Tracing Evolution Changes of Software Artifacts through Model Synchronization *

Igor Ivkovic[1] **and Kostas Kontogiannis**[2]
[1]Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L3G1 Canada
iivkovic@swen.uwaterloo.ca
[2] Dept. of Electronic and Computer Engineering
Technical University of Crete
Chania, 3700 Greece
kkontog@softnet.tuc.gr

Abstract

Software evolution encompasses all activities related to engineering software, from its inception to retirement. Propagating change across software models that are altered due to maintenance activities is a first step towards maintaining consistency between architectural, design, and implementation models. Model synchronization techniques initially presented within the context of Model Driven Architecture provide an instrument for achieving change traceability and consistency. In this paper, we present a framework whereby software artifacts at different levels of abstraction such as architecture diagrams, object models, and abstract syntax trees are represented by graph-based MOF compliant models that can be synchronized using model transformations. In such a framework model dependencies are implicitly encoded using transformation rules and an equivalence relation is used to evaluate when two models become synchronized.

1. Introduction

One of the major problems the software industry is facing is the drift of software designs and architectural descriptions from the source code that is constantly changed due to evolution or maintenance activities. Evolution of software includes modifications performed at all stages of the software development lifecycle, from inception to retirement. Such modifications are performed on a wide range of software artifacts ranging from those at a high level of abstraction, such as business processes, to architecture, design, and source code-level artifacts, such as architecture graphs, object models, and abstract syntax trees. Each change that is performed in any of these models is not always carried out in a systematic fashion, and maintenance problems occur when a change on one artifact is not consistently mapped and applied to another artifact of a software system. The complexity of each change mapping is inherently compounded with decisions regarding potential information loss or gain related to differing levels of expressiveness of assorted artifacts.

In this paper, we investigate the problem of synchronization between software models when one is altered due to evolution or maintenance activities. Furthermore, we present a framework for what we refer to as Model-Driven Software Evolution (MDSE) paradigm that is based on principles of Model Driven Architecture (MDA) [7]. In this context, our view of software is in terms of models, each at a different level of abstraction (*i.e.*, requirements, architecture, design, implementation). Each such model conforms to and is an instance of a corresponding metamodel. [17] Our research is focused on metamodels that have a common graph theoretical basis and are compliant with the Meta-Object Facility (MOF) [10].

Model synchronization can be explicit or implicit. In explicit approaches, the dependence relations between two models are directly defined and encoded (*i.e.*, through a static table). These approaches are rigid as they require the creation and maintenance of structures necessary to track dependencies between models. For instance, relations among models that are once established would need to be updated every time a source or target model is changed. In

[1] This work is funded in part by the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto.

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

implicit approaches however, the relations between models are defined in higher-order terms (*e.g.*, relations between metamodels), and actual dependence relations between models are hence implied. The implicit approaches would offer definite advantages over explicit ones by permitting the establishment of dependency relations that are more flexible, customizable, and easier to maintain. Such relations would not be updated every time a model instance is changed, but would be updated when demands imposed on the synchronization method are changed. In practice, implicit synchronization will not suit all scenarios and a hybrid approach that emphasizes implicitness while supporting explicit mappings would be desirable.

Our goal in this paper is to present a methodology for model synchronization, where change traceability is its central part. For achieving this goal, we first aim to derive a framework that can be used to represent and map models in a way that provides enough semantic information for synchronization. Secondly, we aim to derive a traceability framework, compliant with our representation framework, for tracking changes so that each alteration action is viewed in terms of its atomic units. Finally, we intend to provide a process by which our methodology can be applied to achieve synchronization of models at different levels of abstraction.

To realize the first goal, we introduce an intermediary Graph Metamodel for Synchronization (GMS). This metamodel is an instance of MOF but is less abstract and more capable of providing desired semantic detail. To address the second goal, we introduce a Transformation Metamodel for Synchronization (TMS), where we view each model alteration as a combination of graph changes: insert node/edge, modify node/edge, and delete node/edge. For the third objective, we provide a synchronization algorithm that is based on dependency relations implicitly defined by mapping source and target metamodels as graphs using GMS.

The rest of this paper is organized as follows. Section 2 presents related research and discusses how our findings build on previously published results in the area of model transformation and traceability management. Section 3 presents the basics of model synchronization and examines related aspects. Section 4 introduces our approach to model synchronization, including the representation framework, the traceability framework, and the algorithm for model synchronization through traceability. Finally, Section 5 gives the conclusions and directions for future research.

2. Related Research

Systematic change of software models is attained through model transformation. Czarnecki and Helsen provide a classification of model transformation approaches [2]. Based on this classification, each model transformation consists of transformation rules that are combined to achieve a particular change. Each transformation rule consists of two distinct parts: a left-hand side (LHS), which refers to the source model, and a right-hand side (RHS), which refers to the target model. Both LHS and RHS can be represented as a combination of (1) patterns, such as string, term, and graph patterns; (2) logic, such as computations and constraints on model elements; and (3) variables, which hold model elements of source, target, or some intermediary model. Transformation rules can be unidirectional or bidirectional, and can be parameterized to allow configuration and tuning. In our approach, we do not focus on specification and analysis of model transformations, but instead view transformations as components of the model synchronization framework. Based on the traceability classification from [2], we intend to derive traces that are stored separately while the traceability control will be automatic for all rules and synchronization control semi-automatic (*i.e.*, automatic for some rules).

Engels *et al*. in [5] have discussed the transformation of UML Class Diagrams and UML Collaboration Diagrams into Java code. They have shown how to deal with both the structural and behavioral (*e.g.*, flow) mapping problems between UML and Java using a pattern-based transformation algorithm. The pattern used is an instance of a metamodel from which one can identify parts of the source diagram that is to be transformed. The goal of this approach was preservation of semantic information through transformation but there was no discussion on how the defined transformations could be used in model synchronization.

Shen *et al*. [12] have used UML stereotypes to extend the UML metamodel in order to support model refinement. They have shown how Object Constraint Language (OCL) expressions can be used to define synchronization rules. The limitation of their approach is that they only support changes of UML static models at different levels of abstraction, without considerations of synchronization with other modelling languages.

In the area of consistency checking, change synchronization is interpreted as synchronization of model views. Fradet *et al*. [6] consider synchronization of multiple views of software architecture by first defining each view formally and then algorithmically checking inter-view consistency. Wirsing *et al*. [15] have looked at synchronization of views across the entire development lifecycle by creating an intermediary comparison view. Engels *et al*. have used the Communicating Sequential Processes (CSP) language for checking behavioral consistency among views [4]. Finally, Larsson and Burbeck [9] have shown a model-view-controller approach to consistency checking by insisting that each transformation between a model and the views has an inverse that can be automatically calculated. In our ap-

proach, we have decided to base our methodology on graphs as a common basis and view all synchronized models as graphs. We have also decided to express our framework in less-formal terms (*e.g.*, OCL) in order to facilitate its easier adoption and extension to practical application scenarios.

The importance of traceability and its relation to synchronization was discussed in the area of traceability management. In [3], Desfray points out the importance of traceability management in the MDA-based approaches to ensure consistency among models. The key aspects of traceability are identified as follows:

- Element identification, where using the name of the element as an identifier, was said to be spurious. Instead, it is mentioned that assigning a universally unique identifier to each model element should lead to a more successful traceability management.

- Implicit traceability, where related tools should transparently maintain relations among model elements based on natural properties of models (*i.e.*, traceability based on inherent relations). For example, a message in UML can be related to an operation or can link to an association.

- Explicit traceability, where it will be necessary in certain cases to manually define explicit relations between elements. For example, a link could be created between a Use Case and classes that implement it.

- Traceability and external elements, where external artifacts such as documentation or source code are related to model elements, but they themselves are managed externally. Under such conditions, it is crucial that each external element is uniquely identified by the modelling environment in order to maintain traceability.

Kowalczykiewicz and Weiss confirm these points and also reiterate the importance of traceability integration into software development [8]. Cysneiros *et al*. [1] demonstrate how to use traceability to synchronize i* [16] with UML models. They also demonstrate how to implement model synchronization using XML. However, their framework uses synchronization rules that identify elements by their names and other lexicographical properties, and not based on their unique identifiers as suggested above. Within our approach, we follow the directions for traceability and define an unique identifier for each model element. We incorporate implicit traceability through a mapping between metamodels while leaving an opportunity for explicit relations to be defined through explicit mapping tables.

3. On Model Synchronization

From the related work above [9], a definition of "model synchronization" can be inferred as the process of establishing an equivalence between two models when one of them is altered. The equivalence can be defined through a set of dependency relations between model elements. Additional concerns that may apply include:

- Source and target models can be at the same (intra-model synchronization) or different (inter-model synchronization) level of abstraction. Out of the two, the intra-model synchronization in general would be simpler, as dependencies among model elements would be implied through reflection of the same applicable metamodel. The inter-model synchronization would in general require dependency mapping between different metamodels; for example, as in synchronization of software architecture models with design class diagrams or source code models. We should also note the existence of special cases such as UML, where models that might be at different levels of abstraction would be derived from the same UML metamodel.

- Source and target metamodels can be different and be used to express information at notably dissimilar levels of abstraction (*e.g.*, synchronization of use cases with abstract syntax trees). Mapping dependencies among these metamodels would be quite hard and meaningful synchronization without intermediary models might not be always possible.

- Source and target models may be initially synchronized and the applicable equivalence relation holds. We refer to this scenario as "*Model Synchronization Through Traceability*", and this is the central topic of this paper. In this case, changes performed on the source model are traced, translated, and then applied to the target model. After each iteration, the corresponding equivalence relations are checked to verify that synchronization was performed successfully; if the check fails, additional input will be required.

- Source and target models may also not be initially synchronized and the applicable equivalence relation does not hold. We refer to this scenario as "*Model Synchronization Through Equivalence*" and is a point of future research. In this case, the source model is transformed into its equivalent representation in terms of the target's metamodel. Additional elements of the target model that do not violate the equivalence relations are then added to the newly derived model. The new model once verified to be consistent with the source would then replace the target model.

- Finally, when source and target models can not be synchronized for the given equivalence relation, additional input is required. To achieve synchronization, changes must be made more elaborate or could be performed manually to modify either the affected equivalence relation or one of the models. This topic is out of the immediate scope of the paper where we assume that the models *can* be synchronized.

4. Model Synchronization Through Traceability

In order to derive a model synchronization methodology, our three principal objectives are as follows:

1. To introduce a graph formalism as a representation framework for model synchronization.
2. To derive a transformation framework for model synchronization based on GMS.
3. To derive a synchronization algorithm based on implicitly defined dependence relations, as means by which our methodology can be applied in practical problems.

4.1. Graph Metamodel for Synchronization

Towards our quest for defining a framework for model synchronization, a fundamental question that needs be answered is related to the formalism that should be used to represent models that relate to software artifacts at different levels of abstraction. This formalism has to be concise, formal, yet easy to understand, use, and process.

The objective for such a formalism is not only to allow simplification and enable model synchronization, but also to maintain enough semantic detail so that implicit dependencies among models could be formally established. Our first attempt for a solution to this problem was to use MOF, which provides a solid foundation for the specification of other models. However, we concluded that MOF is too abstract for purposes of synchronization, as it lacks a method for expressing relevant model semantics. In addition, establishing dependency relations among models where MOF is a common method of representation is not trivial and is prone to ambiguity due to an inherently high abstraction level. Instead of MOF, we have considered graphs as a common denominator for representing models. These graphs can then be modelled in MOF for implementation and processing purposes. Graphs provide a simple structure — they represent an organization of nodes and edges — and allow easy establishment of dependencies among instances. Augmented with labels, types, attributes, and directions, graphs are also capable of supporting semantic detail that is required for synchronization. Other approaches that were considered, such as formal logic, did not offer the well suited balance between simplicity and expressiveness found in graphs. As a result, we have decided to use graphs as the core of our representation approach.

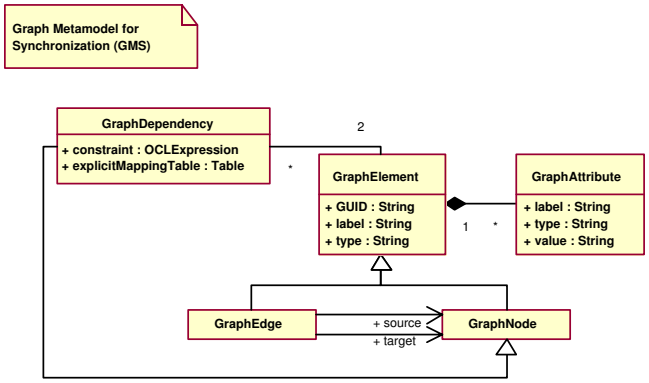

Figure 1. Graph Metamodel for Synchronization

To express GMS (Figure 1) and other metamodels in this paper, we have utilized MOF / UML semantics [11]. GMS is represented in terms of MOF and it was derived to denote directed, typed, labelled graphs with attributes [13]. Each such graph G consists of a set of nodes and edges where each element has its own Globally Unique IDentifier (GUID) and each edge connects exactly two nodes. Each node or edge has a label and type (*e.g.*, "Customer :: UML Class" for a node, "Uses :: UML Association" for an edge) and can additionally have attributes with their own names, types, and values. Dependencies among graphs are established through dependency nodes, which contain applicable constraints (*e.g.*, expressed in OCL-compliant notation [14]). Dependency nodes could also contain a mapping table, which allows explicit mapping among specific model elements using their GUIDs. This table would contain three columns: first for the identifier of the source element, second for the equivalence operator defined globally for all models or locally for specific dependency, and third for the identifier of the target element. Examples of equivalence operator include (1) explicitly-existential, where neither source nor target element can exist without the other, (2) existential, where only source cannot exist without the target, (3) containment, where source contains the target element, *etc*. An example of the containment operator definition is given in Figure 3.

To demonstrate the usage of GMS, we show a UML Class Diagram Metamodel for Synchronization (Figure 2). This model, meant as a simplified example of the UML metamodel, includes the integral elements that are necessary for modelling UML classes and their relationships, and

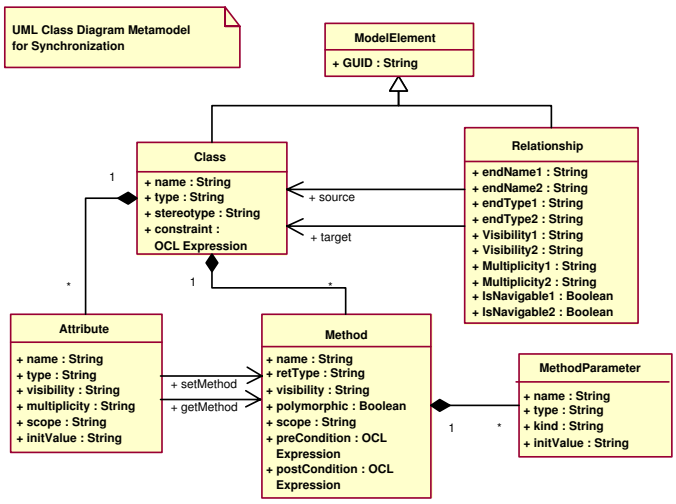

Figure 2. UML Class Diagram Metamodel for Synchronization

it can be mapped to GMS as shown below.

- Class, Relationship, Attribute, Method and MethodParameter entities map to GraphNode, where type of the node (*e.g.*, "UML Class") maps to "type : String" and name of the attribute (if available) maps to "label : String".
- Each of the other attributes (*e.g.*, "stereotype : String") maps to GraphAttribute, where the type (*e.g.*, "visibility") maps to "type : String" and the value (*e.g.*, "public") maps to "label : String".
- Associations among classes map to GraphEdge, where association label (*e.g.*, "source") maps to "name : String" and association type (*e.g.*, "aggregation") maps to "type : String".

The mapping between the derived metamodel for synchronization and GMS is configurable and is based on the desired level of semantic granularity required for synchronization of model elements. For instance, the metamodel presented in Figure 2 is based on the assumption that the synchronization of models instantiated from this metamodel would demand the level of semantic detail presented here. Accordingly, each of the model elements that had several attributes beside its name and type was mapped to a node, and its attributes were mapped to graph attributes. In comparison, if some of the model elements did not warrant the same level of detail and did not include attributes relevant to synchronization, they could themselves be represented as graph attributes.

Based on this example, metamodels for other models that are used to support evolution activities can be derived. Moreover, any two metamodels compliant with GMS that represent models suitable for synchronization (*e.g.*, models from successive stages of development lifecycle) can be matched by establishing dependency relationships. Each dependency node in between two metamodels would, through its existence, indicate that two types are dependent. And, through constraints internal to the node, it would precisely define the mapping between element types and their related attributes. It is worthy to note that each dependency node would connect to exactly one element from each metamodel. However, each metamodel element could have more than one dependency node connected to it, thereby allowing one-to-many or many-to-many relations.

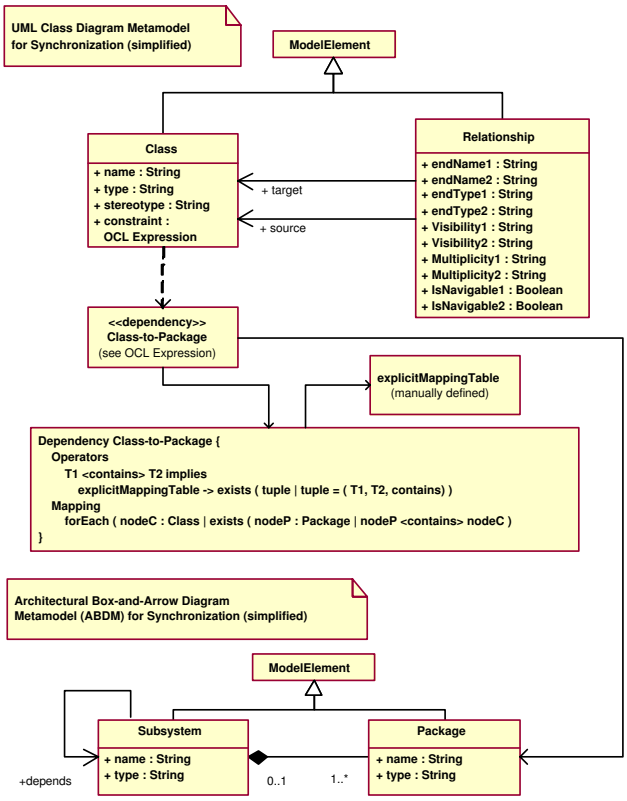

Figure 3. An Example of Metamodel Mapping for Synchronization

To demonstrate the mapping between GMS-based metamodels, we show a an example of Architectural Box-and-Arrow Diagrams and UML Class Diagrams (Figure 3). This instance is based on the assumption that each class in the UML model must be a part of a package in the architectural model and cannot exist otherwise. This assumption is encoded as a containment relationship specified using OCL.

This condition would during synchronization imply that if a package that contains a particular class is deleted, that class and all its attributes, operations and associations would also be deleted. Within the mapping, we make a reference to the explicit mapping table that can be used to manually define containment relations among suitable elements. This table is used to verify the assumption behind the dependence relation.

In our example (Figure 3), we utilize OCL to express relations between individual meta-classes. However, this approach can be adapted to express relations between patterns of meta-classes by choosing a different specification language (*e.g.*, graph grammars).

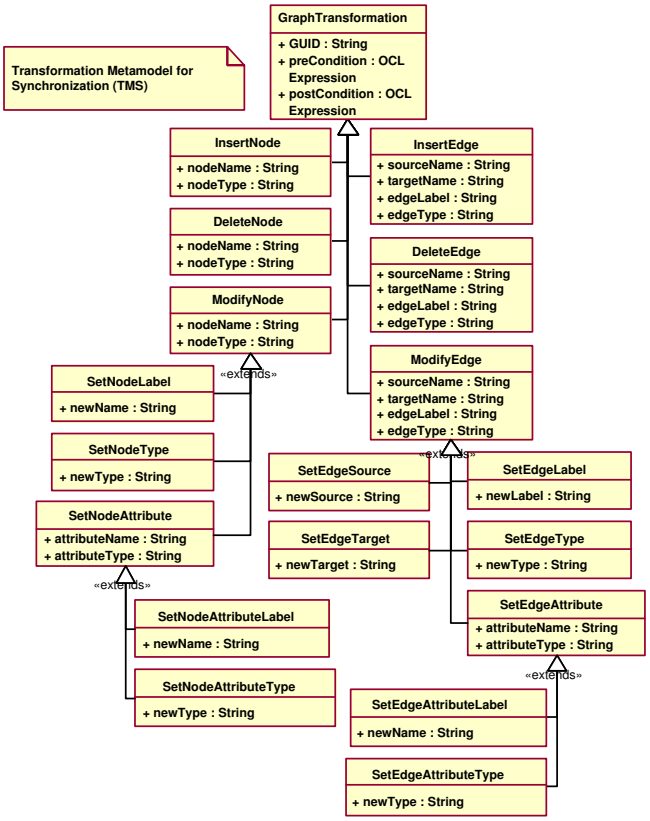

Figure 4. Transformation Metamodel for Synchronization

The presented example underlies the configurability of our approach, where the level of granularity and semantic detail is left fully customizable. Software engineers adapting this framework to their usage scenarios would not be limited by our approach and could easily instantiate their own synchronization mapping.

4.2. Transformation Metamodel for Synchronization

Using GMS as the basis for representation of models, the next goal is to derive a common transformation approach that could be used to establish traceability capabilities for synchronization. Given that all models in our problem scope are viewed as graphs, their changes at their most basic (atomic) level could be viewed as simple graph changes (*e.g.*, insert, modify, or delete node or edge). From there, the actual changes performed could be represented and traced as combinations of graph changes. Hence, we define a transformation metamodel for synchronization, which is an abstraction of actual changes performed on models and is based on GMS. Each (source) metamodel for synchronization $\mathcal{M}_{GMS}$ instantiated from GMS would also have a complementary (target) metamodel $\mathcal{M}_{TMS}$, which would be an instance of TMS and would define transformations for all instances of $\mathcal{M}_{GMS}$. It also holds that all model transformations performed on models that comply with $\mathcal{M}_{GMS}$ would then have to comply with the matching $\mathcal{M}_{TMS}$.

More specifically, TMS, as expressed in Figure 4, identifies basic graph operations as a hierarchy, and is based on the assumption that performing a particular operation will not violate the structural integrity of a graph. For that purpose, each graph transformation will have its own globally or locally defined pre and postconditions, which would ensure the preservation of the structural integrity. For example, a node that is removed should not leave dangling edges, but should be preceded with the removal of affected edges.

The mapping of TMS to an instance for UML Class Diagram transformations is not complex and is described as follows.

- Each change to one of the entities of the class diagram is mapped to its abstract representation in terms of graph changes based on the mapping of UML class diagram metamodel with GMS. For example, operation "InsertClass" is mapped to "InsertNode" and operation "DeleteAssociation" is mapped to "DeleteNode". Also, operation "SetClassName" is mapped to "SetNodeLabel" and operation "SetAssociationMultiplicity" is mapped to "SetNodeAttribute".

- To maintain structural integrity, required pre and postconditions are set in the root of the transformation model hierarchy. These constraints specified in OCL could, for example, ensure that class is not removed until its attributes, operations, or associations are removed or reassigned. Given that each operation inherits its properties from the top level "GraphTranformation" operation, these conditions would apply to all operations across the transformation model.

This UML Class Diagram Transformation Metamodel in conjunction with similarly defined Architectural Box-and-Arrow Diagram Transformation Metamodel could be used to map and enact changes between models of these two types, as discussed in the next subsection.

4.3. An Algorithm for Model Synchronization Through Traceability

Before specifying the synchronization algorithm and its details, it is necessary to mathematically define the notation and the problem.

Let a *model* $\mathcal{M}$ be a set of typed nodes, edges, and properties on these nodes and edges. Each $\mathcal{M}$ is an instance of a corresponding GMS-compliant metamodel $\mathcal{M}_M$. A *node n* of $\mathcal{M}$ is represented as a tuple (GUID, label, type, $\mathcal{A}$) and an *edge e* of $\mathcal{M}$ as a tuple (GUID, label, type, sourceNode, targetNode, $\mathcal{A}$). In each tuple, *GUID* is a Globally Unique IDentifier used to unambiguously identify nodes and edges while *type* is an element of $\mathcal{M}_M$. In addition, let $\mathcal{A}$ be a set of attributes, where each *attribute a* is represented as a tuple (label, type, value).

The problem of synchronization can then mathematically be described as follows: let $\mathcal{M}$ and $\mathcal{G}$ be two models at different levels of abstraction synchronized through an equivalence relation $\mathcal{R}$ (*i.e.*, $\mathcal{M}\ \mathcal{R}\ \mathcal{G}$). Given a sequence of transformations $\mathcal{T}_m$ that alters $\mathcal{M}$ to yield $\mathcal{M}'$, the problem is to identify a corresponding sequence of transformations $\mathcal{T}_g$ that when applied to $\mathcal{G}$ yields $\mathcal{G}'$ such that $\mathcal{M}'\ \mathcal{R}\ \mathcal{G}'$. We assume for simplicity that each transformation could insert, delete, or modify a node, an edge, or an attribute.

The solution based on GMS and TMS frameworks would be a sequence of translations from $\mathcal{T}_m$ to $\mathcal{T}_g$, where each translation is mapped using one of the rules from a set of applicable synchronization rules $\mathcal{S}$. Each synchronization rule would apply to a particular operation (*e.g.*, InsertNode), be based on a particular premise (*i.e.*, relevant preconditions), and would include corresponding actions. A rule could be expressed in a variety of notations including OCL and graph grammars.

Synchronization rules would be based on dependencies between the metamodels of $\mathcal{M}$ and $\mathcal{G}$. Assuming that *Type1* $\mathcal{D}$ *Type2* for a dependency relation $\mathcal{D}$ where $\mathcal{R} \supset \mathcal{D}$, a simplified template for rules that map InsertNode transformations from $\mathcal{M}$ to $\mathcal{G}$ would be as follows. Note that similar templates would apply for other transformations (*e.g.*, DeleteNode, ModifyEdge).

Case 1. For each InsertNode (label, Type1), assuming that $\mathcal{D}$ is one-to-one mapping with no constraints, execute

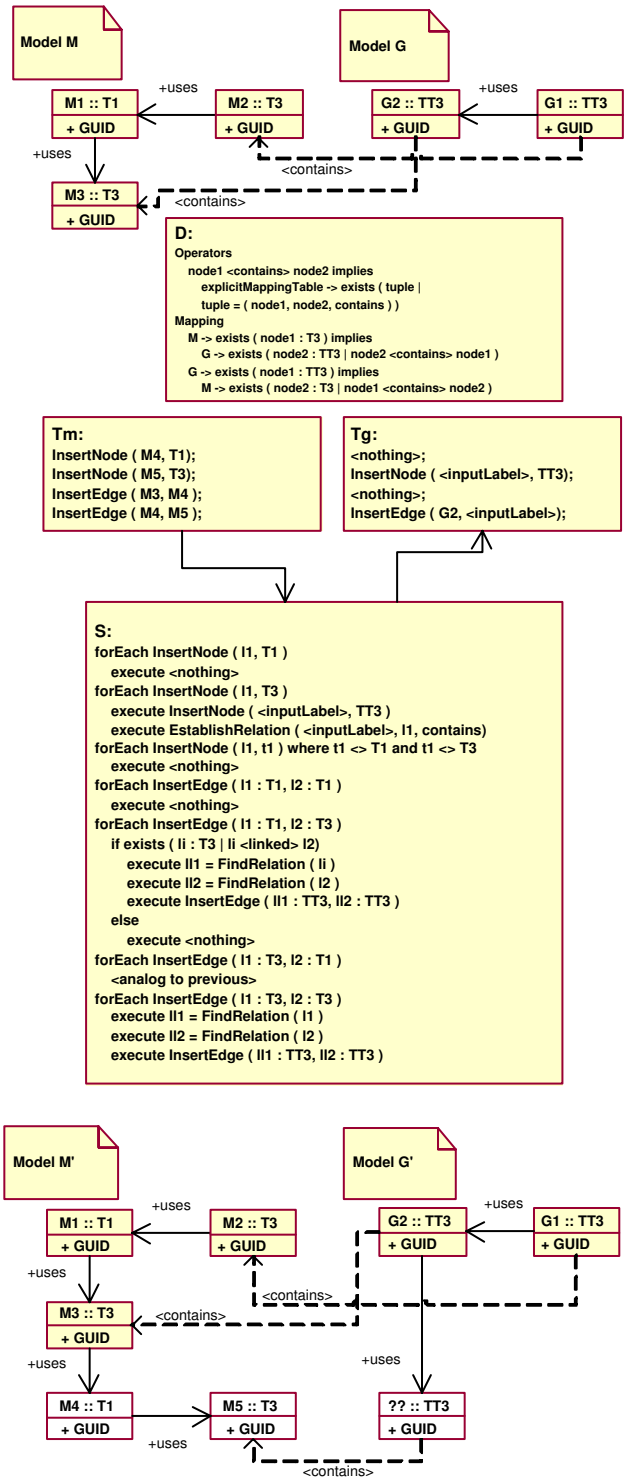

Figure 5. Model Synchronization of Generic Models

InsertNode (<inputLabel>, Type2).

Case 2. For each InsertNode (label, TypeT) where TypeT <> Type1, assuming that $\mathcal{D}$ is one-to-one mapping with no constraints, execute <nothing>.

Case 3 and above. For each InsertNode (label, Type1), assuming that $\mathcal{D}$ is a mapping with constraints, derive a transformation t that satisfies constraints. Else, if constraints could not be satisfied, execute <nothing>.

As it can be seen in examples in Figure 5 and Figure 6, not all of these cases would apply at the same time. Moreover, the rules could also contain implementation specific operations (*e.g.*, FindRelation, EstablishRelation) that are used to comply with the constraints as set in $\mathcal{D}$.

The algorithm for Model Synchronization Through Traceability (abbreviated as MSTT) is as follows.

Algorithm: MSTT

Input:
1. Model $\mathcal{M}$
2. Model $\mathcal{G}$
3. Equivalence Relation $\mathcal{R}$
4. Set of Synchronization Rules $\mathcal{S}$
5. Sequence of Transformations $\mathcal{T}_m$

Output:
1. Model $\mathcal{M}'$
2. Model $\mathcal{G}'$

Steps:

Step 1. Let two models $\mathcal{M}$ and $\mathcal{G}$ be initially synchronized under an equivalence relation $\mathcal{R}$ (*i.e.*, $\mathcal{M}\ \mathcal{R}\ \mathcal{G}$). Also, let $\mathcal{S}$ be a set of applicable synchronization rules.

Step 2. Let $\mathcal{M}$ be modified through a sequence of transformations $\mathcal{T}_m$ resulting in a model $\mathcal{M}'$.

Step 3. For each transformation t_m from $\mathcal{T}_m$, use a corresponding synchronization rule s to obtain transformation t_g. Make each t_g be a part of $\mathcal{T}_g$ in the order analog to $\mathcal{T}_m$.

Step 4. Apply $\mathcal{T}_g$ to $\mathcal{G}$ to obtain $\mathcal{G}'$, and verify that $\mathcal{M}'\ \mathcal{R}\ \mathcal{G}'$. Store the record of transformation, indicating the outcome of applying $\mathcal{T}_g$.

Step 5. If $\mathcal{M}'\ \mathcal{R}\ \mathcal{G}'$ does not hold, identify violated constraints and either manually update the constraints or the model elements that violate them, or declare failure.

To show the usage of the algorithm, we include the following two examples. In Figure 5, $\mathcal{M}$ and $\mathcal{G}$ are two generic models where $\mathcal{G}$ is at a higher level of abstraction than $\mathcal{M}$ and $\mathcal{M}\ \mathcal{R}\ \mathcal{G}$ for some relation $\mathcal{R}$. The set of rules $\mathcal{S}$ for synchronization of $\mathcal{M}$ and $\mathcal{G}$ is based on the dependency

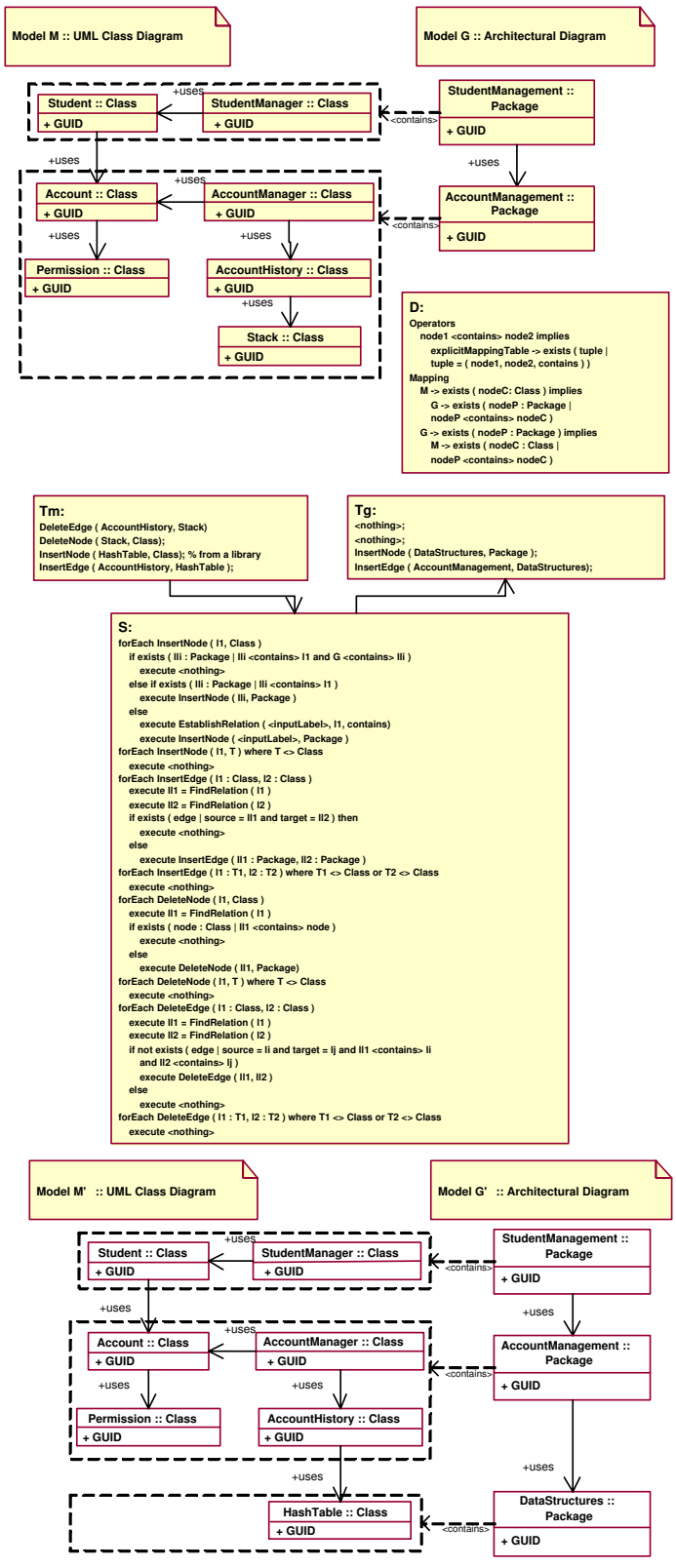

Figure 6. Model Synchronization of Design and Architectural Models

relation $\mathcal{D}$ between types *T3* and *TT3*. The relation $\mathcal{D}$ is illustrated at the top of Figure 5 and it denotes that for each instance of *T3*, there is an instance of *TT3* such that *T3* is contained in the package of *TT3*; and that for each instance of *TT3*, there is an instance of *T3*, such that *T3* is contained in the package of *TT3*. For simplicity, we assume that in this case $\mathcal{D} \Leftrightarrow \mathcal{R}$, while in general, $\mathcal{D} \subset \mathcal{R}$ (*i.e.*, other constraints beside the ones in $\mathcal{D}$ would be present). The model $\mathcal{M}$ is transformed using a sequence of transformations $\mathcal{T}_m$ into $\mathcal{M}'$. The synchronization algorithm maps individual transformations from the given transformation sequence $\mathcal{T}_m$ into transformations for a new sequence $\mathcal{T}_g$. Each mapping is based on a rule that matches the transformation type. For example, InsertNode (M4, T1) is mapped using the first rule from $\mathcal{S}$ — forEach InsertNode (l1, T1) execute <nothing> — into <nothing> (*i.e.*, no execution necessary). Similarly, InsertEdge (M3, M4), where *M3* is of type *T3* and *M4* is of type *T1*, is mapped using the sixth rule from $\mathcal{S}$ into <nothing>, as there is no link between *M3* and another node of type *T3* through *M4*. The resulting sequence $\mathcal{T}_g$ is applied to $\mathcal{G}$ to obtain $\mathcal{G}'$. Finally, it is verified that $\mathcal{M}' \; \mathcal{R} \; \mathcal{G}'$ is true by checking that constraints of $\mathcal{D}$ are satisfied.

Similarly, in Figure 6, we show two models $\mathcal{M}$ and $\mathcal{G}$, where $\mathcal{M}$ is a low-level design model and $\mathcal{G}$ is an architectural model. The model $\mathcal{M}$ is altered through $\mathcal{T}_m$, so that the *AccountHistory* class is implemented not using the internal *Stack* class but the external *HashTable* class from *DataStructures* library. Since *HashTable* is not part of the existing packages, as a result of the mapping of $\mathcal{T}_m$ to $\mathcal{T}_g$, a new package *DataStructures* is created to indicate the external library with a new edge connecting it to the *AccountManagement* package.

4.4. A Process for Model Synchronization Through Traceability

In this subsection, we identify a process that can be used to instantiate our methodology for a practical synchronization scenario. The steps in this process would be as follows.

Step 1. For two models $\mathcal{M}$ and $\mathcal{G}$ that need to be kept synchronized, identify model types $\mathcal{M}_t$ and $\mathcal{G}_t$.

Step 2. For identified model types, derive GMS-compliant representation metamodels $\mathcal{M}_{Mr}$ and $\mathcal{G}_{Mr}$ with enough semantic detail for intended synchronization.

Step 3. Establish necessary dependencies among $\mathcal{M}_{Mr}$ and $\mathcal{G}_{Mr}$ using dependency relationships. If necessary, manually insert tuples into explicit mapping tables to indicate dependencies among specific model elements. Modify $\mathcal{M}_{Mr}$ and $\mathcal{G}_{Mr}$ if necessary.

Step 4. For identified model types, derive TMS-compliant transformation metamodels $\mathcal{M}_{Mt}$ and $\mathcal{G}_{Mt}$ so that they are consistent with $\mathcal{M}_{Mr}$ and $\mathcal{G}_{Mr}$ respectively. To preserve structural integrity, define necessary preconditions and postconditions for those operations that warrant it.

Step 5. Implement traceability capability (*e.g.*, through tracing feature of an IDE, or a model repository) based on $\mathcal{M}_{Mt}$ and $\mathcal{G}_{Mt}$.

Step 6. Verify that $\mathcal{M}$ and $\mathcal{G}$ satisfy dependency conditions and manually update them if not equivalent at the beginning.

Step 7. Assuming interactive synchronization (*i.e.*, end user requests synchronization), fragment the changes performed into basic graph transformations and map them using an algorithm described in the previous subsection. Ensure that a record of transformation is stored after each batch of transformations is processed.

For more automatic support of synchronization, it is clear that this algorithm would have to be implemented using a modelling API and then would have to be integrated into an overall development lifecycle (*e.g.*, made part of the active Integrated Development Environment (IDE)). This implementation problem is out of the scope of this paper, and will be explored in future research.

5. Conclusions and Future Research

In this paper, we have introduced a methodology for model synchronization to achieve traceability of changes of software models that occur during software evolution and maintenance. As part of our theory, we have introduced frameworks for model representation and model transformation that are based on the concept of graphs. Each model that is to be synchronized based on our methodology is viewed as a graph, and each change made on this model is viewed as an instance of a graph change. Based on this approach, we are able to implicitly map changes between models that are based on different levels of abstraction. Within our theory, we have also shown an algorithm that prescribes how our framework can be instantiated and applied to solve concrete synchronization problems.

We have classified this approach as primarily implicit, where relations between model elements are implied from dependency mappings between their metamodels. However, in practice, implicit mappings will not suffice for all conditions. We have therefore incorporated as a first step, a facility for explicit mappings between model elements in terms of mapping tables, which refers to model elements by their unique identifiers.

The focus for this paper is to set the foundations for a methodology and to illustrate its capacity to handle structural synchronization of compatible models (*e.g.*, models from consecutive stages of development lifecycle). As part of on-going work, we have yet to demonstrate the suitability of our approach to handle behavioral synchronization; for instance, synchronization of complex flows. Nevertheless, we believe that the proposed framework is a first step towards establishing a formal, yet tractable framework whereby software models and artifacts at different levels of abstraction can be synchronized and can be maintained consistent with each other when one of these models is altered due to evolution activities.

References

[1] G. Cysneiros, A. Zisman, and G. Spanoudakis. A traceability approach from i* and uml models. In *Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS)*, Lecture Notes in Computer Science, Portland, OR, May 2003. Springer.

[2] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Anaheim, CA, October 2003.

[3] P. Desfray. Mda when a major software industry trend meets our toolset, implemented since 1994. Technical report, SOFTEAM, October 2001.

[4] G. Engels, R. Heckel, and J. M. Kster. Rule-based specification of behavioral consistency based on the uml meta-model. In *Proceedings of the 4th International Conference on the Unified Modeling Language (UML)*, volume 2185 of *Lecture Notes in Computer Science*, Toronto, Canada, October 2001. Springer.

[5] G. Engels, R. Huecking, S. Sauer, and A. Wagner. Uml collaboration diagrams and their transformation to java. In *Proceedings of the Second International Conference on The Unified Modeling Language (UML)*, volume 1723 of *Lecture Notes in Computer Science*, Fort Collins, CO, October 1999. Springer.

[6] P. Fradet, D. Metayer, and M. Perin. Consistency checking for multiple view software architectures. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, Toulouse, France, October 1999.

[7] A. Kleppe, J. Warmer, and W. Bast. *The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[8] K. Kowalczykiewicz and D. Weiss. Traceability: Taming uncontrolled change in software development. In *Proceedings of the IV KKIO Conference*, Tarnowo Podgrne, Poland, 2002.

[9] H. Larsson and K. Burbeck. Codex - an automatic model view controller engineering system. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, Enschede, The Netherlands, June 2003.

[10] OMG. Meta object facility (mof) specification version 1.4. Technical report, Object Management Group (OMG), April 2002. http://www.omg.org/docs/formal/02-04-03.pdf.

[11] OMG. Unified modelling language (uml) specification. Technical report, Object Management Group, March 2003. http://www.omg.org/docs/formal/03-03-01.pdf.

[12] W. Shen, Y. Lu, and W. L. Low. Extending the uml metamodel to support software refinement. In *Proceedings of the Second Workshop on Consistency Problems in UML-based Software Development*, San Francisco, CA, October 2003.

[13] D. Varró and A. Pataricza. Mathematical model transformations for system verification. Technical report, Budapest University of Technology and Economics, May 2001.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[15] M. Wirsing and A. Knapp. View consistency in software development. In *Proceedings of the 9th International Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF)*, Venice, Italy, October 2002.

[16] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1995.

[17] Uml glossary. Online by Rational Software Corporation and MCI Systemhouse Corporation, 2004. http://swiki.hfbk-hamburg.de:8888/MusicTechnology/24.