation testing," in *Proceedings of the 2006 international workshop on Global integrated model management*. ACM, 2006, pp. 13–20.