Locally unique labeling of model elements for state-based model differences

Locally unique labeling of model elements for state-based model differences

Zvezdan Protić
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Z.Protic@tue.nl

August 2, 2010

Abstract

Model versioning occupies a central position in model configuration management systems, which are considered integral ingredients in every large model driven engineering project. There are two major approaches in describing versions of models: the state-based approach and the change (operation) based approach. In both cases, the differences between two different versions of a model are called model differences. We focus on the state-based approach to versioning and show that in this case the model differences require a locally unique labeling for all model elements. Next, we specify a generic method for assigning unique labels to all model elements in versioned models. We also show that the described labeling method is also applicable to ambiguous models, i.e. models which contain structurally identical elements having the same parent element.

The main application area of the presented method is the process of calculating model differences in case that modeling tools do not assign unique identifiers to all model elements.

1 Introduction

Model versioning occupies a central position in model configuration management systems. Two major approaches in describing versions of models are the state based and the change (operation) based approach [5]. In a change (operation) based approach, the version (of a model) is defined as a set of changes (operations) which, when applied to some initial (baseline) model (possibly empty), produces the required model. On the other hand, in a state-based approach, the version is defined as a set of objects that, when used together with a baseline model in a model transformation, results in the required model. In both cases, the differences between the two different versions of the model are called model differences. In this report we will consider the model differences between the two successive versions of one model.

We focus only on a state-based approach to model versioning. We choose the state-based approach because it is more generic than the operation based approach. This is the case because in state-based approaches explicit support by the modeling tools is not required, while this support is an absolute necessity in operation-based approaches. This limits the use of operation-based versioning.

As our first result, in Section 2, we show that, in state-based approaches, model differences require a locally unique labeling of all model elements. Next, we discuss the problem of presenting and calculating differences between ambiguous models (models that contain structurally identical elements), and show that our labeling technique, in combination with the appropriate differences metamodel, solves that problem. In Section 3 we specify a generic (i.e. tool-independent) method for assigning unique labels to model elements in versioned models. It is important to note that this technique is not suited for assigning universally unique labels to model elements (UUIDs), but is nevertheless crucial from the perspective of model differences.

2 State-based model versioning and differences

In this section we support our claim that, in state-based model versioning, it is necessary to have a method for assigning unique labels to all elements of a versioned model, for example named A, in order to be able to obtain the model differences between model A and another model. We will use the term *model element identifiers* or just *identifiers* instead of *model element labels* in the rest of the text. Also, we discuss why some traditional approaches to the calculation of differences have problems in dealing with ambiguous models, and show that our approach solves these problems.

First, we give a set of requirements that model differences must fulfill in order to be used seamlessly in model configuration management systems. The differences should be:

- Model based: The differences should be represented by a formal differences model.
- Minimalistic: The differences should contain a minimal number of objects.
- Self-contained: The differences model must contain all the information autonomously without relying on data contained in the compared models.
- Transformative: It should be possible to transform one model into another model using their differences model.
- Invertible: It should be possible to revert back to the initial model using the transformed model and the differences model.
- Compositional: The result of subsequent or parallel modifications is a differences model whose definition depends only on difference models being composed and is compatible with the induced transformations.
- Metamodel independent: The differences metamodel should be independent of a particular metamodel (e.g. UML).

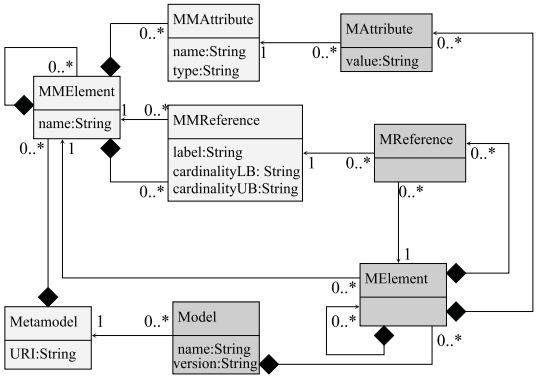

Figure 1: Enhanced metametamodel - EMMM

- Layout independent: The differences metamodel must be agnostic of the presentation issues.

The specified requirements were introduced in [3], and the reader is asked to consult the referenced source for the details on these requirements.

There are several approaches for representing and calculating model differences that treat the differences in accordance with some, but not all, of the specified requirements [4, 8, 7]. However, our research presented in [9], offers a treatment of model differences that satisfies all of the specified requirements. Thus, we use a generic metametamodel defined in [9] and depicted in Figure 1 to convey our ideas. This is a metametamodel expressive enough to represent all graph-based models, but it is specifically tailored to allow for the efficient representation of model differences.

The most important fact to notice is that each model element that is referenced by another element should have a *locally unique identifier*. This stems from the fact that in case the model is persisted from short-term memory to long-term memory (e.g. hard disk), in order to correctly restore the relationships between model elements after the model is loaded back into short-term memory, those relationships must also be represented in some way and persisted together with the model. The representation of references by using local unique identifiers is not the only possible solution, the references could also be represented by using a relative or absolute path from the initial model element to the referenced model element. However, representing references by paths cannot deal with ambiguous models directly, and thus is less preferred than representing references by using locally unique identifiers. Notice that this fact is valid in all metametamodels that use references or associations as parts of their metamodels (e.g. MOF [2] or Ecore [1]).

The introduced metametamodel allows a metamodel-independent representation of model differences. In this report we will assume that model differences conform to the metamodel-independent differences metamodel defined in [9], depicted in Figure 2. The differences model represent the differences between two

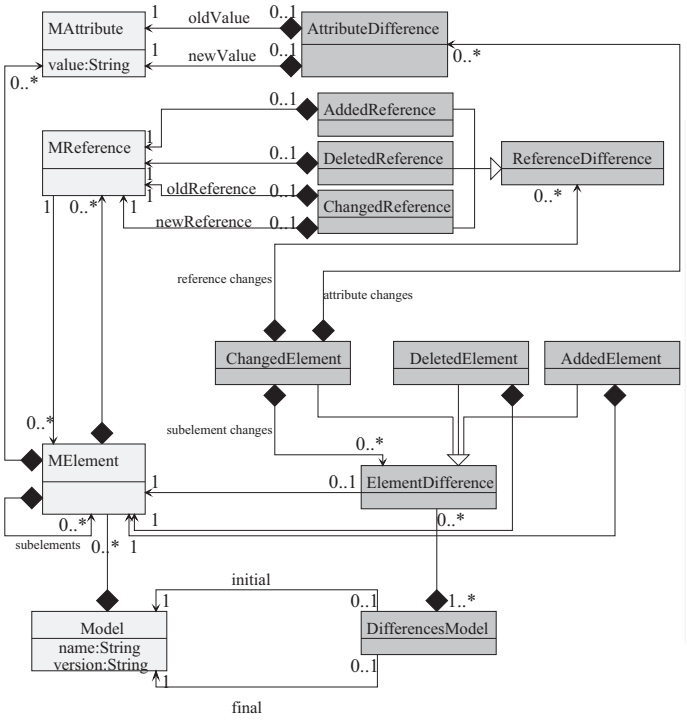

Figure 2: Differences metamodel

models and can be calculated by using a difference calculation algorithm [6, 9]. Observe that, as specified by the introduced differences metamodel, the instances of differences model elements reference the instances of model elements. In fact, it is possible that each model element is referenced by a differences model element.

Based on these two observations - that referenced model elements must have locally unique identifiers and that model elements of differences models reference (possibly all) model elements we conclude that *all* model elements must have locally unique identifiers. However, since modeling tools do not have to persist locally unique identifiers of *all* model elements, but only of the *referenced* model elements, some model elements might not have been assigned locally unique identifiers. This is a problem since state-based model differences require *all* model elements to be assigned locally unique identifiers. Thus, there should be another mechanism that ensures the assignment of locally unique identifiers to *all* model elements, regardless of a tool used to create (or edit) models. This mechanism should ensure that the assigned identifier is identical for each model element every time this mechanism is used on the same model. This mechanism is presented in the form of a labeling algorithm in the next section of this report.

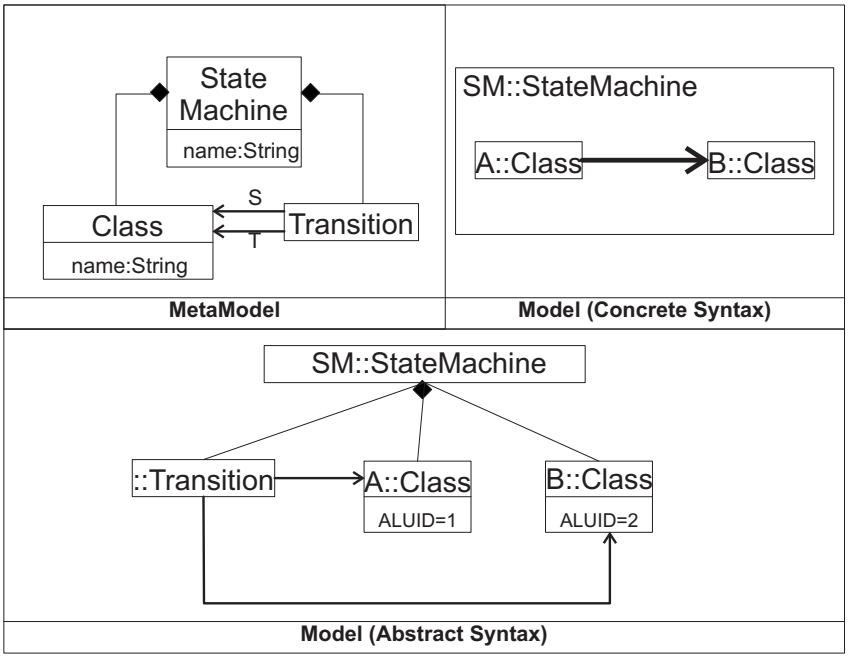

Figure 3: Example metamodel and two representations of the same model

3 Labeling model elements with generated locally unique identifiers

In this section, we present an algorithm that can be used to assign locally unique identifiers to model elements that are part of models which are instances of the metametamodel depicted in Figure 1. Before we define the algorithm itself, we give some preliminaries.

First, we require that, without loss of generality, models are available in their persisted form (i.e. the models are available as files). In this case we ensure a truly tool-independent approach.

Next, we repeat the claim that all the *referenced* model elements must have been assigned locally unique identifiers by tools. We will call these identifiers *assigned local unique identifiers* - ALUID. We will treat ALUIDs as strings in the rest of the text. An example metamodel and two representations of a model conforming to that metamodel are depicted in Figure 3. Since classes are referenced by transitions, the two classes in the example model have been assigned ALUIDs, but since the transitions are not referenced, ALUID is not assigned to the transition in the example model.

As a result of this algorithm, all model elements will be labeled by a *generated local unique identifier* - GLUID. Note that the algorithm generates identifiers that are identical for a specific model element in a versioned model, each time

the algorithm is invoked, up to the level of structurally identical model elements. Notice that, in another version of a model, GLUIDs might not be the same for the same model elements, and thus GLUIDs cannot be used as universally unique identifiers (UUIDs). For a set of identical (ambiguous) model elements in one model, the identifiers are selected from a fixed set of identifiers. However, since these elements are not semantically distinguishable from differences models, this labeling is sufficient.

Since the introduced metametamodel declares that models are hierarchical (tree) structures, we will call the structure of a model - a model tree.

3.1 Labeling algorithm

The labeling algorithm has two phases. In the first phase a bottom-up traversal of the model tree is used to assign temporar local unique identifiers or TLUID to all model elements. In the second phase, based on the assigned values in the first step, the model tree is traversed top-down and the GLUIDs of the model elements are calculated.

3.1.1 First phase

In the first phase, each model element M is labeled with the TLUID. If a model element has been assigned ALUID, then the hash code of this ALUID will be used as TLUID of this element. Otherwise, the TLUID is calculated in the following manner: We will label the attributes (instances of $MAttribute$ element) of a model element as A_i, $i = 1..N$. We will label the references (instances of $MReference$ element) of a model element as R_i, $i = 1..K$. We will label the subelements of (elements contained in) a model element as O_i, $i = 1..M$.

The labeling of attributes is done by sorting all attributes by the string which is a combination of their value and type. Next, each attribute gets enumerated by the position of the string representing that attribute in the sorted list of all attributes. The labeling of references will be done by sorting all references by the string which is a combination of the ALUID of their referenced elements and the name of the reference. Next, each reference gets enumerated by the position of the string representing that reference in the sorted list of all references. Notice that since in the preliminaries we required that all referenced elements have been assigned ALUIDs, this unique representation is possible. The labeling of subelements will be done by sorting all subelements by the string obtaining by combining their TLUID and the ID of a *metamodel* element that the selected subelement conforms to. Next, each subelement gets enumerated by the position of the string representing this subelement in the list of strings of all subelements. Since in this phase the tree traversal is done bottom-up, the TLUIDs of all the children elements of elements on the higher level in the tree have been calculated already, thus this labeling is possible.

Next, in order to increase the speed of the algorithm, we use a hash function and calculate the TLUID of the current element as: $TLUID(M) = hash(value(A_1)+$

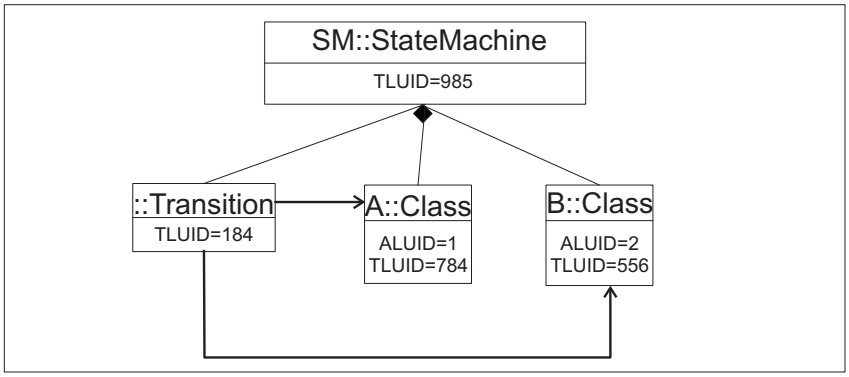

Figure 4: Example model after the first phase of the algorithm

$value(A_2)+..+value(A_N)+ALUID(R_1)+...+ALUID(R_N)+TLUID(O_1)+...+TLUID(O_N)$). Notice that the TLUID of a model element would be valid even without the use of a hash function, but in order to have a faster algorithm a hash-function should be used.

For example, after the first phase, a model depicted in Figure 3 will be changed such that all its elements will be assigned TLUIDs as depicted in Figure 4.

3.1.2 Second phase

In the second phase, a top down traversal of the model tree is performed. For all model elements that are children of the instance of the *Model* element, the GLUID is calculated by sorting the TLUIDs of those model elements, and enumerating the model elements by the position of that model element TLUID in the list of all model elements TLUIDs, appended to a "." (for example, GLUIDs for these model elements might be: ".1", ".2", etc.).

For each of those model elements, the following recursive algorithm is applied: For each model element, all subelements are sorted by their TLUID, and each subelement is labeled by a concatenation of a GLUID of a parent model element plus "." and plus the string value of the position of the TLUID of this subelement in the sorted list. The recursion continues for all subelements of a model element, until there are no more subelements to process.

It might be the case that two or more subelements have the same TLUID, and in that case the model is ambiguous - it contains two or more model elements which have identical structure. If the model elements with the same TLUID are subelements of different model elements, then the generated GLUID will be different for all of them. Thus, in this case, the fact that model is ambiguous poses no problems. In other case, when at least two model elements with the same TLUID are subelements of the same model element the assigned GLUIDs might be different for those model elements each time the labeling algorithm is invoked. However, since these elements have identical structure, and do not have

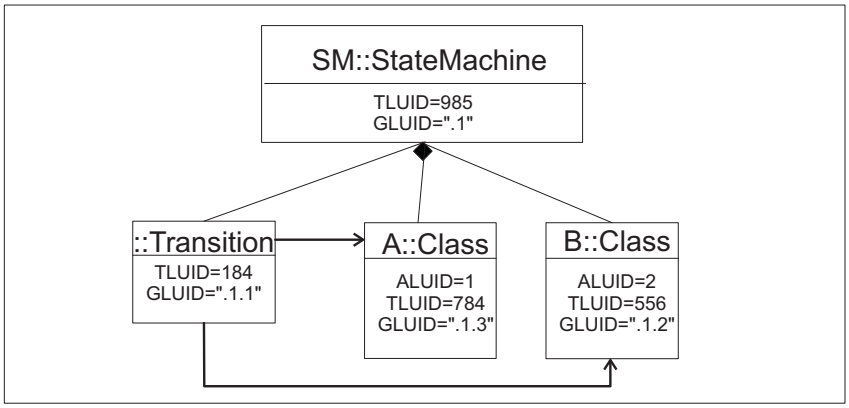

Figure 5: Example model after the second phase of the algorithm

tool assigned identifiers, it is not possible to distinguish them from the outside of model. Since, by construction of the labeling algorithm, all of the subelements of the model element are sorted in the same way, then each time the labeling algorithm is invoked, the elements having the same TLUID will be mapped to the same range of natural numbers. Even though a subelement having the same TLUID as another subelement might have different enumeration, it will always be from the same range of the natural numbers. Since it is not possible to distinguish between elements that have the same TLUID, the fact that the model element as entity will obtain a different GLUID each time, does not pose a problem, since all elements having the same TLUID are anyway considered the same entity from outside of the model.

After the second phase, a model depicted in Figure 3 will be changed such that all its elements will be assigned GLUIDs as depicted in Figure 5.

3.2 Discussion

In this section we discuss the correctness of the algorithm. We discuss three questions: are all elements in a model labeled, are those labels the same each time, and are those labels unique?

We will first prove that all elements in the model are assigned a label. Assume that there is an element that has not been assigned a label. This is impossible if this element is a top-level element, because of the construction of the algorithm. Then, because in the second step of the algorithm all elements that have a parent are assigned a label, this element does not have a parent. However, that would mean that this element is a top level element, and this is impossible because in the second step of the algorithm all top level elements are assigned a label.

In order to answer the second question, remember that this algorithm should be used only with versioned models. Thus, each time the algorithm is run, the model has absolutely identical structure (including the unique identifiers

assigned by the modeling tool to the referenced elements, i.e. ALUIDs). Then, by using the same sorting algorithm each time, the attributes, the references, and the subelements of a model element will always be sorted in the same way. So, by construction of the algorithm, it is clear that the model elements that are leaves of the model tree will have the same TLUIDs. Thus, since the model elements higher in the hierarchy will sort the subelements TLUIDs in the same way, model elements higher in the model tree will also have the same TLUIDs assigned each time. Next, since in the second step of the algorithm all elements with the same TLUID have the same GLUID assigned, all elements in the first level of the tree will have the same GLUIDs, and by recursion it is clear that all elements deeper in the tree will have the same GLUIDs assigned, based on their position in the tree and their TLUID (which is always the same for the same model).

The third question, or the uniqueness of the labels, is correct by construction of the second step of the algorithm.

4 Conclusions and Future Work

In this report we show that it is required to assign labels to all model elements in order to use those models in a state-based model comparison. Furthermore we show that this technique is also applicable to ambiguous models - models that contain structurally identical elements having the same parent. Next, we define a labeling algorithm which assigns locally unique labels to all elements of one version of the model. The labeling algorithm has been described on models conforming to our own metametamodel, but is also applicable to traditional metametamodels such as MOF or Ecore. The optimization (increasing the speed) of the labeling algorithm is the main direction for future work.

Acknowledgements

This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

References

[1] Ecore. `download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html#details` (Viewed April 2010).

[2] MetaObject facility. `http://www.omg.org/mof/` (Viewed June 2010).

[3] A. Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Universita' degli Studi dell'Aquila, 2007.

[4] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, pages 165–185, 2007.

[5] R. Conradi and B. Westfechtel. Towards a uniform version model for software configuration management. In *ICSE '97: Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 1–17, London, UK, 1997. Springer-Verlag.

[6] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for UML models. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 105–116. GI, 2005.

[7] P. Konemann. Model-independent differences. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM '09)*, pages 37–42, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Y. Lin, J. Gray, and F. Jouault. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, August 2007.

[9] M. van den Brand, Z. Protić, and T. Verhoeff. Fine-grained metamodel-assisted model comparison. IWMCP 2010.