Incremental Model Synchronization in Model Driven Development Environments

Ali Razavi[1], Kostas Kontogiannis[1,2], Chris Brealey[3], Leho Nigul[3]
[1]University of Waterloo, {arazavi,kostas}@swen.uwaterloo.ca
[2]National Technical University of Athens, kkontog@softlab.ntua.gr
[3]IBM Canada Lab, {cbrealey, lnigul}@ca.ibm.com

Abstract

Most modern model driven software development environments rely heavily on model transformations for generating various software design artifacts and eventually even source code. However, during development, maintenance and evolution activities, these software artifacts are subject to updates and refactoring operations. In such model driven development environments, these software artifacts need to be re-synchronized every time one of them is altered, so that they all remain consistent according to some specific rules, relations and domain constraints. Until now, the standard approach to model synchronization has been the re-application of all transformation rules, aiming thus for the complete re-generation of all artifacts in all models involved. This complete re-application is a safe yet computationally expensive way to ensure consistency among models. In this paper, we present a method for re-synchronizing software models in an incremental fashion by utilizing an indexing model. In this respect, using the proposed methodology, the time required for maintaining global model consistency is proportional to the size of the changes and not that of the models involved. The proposed approach has been applied for the incremental re-synchronization of large and complex models in the Eclipse Web Tools Platform (WTP). Results indicate that this solution can significantly reduce the time required to re-synchronize models in such comprehensive development environments as WTP.

1 Introduction

Throughout their life-cycle, software artifacts are constantly maintained and evolved. These artifacts specify different aspects of a software project including requirements, architecture, design, and implementation, and therefore are syntactically and semantically interdependent. These maintenance and evolution activities are applied in an incremental way and in iterative evolution cycles. As such, changes made to one, impact the others. Modern Integrated Development Environments (IDESs) aim to provide a workspace in which these artifacts can be programmatically accessed and manipulated. In this respect, IDEs strive to provide facilities for transparent and effective propagation of changes across a diverse set of artifacts in the workspace. These artifacts need to be synchronized when one or more of them is altered so they all remain consistent according to spe-

cific rules, constraints and properties.

Artifact synchronization for modern IDEs has to be a fast and responsive process. Ideally, the users should be able to seamlessly reconcile all the dependent artifacts with the modified ones. This requires the synchronization process to be incremental; meaning that the number of changes that need to be made to bring the artifacts into a consistent state should be proportional to the relative size of the changed elements and not the total size of the models. Most of the current tools equipped with model synchronization rely on non-incremental model transformation techniques which can be time consuming for large projects where many models and artifacts are involved.

A so far neglected, yet imperative case of model synchronization, deals with situations in which two or more artifacts are generated from one another by an existing model transformation. In this context, consistency rules between these models are embedded and hidden in the implementation logic of the model transformation rules themselves. More often than not, these model transformations do not support bi-directional mappings; hence unable to reflect the modifications made to the target artifacts back to the source artifacts. Systematic and automated support for synchronization of models under such conditions utilizing a unidirectional model transformer can potentially save a significant amount of effort, which, otherwise, would be required to reverse-engineer the logic of the generating transformations, and re-implement these transformations in a bi-directional and incremental framework. The solution to this problem becomes more obscure for generative development processes which utilize multiple transformations to form a model driven software development chain.

In this paper we propose a novel method for incremental synchronization of software artifacts when these artifacts are generated using model driven generative development processes and environments. The proposed framework treats its subject transformation as a black-box. Therefore, other than assuming the transformations to possess some generic properties, the framework requires no detail knowledge of the transformation rules. Our proposed scheme has thus the advantage of saving the often toil-some effort of re-implementing such transformations in a new language or framework.

The following diagram illustrates the problem of model synchronization in a more precise manner.

$$\begin{array}{ccc} M & \xrightarrow{T_{Gen}} & N \\ {\scriptstyle \Delta_1}\downarrow & & \downarrow{\scriptstyle \Delta_2} \\ M' & \overset{T_{Gen}}{\cdots\!\!>} & N' \end{array} \qquad \Delta_1 \xrightarrow{Sync} \Delta_2$$

Let T_{Gen} be a collection of transformations that generate a target model N from source model M, and let Δ_1 be a change operation on model M that yields M'. Let's also assume that models M and N are synchronized, that is, they are consistent according to some relation, R. A simple approach to re-synchronizing M' and N is to re-apply all transformations available in T_{Gen} on M' to obtain N'. However, when M is large enough relative to Δ_1, and/or T_{Gen} is computationally expensive, this complete re-generation approach will not be responsive enough for modern interactive development environments. Furthermore, the complete regeneration of M' does not provide any means for reflecting the changes made to the target model, N', back to the source model, M'. As the diagram depicts, a mapping such as *Sync* could be used to generate from Δ_1 a sequence of change operations, Δ_2, that is applicable to N, and results in the same model as $N' = T_{Gen}(M')$ which is consistent with M' according to the same relation, R. We say that *Sync* is incremental, if the number of elements in N that Δ_2 modifies to obtain N' is minimal.

The architecture of our proposed solution is depicted in the block-diagram diagram of Figure 1. The crux of our idea is to store the mutual information for all inter-related models in one place, and refer to them by a unique identifier across all the heterogeneous models in a system. To achieve that, we devise a process called *Conceptualization*. This process involves identifying, abstracting, tagging and centralizing the data encapsulated within a software artifact into logical entities called *Concepts*. Related artifacts share mutual information that can be linked by concepts. In other words, two or more inter-related elements which reside in different artifacts can be made refer to the same

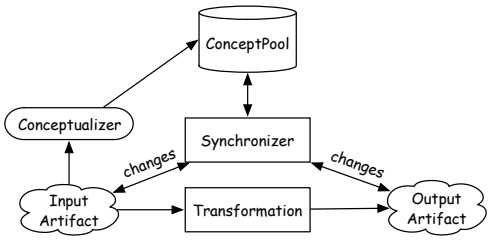

Figure 1: Architecture of the Synchronization Framework (arrows denote dataflow)

concept by referencing its unique identifier.

Concepts are stored in a centralized location referred to as *Concept Pool*. The framework provides facilities to efficiently trace a concept from any given position inside a model, to its corresponding entity in the concept pool and *vice versa*. The synchronizer unit listens for changes in the interrelated artifacts. When a change occurs, the system updates the values of the corresponding concepts in the concept pool. When the models need to be re-synchronized, the values in the affected concepts in the concept pool are propagated to the model elements that are indexed by the same unique identifier of the modified element. This propagation takes the form of value replacement, and can be carried out in linear time with respect to the number of elements involved. As the figure indicates, the synchronizer is able to propagate changes in both directions, notwithstanding the subject transformation's support for bi-directionality or lack thereof.

The rest of this paper is organized as follows. A review of the related work is presented in Section 2. Section 3 presents the preliminary information, establishes our used notation and terminology, and introduces our running example. The detail of the proposed methodology is presented in Section 4. Section 5 discusses the application of the proposed framework on Eclipse Web Tools Platform (WTP) [16] and reports the results of our evaluations. Finally, Section 6 presents possible avenues for future work, and concludes the paper.

2 Related Work

The major premise of our work is to avoid re-implementation of an existing transformation in another language or framework. This sets an immediate diverging point between our approach and that of other incremental synchronization frameworks, majority of which define some sort of specification mechanism, whereby an existing transformation has to be re-implemented. In contrast, we take the transformation's implementation as a black-box, and build the support for incremental synchronization around it, only assuming a few generic properties about the transformation.

A catalogue of various model synchronization schemes is offered in [3]. The paper focuses on formulating the external behavior of model transformations. A general classification of model transformation frameworks can be found in [7]. Two features that specifically concern the problem of model synchronization are bi-directionality and incrementality. The paper introduces several frameworks that support either or both of these features, most notable of which is the Object Management Group's Query View and Transformation (QVT) [5]. QVT proposes the *Relation* and the *Core* languages, both of which capable of denoting incremental and bi-direction transformations with different expressiveness and complementary levels of abstraction.

Triple Graph Grammar (TGG) is a graphical, declarative, incremental and bi-directional model transformation methodology based on graph transformation [14]. It has been advocated to be an effective foundation for tool integration [2]. Beanbag is an emerging framework which offers a language that supports intra-relations between models [18]. Our conceptualization scheme also supports intra-relations.

Bi-directional transformations and their application in model synchronization have been investigated by researchers in the programming languages community. Foster et al. proposed the Harmony framework based on the notion of *Lenses* for bi-directional synchronization [8]. They propose a language in which programs are inherently bi-directional. The Harmony framework has a state-based perspective on change. Contrariwise, Alanen and Porres have investigated syntactic merging, differentiation and union of structural models in [1] by adopting an operational and compositional view of changes.

One solution that works with existing trans-

formations is SyncATL, proposed by Xiong et al. in [17]. They have extended the bytecode of the ATL virtual machine [4], whereby supporting automated backward synchronization of models linked by an ATL transformation. For the forward synchronization, SyncATL relies on re-invoking the transformation and merging the results with the existing target. The proposed technique, however, does not address incremental synchronization at all, as the framework relies on re-executing the transformation for forward change propagation. The other drawback of this approach is its tight integration with a specific technology, i.e. ATL.

Another framework with the theme of building incremental synchronization around existing transformation engines is presented in [12]. This time the Tefkat transformation engine, which has logical programming flavor, is decorated with support for incrementality. In Tefkat, transformation rules are specified as logical predicates, and are performed using SLD resolution. Their synchronization framework avoids redundant computation in successive transformation of the same model by preserving the intermediate SLD trees. Another logic based framework which exploits answer set programming for change propagation is presented in [6].

In [15], Tratt articulates the spectra of challenges involved in model driven tool integration with model transformations being their centerpiece. The paper stresses the importance of the particularly challenging problem of change propagation and enumerates the reasons for inadequacy of solutions based unilaterally on commonly championed panaceas such as bidirectional transformations or traceability. He concludes that a comprehensive change propagation scheme, to be pragmatically effective, should function tractably over the most common patterns of transformations.

The problem of incremental model synchronization parallels the extensively investigated, yet in some degrees open, problem of view maintenance in databases. Two noteworthy approaches to this problem are presented in [9] and [11]. The view maintenance problem, along with the view update problem, have inspired software engineers to look at interrelated software artifacts as essentially different views of a common database. The bi-directionality of synchronization can thus be remedied by solutions transpired for the view update problem, and incremental synchronization becomes analogous to view maintenance. In spite of similarities, there are also key differences between maintaining database views and synchronization of software models that warrant a distinct research agenda dedicated to the latter. Briefly, database views are defined in few, precise view definition languages, upon which the database community has come to a unanimous consensus. In comparison, model transformation frameworks are rather diverse and immature. Furthermore, the relational database model denotes flat structures. In contrast, models comprise containment hierarchies which pose a semantic for deletion that can be peculiar to handle using purely relational techniques.

Finally, CLIME [13] and MView [10] are constraint based consistency management frameworks for incremental maintenance of software artifacts. These frameworks rely on the existence of a well-defined set of constraints for ensuring the consistency of the interlinked models, and are capable of incremental resolution of inconsistencies for such cases.

3 Notations and Definitions

Automatic transformations are used in many software development environments to generate new artifacts from the existing ones. Web Services is one such domain that incorporates various software specifications. For example, at the very core of Web Service models lies the web service implementations coded in a programming language such as Java, and a service description denoted in an XML based format called Web Services Description Language (WSDL). In this context, WSDL can be generated from Java source code via a transformation, as depicted in Figure 2. When an element of the source artifact, such as the name of the method in this example, is updated, or a method is added to the source code, the target file, e.g. WSDL in this case, has to be changed accordingly. We will use this simplified version of Java2WSDL transformation and its pertaining source and target artifacts of Figure 2 as

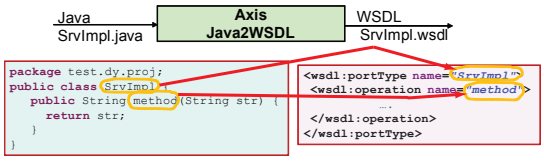

Figure 2: Generating WSDL from Java Classes

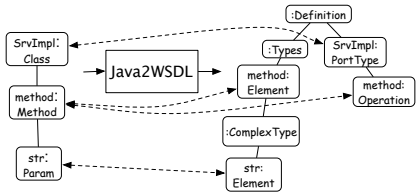

Figure 3: Abstract Models of Java and WSDL

our running example to demonstrate the various steps of our generic incremental model synchronization methodology.

In the running example, the source and target models are respectively in Java and WSDL formats. To cope with this diversity, our prototype utilizes Eclipse Modeling Framework (EMF) to programmatically access the models in a unified environment. The abstracted models of Java and WSDL artifacts of Figure 2 are depicted in Figure 3. As shown, the abstract model of Java code encloses only on the parts of the original artifact that are pertinent for the context of the transformation. As a result, the information regarding the bodies of the methods, such as variable definition or control flow, are discarded during abstraction.

For our purpose, model elements are instances of the types defined in a metamodel that describes their constituting model. They are information bearing entities that enclose primitive and complex information in their attributes. They can also have several containers, each containing other model elements. Model elements can also reference other model elements. The following definition captures the aforementioned characteristics.

Definition 1 *Model element m is a tuple $(\mathfrak{C}, A, \mathfrak{R}, T)$, in which $\mathfrak{C}$ is the list of containers, A is the list of attribute values, $\mathfrak{R}$ is the list of references, and T is the type of m. Each container $C_i \in \mathfrak{C}$ is a list that comprises model elements of certain types. T is a set of mappings to the elements of the metamodel that uniquely ascribe types to each fragment of m.*

We use the same letters as function names for projection of individual components of model elements, that is to say, $m = (\mathfrak{C}(m), A(m), \mathfrak{R}(m), T(m))$. It should be noted that the names, types and constraints of the attributes and containers are all specified in the metamodel and not in the model element itself. This separation allows for type (and also name) agnostic manipulation of model elements. Using this definition, the Java model of Figure 3 is denoted as:

$$\begin{aligned}&\mathsf{Class} = (\langle\langle \mathsf{Method1} \rangle\rangle, \langle \text{"SrvImpl"}, \text{"test.dy.proj"} \rangle, \varnothing, \mathsf{Class})\\ &\mathsf{Method1} = (\langle\langle \mathsf{Param1} \rangle\rangle, \langle \text{"method"}, \text{"String"} \rangle, \varnothing, \mathsf{Method})\\ &\mathsf{Param1} = (\varnothing, \langle \text{"param"}, \text{"String"} \rangle, \varnothing, \mathsf{Param})\end{aligned}$$

Definition 1 is recursive, since the members of containment lists are assumed to be model elements themselves. Similar to model elements, models can also be denoted using Definition 1 by simply assuming that they are the top level elements of their containment hierarchy.

We use a compositional representation for change in models, that is, every change operation is expressed as a composition of finer grained change operators. This requires identifying a number of *atomic* change operations as a starting point. We introduced three atomic change operations that take part in change composition. Atomic changes, and composite changes by extension, are defined as mathematical functions. The notations used for each of the atomic changes are listed as follows:

Table 1: Atomic change operations

Signature†	Description
$\blacktriangle : \mathcal{M} \times \mathcal{T} \times (\mathcal{C} \cup \mathbb{N}_0) \mapsto \mathcal{M}$	Insert Element
$\blacktriangledown : \mathcal{M} \times (\mathcal{C} \cup \mathbb{N}_0) \mapsto \mathcal{M}$	Delete Element
$\blacklozenge : \mathcal{M} \times (\mathcal{A} \cup \mathbb{N}_0) \times \Sigma^* \mapsto \mathcal{M}$	Update Attribute

† In the above signatures, $\mathcal{M}$ is the set of all model elements, $\mathcal{A}$ is the set of all attributes, $\mathcal{T}$ is the set of all types and $\mathcal{C}$ is the set of all containers. Σ^* is the set of all valid attribute values.

The semantics of composition is also similar to that of mathematical functions. For example, the following composite change operation adds a new parameter to the method of the Java example, and then, updates its name to *"param2"*:

$$\blacklozenge(\blacktriangle(\mathsf{Method1}, \mathsf{params}, \mathsf{Param}), \mathsf{name}, \text{"param2"})$$

4 Incremental Synchronization

4.1 Conceptualization

Concepts are defined as primitive abstract entities that semantically associate two or more information carrying elements across a pool of heterogeneous software models. Models consist of model elements, which enclose several attributes to represent information. A concept, however, can be even smaller than an attribute value; attribute values can be composed of multiple concepts. From this point of view, models provide organization, structure and semantics to an amalgamation of concepts by encapsulating them into various model elements of different types.

Furthermore, related artifacts share mutual information. Conceptualization assists the synchronization of this mutual information in two major ways. First, concepts provide a systematic way for tracing the propagation of transformed data piecemeal, inside and outside the boundaries of the artifact in which they are located. Second, concepts can establish fine-grained interdependencies between two or more inter-related artifacts; different elements in multiple artifacts can be forced to refer to the same concept by referencing its unique identifier. *Conceptualization* is the process of extracting, indexing, tagging and centralizing concepts. Concepts are stored in a database embedded in the development environment. This database is referred to as the *Concept Pool*.

For the case of the Java2WSDL example, the dependencies between the source and the target of the transformation are established using concepts, as illustrated in Figure 4. Propagating changes from one model to another in this scheme takes the form of updating the pertinent concepts in the concept pool followed by fetching the new values to each affected element, provided that there exists mechanisms whereby the system can pinpoint the related concepts in the concept pool for a given model element. For example, in Figure 4, if the method argument name *"str"* is changed in the Java model, the framework updates its related concept in the pool and notifies its dependent element in the WSDL side, i.e. WSDL message, to updates its *"name"* attribute with the new value. A modification made to the elements of the target side can also be propagated to the source side likewise.

4.2 Shadow Models

As mentioned earlier, it is essential to locate the concepts associated with each model element, and also respond to queries about elements sharing the same concepts. To enable the latter, an index of identifiers of related model elements for each concept is stored in its corresponding entry in the concept pool. We propose *Shadow Models*, that is, intermediate models intended to facilitate answering to queries about elements sharing the same concepts. Shadow models closely mimic the structure of the original models. They are, in fact, produced by exchanging the values of the identified concepts in the models by their unique identifiers. Shadow models make accessible the identified concepts in the concept pool; e.g a concept entry of an attribute value in the concept pool can be located by obtaining its concept ID from the exact same position of the attribute in the shadow model.

The algorithm for creating shadow models is listed below as Algorithm 1. In the algorithm, the value of the model elements that contain a recognized concept are replaced by the identifier of said concept. An important practical note when generating unique concept identifiers is to ensure that they are still valid identifiers with respect to the grammar of both the source and the target artifacts. For the case of Java, this means that they need to be constructed using the characters allowed in the Java grammar and the WSDL schema for class and method names and also WSDL identifiers. This requirement is to guarantee that the Shadow model is a valid artifact of the same type of the original one, and can be used as input to transformers.

The *de-Shadow* operation, listed in Algorithm 2, performs the opposite operation of the Shadow algorithm. That is, it scans through the shadow artifacts, extracts the patterns for concept IDs from the attributes of model elements (an attribute can contain more than one concept by means of concatenation), it fetches

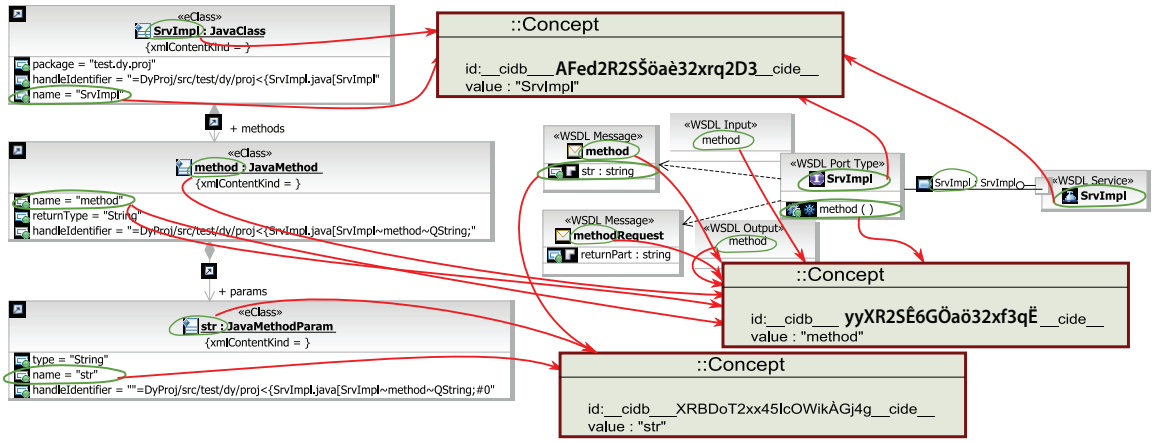

Figure 4: Conceptualization of Java and WSDL models

Algorithm 1 Shadow(M)

Require: Model $M = (\mathfrak{C}, A, \mathfrak{R}, T)$ and Concept-Pool $\mathcal{CP}$
Ensure: Shadow Model S

```
1: S ← Clone(M)
2: for all m ∈ S do
3:     for all a_i ∈ A(m) do
4:         ♦(m, i, CP.getConceptID(a_i))
5:     end for
6: end for
7: return S
```

the values of the found concept IDs from the concept pool, and replaces the IDs with the values.

Algorithm 2 de-Shadow(S)

Require: Shadow Model $S = (\mathfrak{C}, A, \mathfrak{R}, T)$ and Concept-Pool $\mathcal{CP}$
Ensure: de-Shadowed Model M

```
1: M ← Clone(S)
2: for all s contained in M do
3:     for all a_i ∈ A(s) do
4:         id[1..n] ← matchConceptID(a_i)
5:         v ← a_i
6:         for j ← 1 to n do
7:             cv ← CP.getConceptVal(id[j])
8:             v ← replace(v, id[j], cv)
9:         end for
10:        ♦(s, i, v)
11:    end for
12: end for
13: return M
```

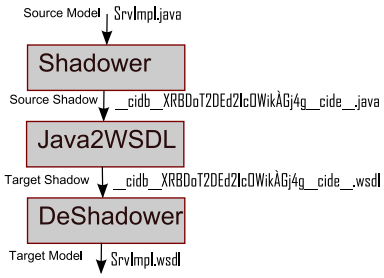

Figure 5: Shadow/Transform/de-Shadow Pipeline

We insist that shadow artifacts be valid documents of the same type of the source model, because we use them as inputs to the transformation to generate shadow models of the target domain, thereby achieving traceability through the common concepts appearing in both shadow models. Figure 5 shows the outline of the process of creating target models using shadow artifacts.

4.3 Synchronization Process

The idea of propagating model dependencies using shadow models relies on the presumption that, under the course of the transformation, the unique concept IDs in the shadow artifact are not subject to manipulations that make them unrecognizable in the resulting target shadow model. In other words, the essence of transformations for which this methodology is applicable is re-organization of concepts. Therefore, the attribute values of model elements should only be subject to a category of re-writing operations that do not dismantle the

concept IDs. For example, the transformations are allowed to concatenate two concept values, or add a prefix (or a suffix) to a concept. Operations such as shuffling the characters of a concept, cutting some of the letters or anything else that does not preserve the concept IDs are not directly permitted. This, however, is not a major limitation for two reasons. On the one hand, concepts are defined to be the most primitive and finest grained pieces of information in a model. With that perspective, a wide range of meaningful transformations are expected to simply re-organize these quanta of information into different encapsulating data types, rather than tearing them apart. On the other hand, it is possible to work around these limitations. In Section 6 we discuss the outlines of an extension to the framework that allows anomaly cases of string re-write operations.

As a result of Conceptualization and exploiting Shadow models, we can ensure that the source of a model transformation and its product, the target artifact, are entangled reciprocally using their mutual concepts. These concepts are stored in the central concept pool along with other concepts in the system, and each is individually and uniquely identifiable by a universal ID. When one of the attributes of any of the objects belonging to either the source or the target artifact is subjected to an update value modification, essentially the value of one or more concepts that constitute the value of that particular attribute are updated in the concept pool. The affected concepts are also tagged as modified and the time-stamp of the latest change is also recorded in the corresponding entry of those concepts in the concept pool. Two strategies are conceivable for propagating the values of updated concepts to the other artifacts that carry those concepts. The first strategy is to disseminate the changes to all the artifacts which incorporate the affected concepts. This needs maintaining a list of referencing artifacts for each concept entry in the concept pool. The second strategy for the synchronization to be carried on lazily, meaning that synchronization is invoked for each artifact, only when it is opened or it is focused by the user. The outline of the synchronization process that is composed of three Phases, is listed below.

```
Change Propagation Process
Phase A: Conceptualization
  Create Models from Artifacts
  Extract Model Concepts
  Store concepts in Concept Pool
  Create Shadow Models

Phase B: Artifact Generation
  Generate Artifact from source Shadows
  de-Shadow the generated shadow artifact

Phase C: Change propagation
 Upon Change in Source or Target:
  Update related concepts in Concept Pool
   For all modified concepts
    Get the locations of all referencing elements
   If synchronization strategy is immediate
      push the changes to all impacted elements
   else
      monitor accessing the affected models
      push the changes when they are opened
```

In the first phase of the process above, we set the stage for incremental model synchronization by abstracting the involved software artifacts as EMF compliant models, creating unique IDs for the concepts, and compiling shadow models. In the second phase of the process, we execute the artifact generations on the created shadow models, thereby obtaining the shadow model for the target artifacts. Subsequently, we run the de-Shadow algorithm to convert the target shadow to the desired target artifact. Finally, the third phase of the process applies the framework whenever a change operation in one model creates the need of re-synchronizing with the rest. The framework traces the modified elements to their concept entries in the concept pool, and updates the pertinent values there.

Figure 6 illustrates synchronization of the Java2WSDL example utilizing the Shadow/Transform/de-Shadow process. On the top of Figure 6, lies the Java source code, which is the input artifact of the Java2WSDL transformation. The *Shadow* operation, depicted as an arrow of the same name, encapsulates the following operations in the order given: First, abstraction of the Java code into the model format (Figure 3). Second, Conceptualization of the resulting model. Third, performing Algorithm 1 on the abstract model. Finally, serializing back the resulting shadow model into the Java code format. The result is the shadow code, which, as portrayed in the figure, is structurally identical with the original code in the segments that are relevant to the transformation, save the values of the

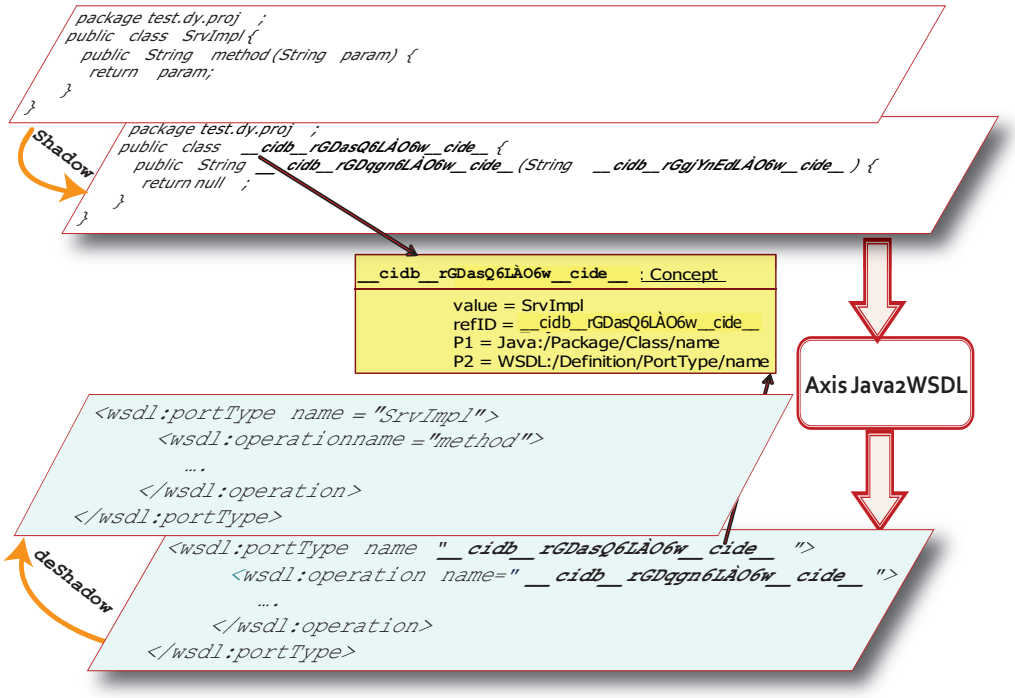

Figure 6: Synchronization by Shadow

concepts which are replaced by their unique identifiers. Specifically for this transformation, the bodies of the methods are ignored by the transformation, hence no manifestation in the abstract models, and consequently, neither in the shadow code.

The shadow code is used as the input to Java2WSDL, whereby yielding the shadow WSDL. To obtain the target model, Algorithm 2 is run over the target shadow, as illustrated in Figure 6 as the *de-Shadow* arrow. Any two related elements in either sides of the transformation, e.g. the name of the Java *class* and that of the WSDL *portType*, refer to the same entity in the concept pool and thus have the same value, inasmuch as the concept IDs obtained by looking at the same locations of these elements in their shadows are identical.

It should be noted that the update synchronization aided by concepts and shadow models is two-way, even if the original transformation is unidirectional. This is one of the major benefits of the proposed approach. The synchronization engine only exploits the artifacts, their shadows and the implicit correspondences denoted in the concept pool. No further invocation of the original artifact generator is required in this process. There is no distinction between the source and target of the transformation after it is applied initially. It is therefore possible that the target of the original transformation becomes the source of model synchronization, i.e. the updates that need to be propagated are made to the model which was the product of the transformation. The overall process of synchronization using concepts and shadow models is listed in the following. The system allows two possible synchronization strategies. The first strategy is classified as *direct*, that is, the framework pushes all changes to all impacted models as the changes happen. The second strategy is classified as *lazy*, that is, the models are left inconsistent, and the invalidated elements are updated from the concept pool, whenever they are accessed for the first time or when there is an explicit request for re-synchronization.

A special case that deserves further attention is when an updated attribute carries more than one concept. It can be represented by concatenation of conceptualized values and prefixes (or similarly suffixes). For example an attribute can comprise a fixed prefix, the first name, a hyphen–which is another invariant segment–

and the last name of a person. The first name and the last name are the conceptualized and variable segments of the attribute value, whereas the title and the hyphen are constant. When the value of such attribute is updated, the system should sort out the concepts that are modified and extract the new segmental values corresponding to those concepts. To detect the altered concepts, the attribute value is screened against its counterpart in the shadow model. The fixed segments appear in both sequences and are used for aligning concept IDs and their segmental values.

4.4 Propagation of Insertion

Unlike Update operations, Insertion and Deletion operations are structure altering changes. Propagation of insertion and deletion induced changes for an arbitrary model transformation can be multifaceted and complex. This complexity can fortunately be addressed by assuming that the transformation has certain properties, namely, *continuity* and *monotonicity*. The former is to guarantee that the transformation is not sensitive to any particular instance model in its domain, but rather it transforms them all uniformly. The latter requires the change operations to have similar effects on both sides. These two properties seem to be valid for a wide range of model transformations used in practice..

Model elements, according to Definition 1, have containers, which can be inter-dependent across multiple models. In other words, the dependency of element types can be viewed as dependency between the containers of those types. This interpretation of dependency links implicitly requires that insertion of an element to one container (only) result in addition of elements in its inter-dependent containers. We assume that, Insertion (and similarly Deletion) homogeneously results in Insertion (and respectively Deletion) type of changes in other interrelated containers. We refer to this property of transformations as *monotonicity*. If insertion of an element results in update or deletion type of changes in the target model, then the transformation is non-monotonic.

Furthermore, we assume that inducing an element into a container follows a uniform pattern of insertions that is independent to the number of elements inside said container. For example, addition of the third parameter to a method does not trigger a different pattern of changes in its inter-dependent containers, than does the addition of the second parameter for that matter. We call this property of transformations *continuity* and such transformations are called *continuous*.

Although these two assumptions may seem too restraining, in practice they are in compliance with the majority of artifact generators. In fact, the space of transformations that common relational frameworks such as QVT, TGG, Tefkat etc. are capable of expressing are also continuous and, for the most part, monotonic. The explicit definitions of the transformation rules that happen to violate either of these two assumptions have to be known. The proposed framework allows for extension points inside the concept pool for incorporating such rules, so as to achieve complete synchronization.

The proposed methodology for the propagation of Insertion involves deliberately injecting each container in the shadow model with a dummy placeholder element called a *μ-template*. These template-injected shadow models are thereafter used as input to the artifact generation process, and, as a result, the μ-templates will be transformed and instantiated as target artifacts.

More specifically, using μ-templates, the target artifact is generated with an additional hypothetical new element. When an actual insertion change to a container takes place, the added element replaces the available μ-template in the container. In other words, this μ-template is *consumed* into an actual element in the source artifact and a new unsubstantiated μ-template is inserted to accommodate future insertions.

In other words, in this process we *a priori* assume the possibility of having an additional element to be inserted in the future for each container, and reserve in advance an appropriate structure and space (i.e. the μ-template) for such elements in the container. When needed, we use these reserved places for adding a new element to the containers they reside in, by updating the values of their attributes and rendering them as visible.

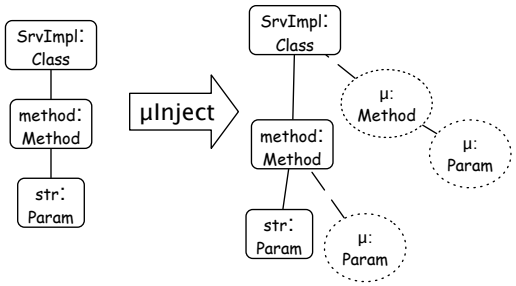

Figure 7: Injecting μ-templates

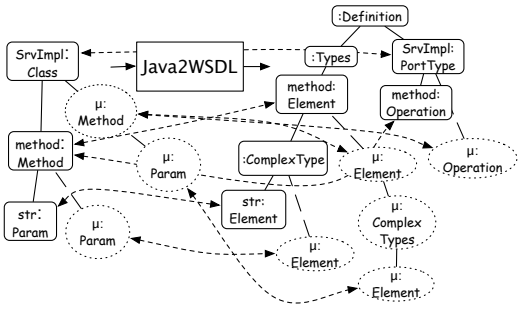

Figure 8: Transformation of μ-templates

Consuming the reserved space of μ-templates prohibits the opportunity for adding more elements to the container in the future; simply because each container only had one extra space. Therefore, to continue supporting insertion of new elements into the container, it is important to create a new reserved space before consuming the available μ-template. The second assumption enables us to provide a new μ-template by simply duplicating the old one and assigning new concepts to it and so on.

To better demonstrate how Insertion induced changes can be propagated by utilizing μ-templates, we proceed with the example scenario of synchronizing an insertion of a method argument to the Java side with its corresponding WSDL model. The process is illustrated in Figures 7, 8, 9 and 10. The first step of the process is injecting μ-templates into the containers of the source artifact. To disguise this amalgamation from the user and make the synchronization as transparent as possible, μ-templates are in fact injected into the Shadow models right after their creation. Shadow models, as discussed, closely mirror the structure of an artifact and are hidden from the user. Figure 7 shows (the shadow of) an abstracted Java code for a web service on the right hand side. On the left, the same Java model is shown after it is populated by μ-templates. As the figure illustrates using dashed lines, in every container in the model, an extra element is injected.

As stated previously, the shadow of the source artifact, instead of the artifact itself, is used to generate the target artifact. The same steps are essentially involved for synchronizing Insertion changes, with the addition of injecting the source shadow model with μ-templates. Figure 8 shows the shadow model of the Java abstraction populated with μ-templates, and its resulting WSDL model which includes μ-templates in several places. μ-templates are discerned from normal elements by having all the concepts they encapsulate marked as μ-template in the concept pool. In other words, any model element that hosts a μ-template concept is a μ-template, and therefore, shall not have a manifestation in the target model. This requires a slightly different de-Shadow algorithm to check and skip over elements that contain concepts flagged as μ-template.

When an element is inserted in one of the containers of the source artifact, the first step is to conceptualize the new element, i.e. extracting the concepts that appear in the new model element. This is realized by *consuming* the μ-template that had been provided for insertion of new elements in the container. The μ-template is turned into a normal element by essentially removing the μ-template flags from all its concepts' entries in the concept pool. When consumed, μ-template concepts' IDs remain intact. The synchronization framework, when encounters such concepts in other models, deduces that they belong to the consumed μ-template concepts. The container of the consumed μ-templates need to be populated by new μ-templates to allow for further insertion of model elements. Therefore a new μ-template is created and injected into the container. Figure 9 illustrates the steps involved during insertion of a new element dubbed as consuming a μ-template.

In order to synchronize the target model with the modified source model, the de-Shadower is reapplied on the target shadow model. As mentioned, the μ-template concepts were con-

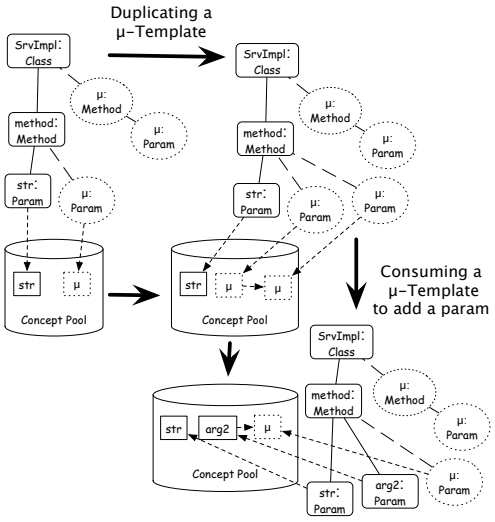

Figure 9: μ-template Consumption

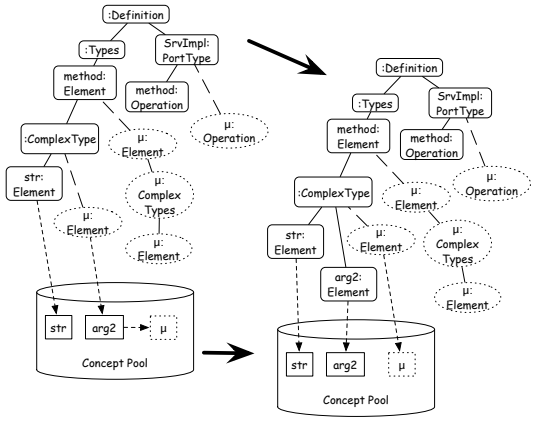

Figure 10: Propagating Insertion

verted to normal elements in the concept pool at consumption time, i.e. when the source model was subjected to insertion. Therefore, the de-Shadower, when reinvoked over the target shadow model, would no longer discard these elements and will render them in the output model, as demonstrated in Figure 10. Similarly to the source model, a new μ-template needs to be placed in the container to enable further insertion of elements in it. Unlike the source model, providing a new μ-element to the target model involves a few more steps than simply duplicating the former μ-element and adding brand new concepts to it. In particular, the dependency links between the new μ-template and the one that was just inserted in the source model have to be established by making them reference the same concepts. To achieve that, we need to find out the concept IDs that are assigned to the newly created μ-template in the source model, when the old μ-template was consumed. As usual, our medium for communicating such information is the concept pool. Therefore, this can be enabled by providing pointers in the entries of consumed μ-template concepts in the concept pool to the new concepts created for the new μ-template. For example, in Figure 9, when the μ-template is consumed and its value is updated to *"arg2"* in the concept pool, it points to the concept associated with the newly created μ-template in its container. The target side is only aware of the consumed μ-templates' IDs, since the new ones were not present in the model at the time of transformation. However, the de-Shadow algorithm follows these pointers for each concept to reach the new μ-template's concept IDs.

Using μ-templates, we have reduced the problem of propagating Insertion to an already solved problem of propagating Update.

4.5 Propagation of Deletion

Deletion is handled simply by hiding model elements whose concepts are tagged with the special flag *deleted* in the concept pool. For example, when the parameter of a method is deleted from the source code, all the concepts of its corresponding element in its abstract model are tagged as *deleted* in the concept pool. When synchronization is carried out on other models, the framework simply conceals the elements, any of whose concepts are flagged as *deleted* in the concept pool.

When considering containment, propagating deletion changes raises some ontological issues. More specifically, we need to recognize the existential causality relationships between the model elements in order to properly identify the elements that have to be purged as a result of a deletion change. The semantics of containment relationship provides useful guidelines for such

reasoning. Briefly, deletion of a containment results in purging all its contained elements and, consequently, their constituting concepts. When performing inter-model change propagation, all the elements whose any concepts are flagged as *deleted* will be flagged as *deleted* likewise.

5 Experiments and Evaluation

For the evaluation of the proposed framework, we have designed a prototype which we applied it for the incremental synchronization of models in the Eclipse Web Tools Platform (WTP). WTP encompasses several types of software artifacts each having different format and schema. Furthermore, it extensively uses transformations for converting these artifacts to one another. One such transformation, which we have used throughout this paper for presentation, and also for conducting our assessments, is Java2WSDL. Java code is a textual file that is parsed into an Abstract Syntax Tree (AST), and is compiled into a binary class file. In contrast, WSDL is an XML document that conforms to the WSDL schema, which is specified in the XML schema format. Moreover, because of its extensibility type definitions, which are XML schema elements, the type definition part of WSDL conforms to *XML Schema for XML Schema* (the meta-meta-model of XML).

Figure 11 compares the execution time of synchronization against that of re-transformation of increasingly larger Java classes, i.e. with more methods, and their resulting WSDL files. The Y (i.e. elapsed-time in seconds) axis is outlined in logarithmic scale. This graph also shows the framework's setup time, that is, the time spent to initialize the framework and create the shadow models. It is evident that the time cost of synchronization is almost independent of the models' sizes; contrary to the time required for regeneration, which acutely increases as the size of the input model grows. From the graphs in Figure 11, we observe that a Web Service system exposing close to ten thousand methods (a quite excessive figure for practical systems) is taking approximately 1000 seconds to regenerate all the models using all the available transformation rules (top line), while

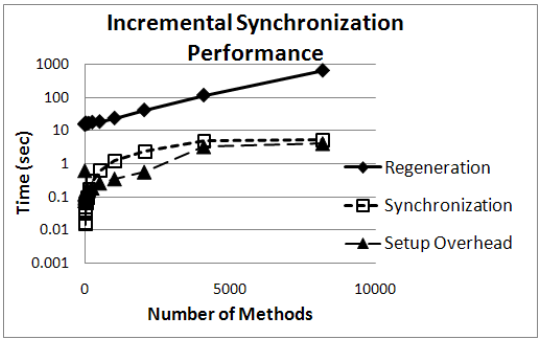

Figure 11: Performance Evaluation(Log Y-Axis)

the time to perform the initial setup and to incrementally synchronize the models is approximately 6 seconds and 8 seconds respectively.

The Java2WSDL transformation, in fact, ranges over multiple input files. This situation arises when one Java class references another class through methods' argument types or return types. In such cases, Java2WSDL also creates type definition for the referenced class in the WSDL file. To further assess the performance of our synchronization framework, we utilized this aspect of the transformation as an instance where the complexity of the subject transformation also progressively increases. Figure 12 shows the result of synchronization for hierarchies of multiple Java beans and their corresponding WSDL. The setup time is higher than the previous experiments, albeit still markedly faster than regeneration. This difference is predominantly due to the relatively high overhead of file I/O in Eclipse workspace, and the fact that the experiment with multiple beans naturally involves many more such operations. The results in this Figure indicate that for a system composed on 512 beans the complete regeneration takes approximately 300 seconds while the setup time for the incremental synchronization process takes approximately 150 seconds. Once the setup process is complete, then synchronization due to insertion and undate induced changes takes almost constant time of less than 10 seconds. Nevertheless, the setup process takes place only once at the beginning, so, in effect, it does not slow down the synchronization phase.

The extra space required by the shadow mod-

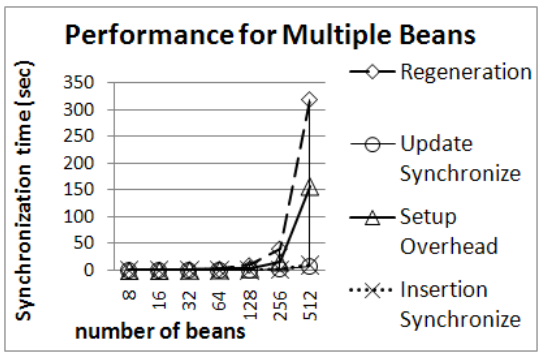

Figure 12: Performance for multiple beans

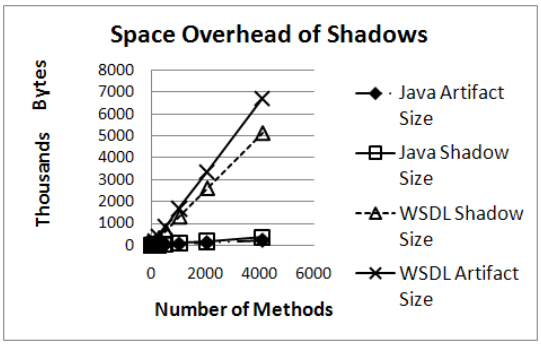

Figure 13: Shadow files space overhead

els is reported in Figure 13. As the figure depicts, the space overhead of shadow models and μ-templates tends to be on the same order of magnitude of the size of the models, hence it does not pose any limitations on the system.

6 Conclusion and Future work

In this paper, we propose a novel method for incremental synchronization of software artifacts. Our technique differs from the previous undertakings primarily in the fact that it uses the original artifact generators as black boxes. As such, other than some generic assumptions about the type of the transformations, it needs no detailed knowledge of the consistency rules. In contrast to the approaches based on incremental transformation engines, our proposed model synchronization framework neither needs denotation of transformations in a new language, nor it requires the transformations to be re-executed after they are used for the initial creation of the target models. The framework even when used in conjunction with unidirectional artifact generators, is capable of propagating updates in the opposite direction of that of the used artifact generator. The synchronization scheme results in models that comply with the original transformation.

The proposed approach is based on a process we refer to as *Conceptualization*. This process extracts the mutual information of two or more inter-related artifacts and stores them in a central concept pool. *Shadow Models* are used as input to the transformations for providing effective traceability between concepts and model elements in interlinked models. We utilize the technique to provide instant and incremental propagation of Update induced changes between models. To support incremental synchronization of Insertion induced changes, we also propose the notion of μ-templates, some localized place holders in the shadow models.

Treating transformations as black-boxes has the advantage of eliminating the cost associated with reverse engineering of consistency rules between software artifacts. However, the proposed solution, even though covering a wide spectrum of practical model transformations, is limited by the concept preservation assumption made for the case of update changes, and also by continuity and monotonicity assumptions made for insertion propagation. The idea of concept pool is nonetheless extensible. In addition to plain values, concepts can be allowed to assume embedded rules defined in an extension language for explicit denotation of concept dependencies inside the concept pool. For a given transformation, we partition its domain into segments that comply with the assumptions, as well as singularity areas that need special treatment. We can leverage such embedded rules to also automate the synchronization of the singularity areas in the transformation's domain along side the regular segments. We are currently integrating this extension into the framework.

Further improvements to the conceptualization process can be achieved through automated meta-model comprehension. We plan to utilize knowledge representation techniques such as formal concept analysis to enhance that stage of our framework, and to apply the framework for the re-synchronization of software models in Integrated Development

Environments such as RSA.

Acknowledgement
This work is supported by an IBM Center for Advanced Studies Fellowship.

References

[1] M. Alanen and I. Porres. Difference and union of models. *UML 2003 PROCEEDINGS*, pp. 2–17, 2003.

[2] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based tool integration with MOFLON. In *30th Int. Conf. on Software Engineering (ICSE'08)*, pp. 807–810, Leipzig, Germany, May 2008.

[3] M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In R. Lämmel and J. Visser, editors, *GTTSE'07*, LNCS. Springer, 2008.

[4] ATL. *Specification of the ATL Virtual Machine version 0.1*. LINA and INRIA, Nantes, France, 2005.

[5] MOF QVT final adopted specification, Nov 2005. OMG document `ptc/05-11-01`.

[6] A. Cicchetti, D. Di Ruscio, and R. Eramo. Towards propagation of changes by model approximations. In *EDOCW '06: Enterp. Dist. Object Computing Conf. Worksh.*, page 24, Washington DC, USA, 2006.

[7] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621 – 45, 2006/07/.

[8] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[9] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. pp. 328 –, San Jose, CA, USA, 1995.

[10] J. Grundy, J. Hosking, and W.B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engingeering (TSE)*, 24(11):960 – 81, 1998/11/.

[11] A. Gupta, I. S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. volume 22, pp. 157 – 166, Washington, DC, USA, 1993.

[12] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. volume 4199 LNCS, pp. 321 – 335, Genova, Italy, 2006.

[13] S.P. Reiss. Incremental maintenance of software artifacts. *IEEE Transactions on Software Engineering (TSE)*, 32(9):682 – 97, Sept. 2006.

[14] A. Schürr. Specification of graph translators with triple graph grammars. volume 903 of *LNCS*, pp. 151–163, Herrsching, Germany, June 1994.

[15] L. Tratt. Model transformations and tool integration. *Journal of Software and Systems Modeling*, 4(2):112–122, May 2005.

[16] Eclipse Webtools Platform Project `http://www.eclipse.org/webtools/`

[17] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi and H. Mei. Towards automatic model synchronization from model transformations. In *ASE'07: Proceedings 22nd conf. on Automated Software Engineering*, pp. 164–173, New York, 2007.

[18] Z. Hu, M. Takeichi, H. Song, H. Mei, Y. Xiong, H. Zhao. Beanbag: Operation-based synchronization with intra-relations. Technical Report GRACE-TR-2008-04, Tokyo, Japan.