A Language-based Approach to Model Synchronization in Software Engineering
（ソフトウェア工学におけるモデル同期に関する言語論的研究）

Yingfei Xiong
（熊　英飛）

Department of Mathematical Informatics
Graduate School of Information Science and Technology
University of Tokyo
（東京大学　情報理工学系研究科　数理情報学専攻）

Advisors:
Zhenjiang Hu, Professor, National Institute of Informatics
（国立情報学研究所　胡振江教授）
Masato Takeichi, Professor, University of Tokyo
（東京大学　武市正人教授）

September 2009

Acknowledgements

It was my greatest fortune to have Zhenjiang Hu and Masato Takeichi as my advisors. During my Ph.D. studies, Prof. Hu spent countless hours advising and helping me not only on my research but also on many aspects of my life. His attitude towards research and life influences me to be what I am today, and will continue to guide me in my future career. On the other hand, Prof. Takeichi's wisdom and perspective greatly helped me to keep my research in the right direction, and his kindness to students constantly inspired me forward. They both are great scholars for me to follow beyond the thesis. I would like to express my deepest gratitude to them at the very beginning of my thesis.

I would also like to thank my collaborators. Dongxi Liu taught me many basic techniques in research while contributed a lot to the early work of this thesis. Hui Song extended the research to a broader scope and made useful feedbacks to push the work forward. Haiyan Zhao provided invaluable insights during discussion and countless direct contributions to the writing work. Hong Mei built a perfect environment to make the collaboration between Peking University and the University of Tokyo possible and enjoyable. Hiroshi Hosobe brought to me knowledge and insights in the constraint satisfaction problem. Yijun Yu applied my existing research to a new area.

This thesis work has benefited a lot from discussion with people on conferences and workshops or through e-mails. Daniel Ruiz, Antonio Vallecillo, Jiayi Zhu and Xin Peng used my tool in their research and provided useful feedbacks. Alexander Egyed shared his data from industrial that greatly helped shape my research. Jian Zhang, Krzysztof Czarnecki, Michal Antkiewicz, Andy Schurr, Antonio Cicchetti, Davide Di Ruscio, Holger Giese, Stephan Hildebrandt, Zongyan Qiu, John Hosking and John Grundy helped me to understand important issues and related research during conference and email discussions.

Members of the Information Processing Laboratory in the University of Tokyo and members of the BiG group in the National Institute of Informatics helped me a lot during my Ph.D. study. Discussion during and outside seminars inspired new ideas. Entertainment and friendship made my life in a foreign country enjoyable. Special thanks give to Hao Wang who, as my first tutor, helped me survive in a foreign country.

Besides them, I would like to thank all my friends in the University of Tokyo, Peking University, University of Electronic Science and Technology of China, and many other places. Their delightful company and selfless support help me go through difficult times.

I could not forget to thank Fuqing Yang, Hong Mei, Yue Wu, Gang

Huang, Yanchun Sun, Tao Xie and Lu Zhang, who advised me during my undergraduate and pre-Ph.D. study, leading me into the world of computer science research.

Last but not least, I am very grateful to my parents and my girlfriend, for their support, encouragement and patience during these years.

Abstract

Software development often involves a set of heterogeneous artifacts, such as requirement documents, design models and implementation code. Maintaining these artifacts has been a notorious problem in software engineering. When we update part of an artifact, we need to propagate the update across all artifacts to make them consistent. Such synchronization of heterogeneous artifacts is known to be time-consuming and error-prone.

Recently, meta model technique emerges to give a unified representation of artifacts by capturing them as models, enabling a unified way for programs to access artifacts. Based on models, people develop tools to automate some synchronization tasks, and these model synchronization tools have shown their usefulness in software development. However, as synchronizing artifacts may be very complex, such tools are usually difficult to develop and maintain.

In this thesis we propose a language-based approach to facilitating the development of model synchronization tools. We design specification languages for model synchronization. The specification languages mainly describe the consistency relations between artifacts, with a small amount of additional information to confine the synchronization behavior. When users give high-level specification of synchronization in these languages, we generate a synchronizer from the specification, and the synchronizer automatically synchronizes models to keep them consistent. We also identify the requirements for model synchronization, which consists of three properties to ensure the correctness of synchronization. We prove that our generated synchronization procedures satisfy the requirements we proposed.

We design the specification languages from the languages that developers use to describe the consistency relation in practice. In this way developers need less effort to learn our languages and can reuse their existing programs. We identify two typical types of model synchronization and design different specification languages for the two types. The first type, off-site synchronization, is used to integrate different software development tools. In tool integration developers usually describe the consistency relation between models in model transformation languages. A program in such languages converts models from one format into another but cannot synchronize the models after the transformation. We adopt a unidirectional model transformation language as specification language and use a trace-based technique to derive a synchronizer from a transformation program. In cases where there are already two existing transformations to transform between two models forwardly and backwardly, we also design an algorithm to wrap them into a synchronizer that allows parallel updates on the two models.

The second type is about the synchronization within one tool. In such

cases developers usually describe the consistency relations over models in a logic-based specification language. We design Beanbag, a specification language whose syntax is similar to first-order logic but developers can customize the synchronization behavior by adjusting their programs. We also discuss the implementation issues of the Beanbag language.

All our languages have been implemented, and we have applied them to real-world cases. The result shows that our approach is useful in practice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background: Model Synchronization

Software maintenance is costive. Reports [Sut95, Ede93] show that annual software maintenance cost in USA has been estimated to be more than 70 billion US dollars. The cost is surprisingly high when we compare it to the total cost in software life cycle. A recent report [Erl00] shows that software maintenance has cost about 90% of the total cost. Methods and tools are needed to reduce the cost of software maintenance.

To reduce the cost, let us consider what activities are involved in software maintenance. The ISO/IEC 14764 standard [ISO06] defines four categories of software maintenance, as follows.

Corrective maintenance Reactive modification of a software product performed after delivery to correct discovered problems.

Adaptive maintenance Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.

Perfective maintenance Modification of a software product after delivery to improve performance or maintainability.

Preventive maintenance Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

Although the four categories are disjoint, there is one word across the descriptions of all categories: modification. Modifying the software product is the main activity we have to perform in software maintenance. If we could make software easy to change, we can reduce the cost of software maintenance.

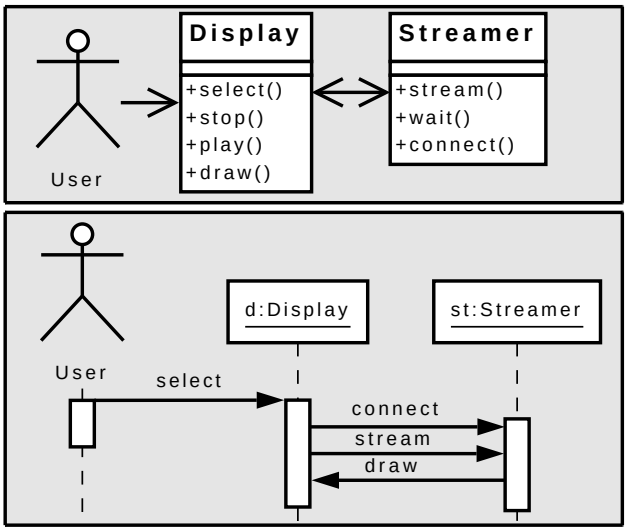

Figure 1.1: A consistent UML model

It seems unreasonable that software is difficult to change since software is known for its "softness", where compared to hardware, software is much cheaper to change. However, this becomes reasonable when we take into account the complex synchronization we are going to perform when changing a software artifact. In software development developers often create various kinds of artifacts at different stages of development. For example, at the requirement stage, developers create requirement documents, use case models and/or feature models; at the design stage, developers create Unified Modeling Language (UML) models, Entity-Relation (ER) models and/or pseudo code; at the implementation stage, developers create implement code; at the deployment stage, developers create configuration files. There artifacts are related to each other. Complex consistency relations exist not only between these artifacts but also between different components of one artifact. An example of consistency relation between two artifacts is the related UML models and implementation code. For each class in the UML diagram, there should be a corresponding class of the same name in the implementation code. An example of consistency relation between different components of one artifact is that, in a UML model, a class diagram and a sequence diagram is related, as shown in Figure 1.1. For each message in a sequence diagram (e.g. the `select` message), there should a method in the class diagram where the receiver object of the message should own the method in the class diagram (e.g. the `select` method in the class diagram).

When developers update some part of an artifact, some consistency relations on the artifacts may be violated, causing inconsistencies. In Figure 1.2

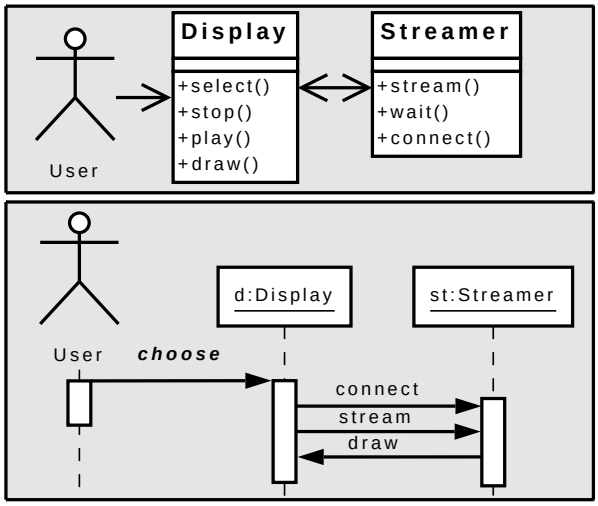

Figure 1.2: An inconsistent UML model

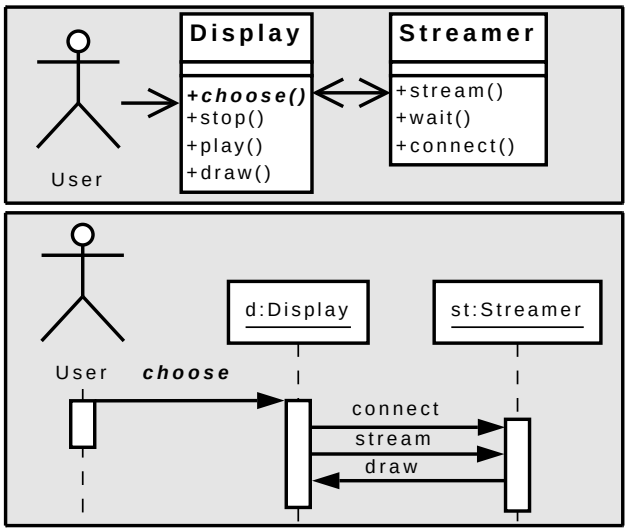

Figure 1.3: A synchronized UML model

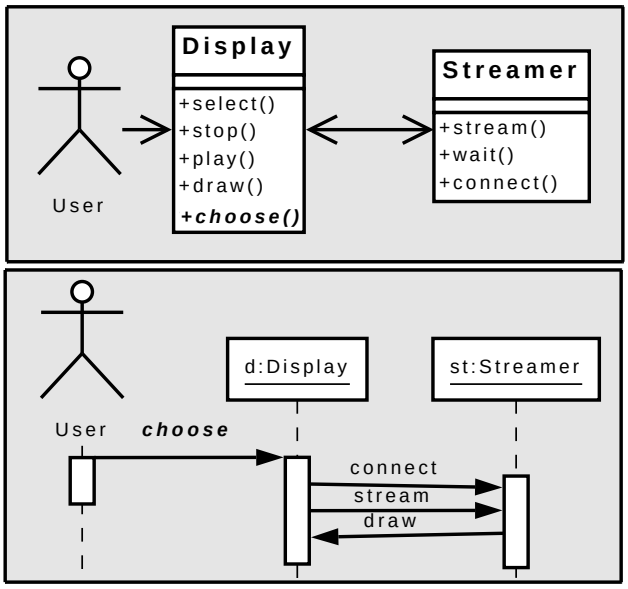

Figure 1.4: Another synchronized UML model

developers rename message `select` to `choose`, and the sequence diagram and the class diagram become inconsistent. To make all artifacts consistent, we must propagate the update to other artifacts as well as other parts of the artifact. Figure 1.3 shows one way to propagate the update: renaming the `select` method in the class diagram to `choose`. The process of enforcing consistency among a set of artifacts is called *synchronization* [AC08]. This kind of synchronization is also called *heterogeneous synchronization* to distinguish with synchronization between duplicated copies of one artifact; here we synchronize data with different structures.

Synchronization is not an easy task. First, we need to identify all locations that are related to the updated location. The number of affected locations may be large as many different relations interleave. For example, when one class is changed in the implementation code, not only the classes in the class diagram but also the object instances of the classes in the sequence diagram are affected. Furthermore, the change of a UML model may further affect other artifacts like ER models. Second, we need to decide how to update the identified locations. There may be various numbers of ways to achieve consistency. For example, besides the synchronization in Figure 1.3, we may also insert a new operation in the class diagram, as shown in Figure 1.4.

Since synchronization is one of the main tasks in changing software products, we can reduce the cost of software maintenance if we can automate the synchronization task. Although previous studies have recognized that not all

artifacts need to be automatically kept synchronized at all the time [Bal91] and not all inconsistencies are suitable to be automatically synchronized [Egy07], methods and tools that automate some part of the synchronization task can greatly ease the difficulty of changing software [ELF08]. Automated support can range from identifying possible locations to change (impact analysis) [Arn96], listing change actions for users to choose [NEF03], and fully automated certain types of synchronization task [GHM98].

This thesis focuses on the last type of synchronization support. To fully automate some types of synchronization, we need to consider the following three questions.

- How do we access software artifacts during synchronization?
- What do we synchronize?
- How do we synchronize?

1.1.1 Models and Meta Models

Let us start with the first question. To develop a synchronization tool, we need to read and write software artifacts. As discussed before, artifacts are stored in different formats. A modeling application may store models in a binary format while programs are stored in text format. The formats of the artifacts may be a secret to software engineering companies, and synchronization tool developers may not know the formats of the artifacts. Even if the developers know the format, they have to deal with many trivial details like data encoding (which may be different at different machines), reorganizing sequential data into structural, and etc. Developing such an artifact-accessing component is time-consuming and error-prone. Furthermore, such a component is difficult to maintain (when the format of the artifact changes, the access component needs to change) and to reuse (a component for one application cannot be used in another application).

To overcome the problems, people develop standard for data representation and access. One of the most well-known standards is Extensible Markup Language (XML) [YBP+04] defined by World Wide Web Consortium. This standard defines a markup language that represents tree-structured data in a standard way while hiding the trivial details in data access. Standard API for reading and writing XML files have been defined and is available in many languages. In addition, XML Schema (XSD) [FW04] is provided to define the structural of an XML file, allowing us to define meta tools for XML files. If a development application creates or converts its artifacts in XML format, we can easily read it in a structural way using the standard API of XML.

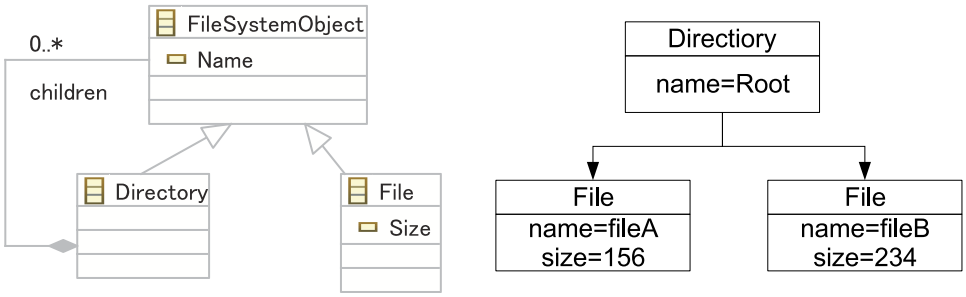

Figure 1.5: A file system model (right) and its meta model (left)

Furthermore, such application is easy to maintain and reuse, because the XML format usually changes less frequently than the internal format and programs accessing one XML file can be reused to access other XML files of the same schema.

In the software engineering area, Object Management Group defines a similar standard, MetaObject Facility (MOF) [OMG02], to facilitate access to software artifacts. MOF standard captures software artifacts as models and provide meta models to define the structure of the models. Different from XML that represents data as trees, MOF represents data as objects, which have a natural correspondence to the object-oriented paradigm in software development. In addition, Object Management Group also defines another standard, XML Metadata Interchange (XMI) [Obj07] for storing models as XML files, enabling a standard format for persisting MOF models.

The left of Figure 1.5 shows an example MOF meta model representing a file system. In this meta model there are three classes. `Directory` class represents the directories in the file system and `File` class represents the files in the directories. Both class are inherited from `FileSystemObject`. `FileSystemObject` has an attribute, `Name`, indicating both directories and files have their names. `File` class has `Size` attribute indicting the size of a file. In addition, between `Directory` and `FileSystemObject` there is an aggregation relation specifying every `Dictionary` object contains zero or more `FileSystemObject`s. Figure 1.6 shows a simplified XMI representation of this model. In this representation, classes are represented as nodes named "`eClassifiers`" and attributes are represented as nodes named "`eStructuralFeatures`". An Inheritance relation is represented by an attribute, `eSuperTypes`, and an aggregation relation is represented by a node of `eStructuralFeatures` where XPath [CD+99] is used to refer to other classes.

From the examples we can see that the MOF models share the same

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    name="File System">
  <eClassifiers name="Directory"
    eSuperTypes="#//FileSystemObject">
    <eStructuralFeatures upperBound="-1" eType="#//File"
        containment="true"/>
  </eClassifiers>
  <eClassifiers name="File"
    eSuperTypes="#//FileSystemObject">
    <eStructuralFeatures name="Size" eType="Int"/>
  </eClassifiers>
  <eClassifiers name="FileSystemObject">
    <eStructuralFeatures name="Name" eType="String"/>
  </eClassifiers>
</ecore:EPackage>
```

Figure 1.6: File system meta model model in a simplified XMI format

concepts as usual object-oriented languages. Software engineering tool developers can easily convert artifacts from their internal presentation to MOF presentation. Because of this similarity, many software engineering tools have provided the support for MOF models. Besides storing models as XMI files, MOF standard also defines a set of APIs for users to access models and meta models. Because the APIs provide powerful manipulating ability, more and more applications are directly built on models [BBM03, BDE+05]. When we synchronize artifacts in these applications, we can directly use the MOF API to access models, without dealing with trivial details. We call the synchronization of models as *model synchronization*.

1.1.2 On-Site Synchronization and Off-Site Synchronization

Let us proceed to the second question: what we synchronize. In general, synchronization can be classified into two categories according to the applications we synchronize [FGK+05].

In the first category, we consider integrating two applications that are developed without synchronization in mind. For example, a UML modeling tool and an ER modeling tool which are independently developed by two companies. When we want to integrate the two tools into one workbench, we need to keep the related data in the two applications consistent. Figure 1.7 shows the structure of this kind of synchronization. Two applications export

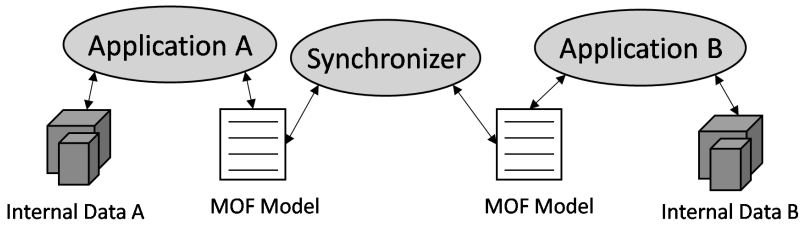

Figure 1.7: Off-site synchronization

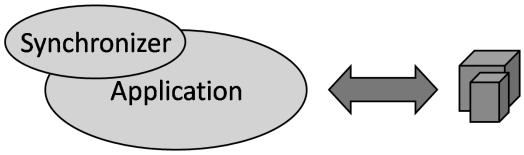

Figure 1.8: On-site synchronization

their internal data in some intermediate format, e.g., MOF models. Then a synchronizer modifies the two intermediate data to make them consistent. After synchronization, the two applications import the intermediate to update their internal data. Synchronization in this category is often performed when a user explicitly invokes a "synchronize" command, or automatically when application saves the models. Because the synchronization happens outside the applications, we call it *off-site* synchronization.

In the second category we consider the synchronization within one application. Models in Many software development tools have complex consistency relation between its components. For example, a UML modeling tool has to deal with related UML diagrams. Such kind of applications often provides dynamic synchronization support. When part of the model is updated by a user, the application dynamically updates other related parts of the model to make the model always consistent. One example application is IBM Rational Software Architect (RSA) [BDE+05]. When users update a model in the same way as Figure 1.2, IBM RSA will dynamically propagate the update to the class diagram as Figure 1.3. In this way the class diagram and the sequence diagram are always kept consistent. We call it *on-site* synchronization because it happens "on site" within one application. Different from off-site synchronization, on-site synchronization is considered by the application from requirement stage and the synchronizer is tightly integrated into the application. Figure 1.8 shows the structure of on-site synchronization. A synchronizer is integrated into an application and manipulates the internal data of the application directly.

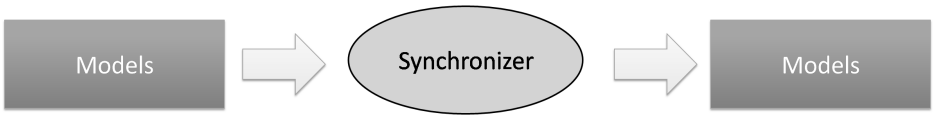

Figure 1.9: State-based synchronization

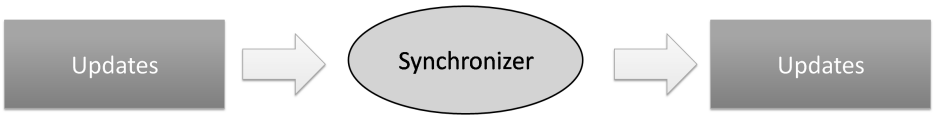

Figure 1.10: Operation-based synchronization

1.1.3 State-based Synchronizers and Operation-based Synchronizers

The third question concerns how we synchronize models. Based on the types of information used in synchronizers, we can classify synchronizers into two categories, *state-based* and *operation-based* [FGK+05].

A state-based synchronizer takes models as input and produce new consistent models as output, as shown in Figure 1.9. Because the synchronizer synchronizes models only depending on the current states of the models, we call this kind of synchronizers state-based.

An operation-based synchronizer takes updates that users perform on the models and produces new updates that are expected to be applied on the models to make model consistent, as shown in Figure 1.10. Because the synchronizer synchronizes models based on the update operations, we call this kind of synchronizers operation-based.

The distinction between state-based and operation-based synchronizers is not black and white. For example, we can perform state-based synchronization using an operation-based synchronizer by comparing the models of different versions to obtain (hypothetical) update operations. On the other hand, we can perform operation-based synchronization from a state-based synchronization by first applying the update operations to the models and then comparing models to obtain the result operations. In addition, there exist various hybrids approaches between the two. For example, in practice operation-based synchronizers often cannot perform synchronization solely based on the update operations but need to know the state of the models as well. This is achieved either by taking the models as input [XLH+07], or memorizing the needed information of the models as part of the synchronizer [XZH+08]. Another example is that a state-based synchronization approach, Bi-X [LNH+07], requires users to mark the locations that have been changed.

State-based synchronizers are more loosely coupled with applications, in the sense that applications neither have to record its update operations, nor have to apply the produced updates to the application data. Less integration work is required for applications. On the other hand, operation-based synchronizers often produce better result because they have more information available at synchronization time. As a result, state-based synchronizers are more suitable for off-site synchronization while operation-base synchronizers are more suitable for on-site synchronization. However, such choice is not absolute, either. There are approaches using operation-based synchronizers [LNH+07, LHT07] to perform off-site synchronization and approaches suing state-based synchronizers [NEF03, FGH+94, SMSJ03] to perform on-site synchronization.

1.1.4 Problems in Practice

As mentioned above, in practical there are available tools and development environments that provide automated synchronization of models. In the implementation of these tools, there are mainly two types of approaches used to implement the synchronizer based on whether the synchronization is off-site or on-site.

For off-site synchronization, many existing tools use model transformation [SK03] to synchronize models. The basic model transformation is batch/stateless transformation [Tra08], which is a program that convert model from one format into another format. In this way we can convert models created by one application into models created by another application. However, this disallows us to update the target model because the updates will be lost if we transform again. To solve this problem, Laurence Tratt [Tra08] proposes change-propagating transformation, where the transformation only updates the target model but not completely overwrites the target model. Using a pair of change-propagating transformations, a forward transformation from source to target and a backward transformation from target to source, we can keep two models synchronized by performing a transformation when users update one of the two models.

For on-site synchronization, existing tools mainly use the rule-based approaches [GHM98, FGH+94, SMSJ03]. In these approaches, programmers write a set of rules in the "condition-action" form, each rule states that when the condition is satisfied, the program should take the action. For example, an operation-based rule for the UML model in Figure 1.1 may be "when users rename a message in a sequence diagram, rename the related method in the class diagram". A state-based rule may be "when the receiver of a message does not have a method of the same name as the message, insert

a new method in the class of the receiver object".

These approaches work to support many existing software development tools in practice. However, both types of approaches require developers to develop the synchronizers by writing separate programs: the model transformation approach requires a forward transformation program and a backward transformation program, and the rule-based approach requires a set of separate rules. Such separation of programs has serious problems in practice. Here we mention three main problems.

- First, separation of programs increases the development cost. A large portion of the separate programs are related and can be deduced from each other, but we have to implement the related part in each program. Consider a simple synchronization where we keep two models equal. From the specification we know the reasonable behavior is just to overwrite the unmodified one with the modified one, but we have to implement two programs using the model transformation approach, one overwrites model in the forward direction and one overwrites model in the backward direction. The problem is even more serious in the rule-based approaches. As many kinds of update may cause a related part to become inconsistent, we usually need a large number of rules, and code is duplicated many times among these rules.

- Second, separate programs are difficult to maintain. As every separate program embodies part of the model structure, when the structure of the models is changed, every program is needed to be updated. Omission of identifying some programs and incorrect updating of programs are both sources of bugs.

- Third, it is difficult to verify the consistency of the programs. For example, a forward transformation program from a UML model to an ER model may convert an association between classes into a table, while a backward transformation program may (erroneously) convert every table into a UML class. Detecting the inconsistencies among programs requires intricate behavior reasoning, and the failure of detecting and repairing all inconsistencies leads to low quality software product.

Because of these problems, developing a synchronizer is still very difficult in practice. In fact, we have found that many widely-used tools have failed to implement a correct synchronization behavior. For example, IBM RSA has implemented the synchronization between a class diagram and a sequence diagram as shown from Figure 1.1 to Figure 1.2, but it has failed to capture other updates that also cause an inconsistency, such as changing the receiver

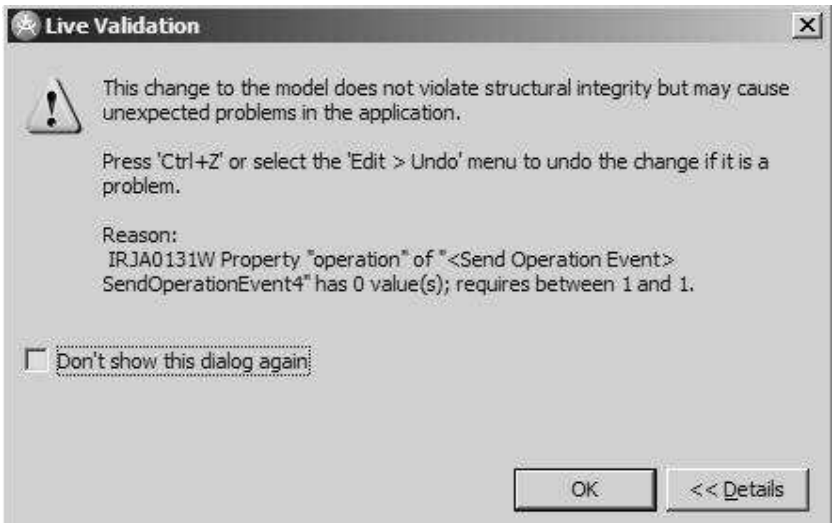

Figure 1.11: Incorrect synchronization results in error message in IBM RSA

of a message or changing the type of a receiver object. If a user performs these actions in IBM RSA, the system will prompt an error message, as shown in Figure 1.11.

1.2 Contributions of this Thesis

In this thesis we propose a language-based approach to the problem of program separation in model synchronization. Instead of writing a set of separate programs, in our approach developers write only one program. This program is mainly a specification of the consistency relations between artifacts, while containing a little additional information to define the synchronization behavior. After users have captured the consistency relation in the program, we automatically derive a synchronizer from the program. In this way we solve the problems from the separation of programs. First, the duplicated code piece is only written once in the program, reducing the development cost. Second, when the structure of the model changes, we only need to change one program. Third, the consistency of synchronization behavior can be ensured by the automatic derivation.

One important issue in this approach is what language we provide to write the synchronization program. Our choice is to design languages that are the same as or similar to the languages that developers use to describe the consistency relation in practice. In off-site synchronization, we use model transformation languages. In on-site synchronization, we design a language

similar to logic expressions. This choice has two advantages. First, developers can reuse the existing programs in our system without much effort. Second, the programming paradigm of the language is familiar to developers. It is usually easier for developers to learn a language in a familiar paradigm than a language in an unfamiliar paradigm. For example, it is usually easy for a C++ programmer to learn Java but difficult for a C++ programmer to learn Haskell. As developers have already used similar language to define consistency relations, they can learn our languages more easily.

The detailed contributions of this thesis are introduced in the following three sub sections.

1.2.1 Formalization of Model Synchronization

It is meaningless to talk about a solution without clarifying the requirement. One contribution of this thesis is that we formally define the requirement for model synchronization. Our requirement mainly consists of three properties: CONSISTENCY, PRESERVATION and STABILITY[XLH+07, XSHT09]. These properties form together to define the correctness of synchronization in general. The three properties are motivated from previous studied on updating semantics of database views [BS81] and the well-definedness of bidirectional transformation [FGM+07, Ste07], we significantly generalize the previous results for model synchronization.

Our requirement applies to both on-site synchronization and off-site synchronization and connects operation-based synchronizer and state-based synchronizer. We build a general model that treats the relation between models and relation between components within one model in the same way and thus capture both on-site synchronization and off-site synchronization. Our definition starts from operation-based synchronizer [XZH+08], and then we show how to lift the definition to state-based synchronization using model difference approaches [XSHT09].

As we consider operation-based synchronizers, we need to represent updates on models. Existing update representations [AP03, CRP07] are directly defined on models. Because the definition of models is complex, these update definitions are also very complex. Such a complex definition is easy for developers to use, but is not suitable for research studies as the researcher has to consider too many concepts together.

In this thesis we propose a lightweight representation for models and updates on models [XZH+08]. This representation is based on dictionaries that map keys to values. We show that dictionaries can represent all core concepts of models as well as updates on models. Because the definitions of dictionary-based data and update are very small (each consisting of only five

BNF rules), this representation is suitable for researchers to study. Also the research result on dictionaries can be directly applied on models using our conversion algorithm between models and dictionary-based data.

1.2.2 Support for Off-Site Synchronization

In off-site synchronization developers use model transformations, and we notice two facts in the current use of model transformations. The first fact is that there exist many batch transformation programs that convert a model from one format into another, but the converted model may both be modified after the transformation. In other words, we need to synchronize two models but we only have a forward, batch transformation that cannot perform synchronization. The second fact is that some tools provide model synchronization using two change-propagating transformations, one from a source model to a target model and one from the target model to the source model. Such two transformations are possible to be generated from one language using existing bidirectional model transformation languages [Obj08, SK08a], but they only propagate updates from one model to the other at a time and cannot help when the two models are simultaneously updated by users.

To provide synchronization support for first case, we assign synchronization semantics to one of the widely-used transformation language, Atlas Transformation Language (ATL) [JK06], and derive a synchronization procedure from an ATL program [XLH+07]. Using our approach, developers only have to provide a forward transformation that converts a source model into a target model, and they automatically get a synchronizer that keeps the two models consistent. Our implementation of this work, SyncATL, has been published and has been used by other researchers [YKW+08].

In this work we propose a trace-based technique to produce a change-propagation backward transformation from a forward transformation. When executing the forward transformation, we record an executable trace. This trace not only records how the values in the target model corresponds to the values in the source model, but also can reflect the update on the target model back into the source model. After we have the two transformations, we propose a novel algorithm to wrap the two transformations into a synchronizer.

For the second case, we improve the wrap algorithm so that it wraps two change-propagating transformations into a synchronizer for parallel updates [XSHT09]. In this way developers can reuse their transformation programs to support parallel updates without additional work. One feature of the improved algorithm is that we can customize how it handles conflicts. Using this feature, we use this algorithm to develop a runtime management frame-

work, where a high-level management interface and a running system are synchronized [SXH+08].

1.2.3 Support for On-Site Synchronization

For on-site synchronization, we notice that in practice many developers first capture the consistency relation over models in a logic-based specification language, such as object constraint language (OCL) [Obj06], and then implement "condition-action" rules to synchronize models to satisfy the consistency relation. After synchronization, the system invokes the specification program to check whether the consistency relation is satisfied, and reports an error when the relation is not satisfied. In this way even if the synchronizer is not correctly implemented, the system can capture the errors at runtime. One example of such systems is IBM RSA as we have seen its error message in Figure 1.11.

To make use of existing relation specification programs and the developer knowledge, we design a synchronization language, Beanbag [XHZ+09], that has a similar syntax to OCL. Beanbag defines the consistency relation in the same way as OCL, but every relation in Beanbag also has a fixing semantics describing when some parts of the data are changed by users, how to change the other parts to ensure consistency.

It is worth noting that in most cases there are multiple synchronization behaviors corresponding to one consistency relation. One example of this has been shown in Figures 1.3 and 1.4. This is captured in Beanbag by allowing more than one way to construct one relation, where a different way indicates a different fixing behavior.

We also discuss issues in the implementation of Beanbag. We show how functional programming techniques and aspect-oriented programming techniques can help us to derive clean and elegant implementations. In addition, we describe an application that embodies a typical software architecture of integrating a Beanbag synchronizer into an application.

1.3 Related Work

In this section we discuss the work related to the basic idea of the thesis. For work related to a specific chapter, we will discuss it in the chapter.

1.3.1 Bidirectional Transformation

The basic idea of this thesis originates from bidirectional transformation research [CFH+09]. Bidirectional transformation research focuses on the problem of synchronizing two heterogeneous artifacts. A basic approach to synchronize two artifacts is to write two transformation programs to transform between the two artifacts. However, this approach suffers from the problem of program separation. Researchers in bidirectional transformation community propose bidirectional programming languages, where a program in a bidirectional language represents two transformations, a forward transformation transforming source to target and a backward transformation transforming target to source. Based on the relations between the two artifacts, we can classify bidirectional languages into two categories.

Asymmetric Languages The first category of languages synchronizes two artifacts where one is an abstraction of the other. This category of bidirectional languages is mainly studied by the programming language community [BFP+08, FGM+07, LHT07, LNH+07, BMS08, MHN+07]. Given two domains, C, a domain of concrete artifacts and A, a domain of abstract artifacts, a bidirectional program in this category generates two functions: a stateless forward transformation $f : C \to A$ and a change-propagating backward transformation $g : C \times A \to C$. Since the abstract artifact can always be produced from the concrete artifact, the forward transformation does not have to be a change-propagating function but we can propagate changes between the two artifacts.

Symmetric Languages The second category synchronizes two artifacts of more flexible relations, and this category is mainly studied by the software engineering community [Mee98, Obj08, SK08a, Ste07, KH06]. Given two domains, M and N, a bidirectional program in this category consists of three components: a consistency relation $R \subseteq M \times N$, a forward transformation $f : M \times N \to N$ and a backward transformation $M \times N \to M$. Using the two transformations, we can propagate updates between the two artifacts. We may also create an artifact from the other side if the empty artifact is contained in M and N.

Bidirectional languages have been applied to many areas to synchronize different artifacts. However, as the two transformations always update one artifact according to the other, bidirectional transformations do not allow parallel updates on the two artifacts. In addition, as bidirectional languages are mainly designed as state-based synchronizers for off-site synchronization,

they cannot be easily applied to on-site synchronization (more detailed discussion can be found in [XZH+08]). This thesis is motivated by bidirectional transformation research and successfully extends bidirectional transformation in both on-site synchronization and off-site synchronization.

1.3.2 View-Updating Problem in Database

In database area, one classic problem is that when we get a view from a data source through some query, how to reflect the update back into the source. This problem has been studied by researchers since 70s in the last century. Some bidirectional transformation approaches [FGM+07] are also inspired by studies of the view-updating problem.

Compare to the studies in other area, studies on view-updating is more intensive. The studies focus on relation data and database query languages like SQL and XQuery, and the problem here is mainly how to put back the update on the view to the source. Researchers propose theories [BS81, DB82], algorithms [Mas84, Kel85] and design tools [MT86] for this problem.

One interesting issue is the "semantic ambiguity" problem: given a specific query, there is often more than one way to put back the update. Researchers need to find a method to uniquely determine one from all possible ways of putting-back. Bancihon and Spyratos [BS81] introduces the complement of a view, which includes missing information when the view is queried from the source. Given a fixed complement, there is only one way to put updates back into the source. Masunaga [Mas84] defines an algorithm to translate the updates on view to those on source, and the algorithm queries users when there is more than one way to put back the data. Keller et al. [Kel85] and Medeiros et al. [MT86] lift the user interaction to design time, and their tools interactively construct an update translator from users.

Our approach is partially based on these studies but extends these studies to another form of data and more general specification languages. In particular, we are facing the same semantic ambiguity problem, and our solution is to extend the original specification language so that the relation with ambiguity can be constructed in multiple ways, where each way corresponds to a unique synchronization behavior.

1.3.3 Constraint Satisfaction Problem

One traditional work on consistency management is the constraint satisfaction problem (CSP) [Tsa93]. In general, approaches to CSP try to find a set of values that satisfy a given set of constraints (consistency relation). This problem is known to be computationally expensive because we usually have

to search the whole state space to find a solution. Approaches to this problem are mainly focus on scalability: try to cope with larger data set in an acceptable time.

Our approach is similar to CSP as we also seek for a set of values satisfying a consistency relation. Nevertheless, there are two main differences. First, we focus on synchronization. We start from the previous models and updates and try to satisfy a set of synchronization properties. Second, our approach is more light-weight in the sense that we do not try to find the values solely from the relation but require users to describe the synchronization behavior in the synchronization program. As a result, we do not suffer from the scalability problem.

One branch of CSP problem concerns when a satisfied constraint is violated by some variable changes, how to reestablish the constraint. This kind of CSP problem is often related to graphical user interface [Bor81] where, for example, when users move a vertex of a square, we need to move the lines and other vertexes to ensure it to be a square. Typical approaches of this type include the constraint hierarchy theory [BFBW92] for specifying the synchronization behavior and several algorithms, such as DeltaBlue [SMFBB93] and Cassowary [BBS01], to satisfy a constraint hierarchy.

These approaches are particularly related to our approach because we also concern how to reestablish a consistency relation when it is violated by user updates. However, the data that we are dealing with are different from the data in these approaches. These approaches focus on arithmetic primitive variables and conjunction expressions, while we are dealing with MOF models that are much more diverse. MOF models are inherently graphs, while an object in model is structured and may refer to other objects. Relations on MOF models not only include conjunctions, but also may contain disjunction, universal quantifiers, or existential quantifiers. We need new ways to specify synchronization behavior under these relations and need new algorithms to propagate updates on these complex data structures.

1.3.4 Inconsistency Management

The rule-based approaches are mainly discussed in the inconsistency management area. These approaches differ in how they define the "condition" part of the "condition-action" rules. Some approaches define conditions from logic perspective. These approaches range from more theoretical, first-order logic [FGH+94] and description logic [SMSJ03], to more practical, OCL expressions [SSZH07]. In general, these approaches require developers to write the "condition" part of the "condition$\mapsto$action" rule in a logic expression and write the "action" part in an imperative language. The system checks

the condition of each rule, and executes the action when the condition is satisfied. Some approaches define conditions from the updates. Grundy et al. [GHM98] propose a general framework for managing inconsistency in multiple-view software. In their framework, developers write programs to check what types of updates have been performed by users and take necessary actions to propagate the updates.

Some approaches seek for automated means to generate a synchronizer from logical expressions. Typical approaches include the white-box analysis of first-order logic [NEF03] and the black-box analysis of the consistency rules [ELF08]. Compared to ours, these approaches generate a synchronizer purely from a consistency relation, but the synchronizer needs to interact with the user in synchronization, by asking the user to specify some locations to fix, choose one among a set of actions or fill some missed parameters. Compared to them, synchronizers generated by our approach execute in a fully automatic way without user interaction. Nevertheless, our approach is not aimed to replace the previous ones. Both types of synchronizers are important in model synchronization, because while in some cases the models are suitable to be automatically synchronized, in some cases the models are suitable to be only manually synchronized by humans.

Liu et al. [LHG07] use spreadsheet-like mechanism to define the relation over model and thus the fixing process is automated as reevaluating the cell expressions. However, this approach only propagates updates in one direction and cannot help if the location updated by users is the result of some expression.

1.3.5 Optimistic Replication

Another branch of research studying the synchronization of data is optimistic replication. Optimistic replication research considers a set of replicas edited at different people at different locations, and the system converges (synchronizes) the data at a proper time. The word "optimistic" here means the researchers are optimistic about the probability of conflicts; they assume a small probability that the user edits will conflict. Many of the optimistic replication approaches [DDH01, KRSD01, MOSMI03, EMRP01] are operation-based. They record a sequence of update operations at each replica, and merge the update operations at converging by reorder the update operations according to the semantic constraints between operations.

Compared to our approach, most the optimistic replication approaches deal with homogeneous synchronization, that is, the replicas at different locations are copies of the same source. No data transformation or format conversion is needed. On the other hand, our approach focuses on hetero-

geneous synchronization. We synchronize models of different structures and transform updates during propagation. Nevertheless, our update representation ignores the order of update operations and considers the last effect update operations. For example, if a value is first updated to "a" and then updated to "b", we only consider the latter update. As operation replication concerns a lot the order of update operations, potentially we can combine the two approaches to get more sensible synchronizers.

As far as we know, the only optimistic replication approach dealing with heterogeneous synchronization is the Harmony framework [FGK+05]. The Harmony framework is designed to synchronize two heterogeneous replicas. In this framework, developers design a common abstraction of the two replicas, and write two bidirectional transformations to map the two replicas to the common abstraction, respectively. Developers can write the bidirectional transformations in bidirectional languages or by writing separate programs in other languages. Different from Harmony, our approach aims to reuse the consistency relation specification in practices, and developers do not have to design the common abstraction and develop new transformation programs.

1.3.6 Others

Repairing broken data structures [EGSK07] is also loosely related to our approach. This work dynamically repairs faults in data at runtime according to the consistency relations implicitly specified by the assertions in code. The core to this work is a set of heuristics to fix inconsistency. Different from this work, our approach aims at providing clear, predictable synchronization semantics, so that end users can clearly know how their updates affect other parts.

Paraconsistent logic [PSR+89] is a logic allowing inference under inconsistent (contradict) premise. In classic logic, if the premise is inconsistent, anything can be inferred. Formally, $\{A, \neg A\} \Rightarrow B$ for any A and B. In paraconsistent logic, not all such premises infer everything. As synchronization is a process to make inconsistent data consistent according to a consistency relation, paraconsistent logic is potentially useful in formalizing the synchronization behavior when the consistency relation is described by a logic expression. We leave this connection for future work.

1.4 Organization of this Thesis

The rest of this thesis is organized as follows. We first build the foundations for model synchronization. Chapter 2 explains the requirement for model

synchronization, including the three properties we proposed. Chapter 3 introduces our dictionary-based representation of model and update. We also discuss the dictionary-based updates are compatible with the general model of updates.

Then we introduce our support for off-site synchronization. Chapter 4 describes how we derive a synchronizer from an ATL transformation program. Chapter 5 describes how we wrap two transformations into a synchronizer and how we use this technique to develop a runtime management system.

After that, we explain our support for on-site synchronization. Chapter 6 introduces our first-order logic synchronization language, Beanbag. Chapter 7 discusses the implementation and applications of Beanbag.

Finally, Chapter 8 concludes this thesis and discusses future work.

Chapter 2

Requirement of Model Synchronization

In this chapter we introduce the requirement for model synchronization formally. We start from the definitions of model and update, then define the requirement for an operation-based synchronizer, and finally remap the requirements to a state-based synchronizer using a model difference procedure. In this chapter we focus on abstract models and updates. We will give a concrete representation conforming to the abstract definitions in the next chapter.

2.1 Running Example

In this section we use an example to illustrate different concepts in synchronization. This example is a simplified example of the classic UML to RDBMS example [BRST05]. Figure 2.1(a) shows a basic Unified Modeling Language (UML) model containing a `Book` class with two attributes. To make the objects in this design persistent, we need to build a database model to store the objects into a database. The database model is shown in Figure 2.1(b). For each UML class whose `persistent` feature is true, we create a database

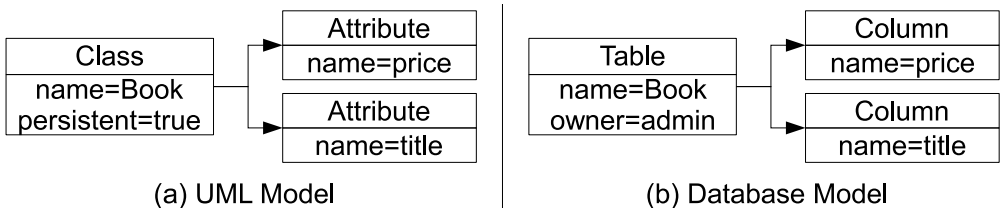

Figure 2.1: Transforming a UML model into a database model

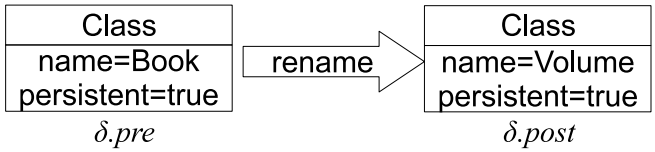

Figure 2.2: An example update δ

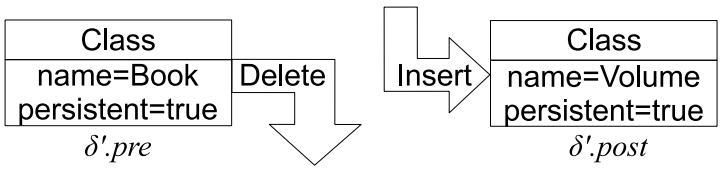

Figure 2.3: Another example update δ'

table of the same name. For each attribute belonging to a persistent class, we create a column with the same name. The database model also contains other information, the `owner` feature, and we first set it with a default value, `"admin"`. When some parts of the two models are updated by users, we need to synchronize the updates to make the two models consistent.

Note this example is an example for off-site synchronization because we are integrating two applications, a UML modeling tool and a database modeling tool. However, if we consider an integrated environment where the UML model and the database model are represented as two views, the synchronization is an on-site synchronization. Therefore on-site synchronization and off-site synchronization are the same if we only consider the abstract level. In the following we also consider in the abstract level and the synchronizer we defined can be applied to both on-site synchronization and off-site synchronization.

2.2 Models and Updates on Models

In the definition of requirement, we consider only abstract models and updates. We consider a model is a primitive unit and a meta model is a set of models. If a model conforms to a meta model, it is in the set of the meta model.

Following the definitions of Diskin [Dis08], we consider that updates on a meta model connect the models in the meta model M as a directed graph; its nodes are models, and its arrows are updates. We call the starting node of update δ the *pre-model* of δ (denoted as $\delta.pre$) and the end node of δ the

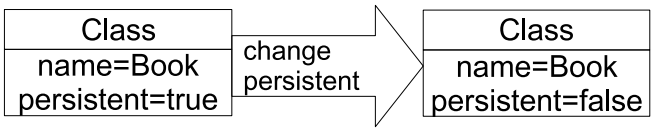

Figure 2.4: An update δ'' changing `persistent`

post-model (denoted as $\delta.post$). For example, Figure 2.2 shows an update that rename class `Book` into `Volume`. The model on the left is $\delta.pre$, the model on the right is $\delta.post$. The update itself consists of update operations indicating how the model is updated. Note the representation of the update is just an example. In our definition we only model abstract update and developers can choose any update representation in their implementations.

We also require two other properties of the graph. First, the graph is a multi-graph. In a multi-graph there can be more than one arrow between two nodes. Consequently there may be different updates leading from one model to another. For example, Figure 2.3 shows an update that is different from the update in Figure 2.2 but has the same pre-model and post-model. Second, the graph is a complete graph. In a complete graph there is at least one arrow between any two nodes. Consequent any model conforming to a meta model can be updated to any other models conforming to the same meta model. In addition, any model can be updated to itself. We use Δ_M to denote the set of updates on meta model M.

Definition 2.1. We say Δ_M represents an *update set* on meta model M if and only if there exists two functions, $pre : \Delta_M \to M$ and $post : \Delta_M \to M$ where for each pair of models (m_1, m_2) in $M \times M$, there is an update δ in Δ_M such that $pre(\delta) = m_1$ and $post(\delta) = m_2$.

Some approaches [CRP08, XZH+08] represent updates as functions, and allow an update to be applied to any model conforming to the same meta model. The ability of applying an update to multiple models is called *applicability* [CRP08]. In our definition, each update has only one associated pre-model and only one associated post-model, and cannot be directly applied to other models. The reason of this design is that we try to capture more types of update representations in our model. Some approaches [AP03] only allow one update to be applied to one model, and they conform to our model. For those approaches where one update can be applied to multiple models, we can map them to our definition by making a pair that contains an update and a pre-model.

One purpose of explicitly representing updates is to calculate the union

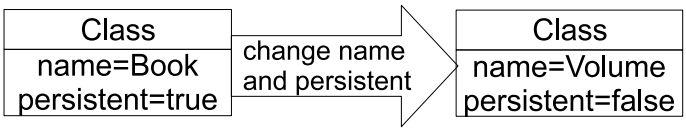

Figure 2.5: The union of δ and δ''

of two updates being applied to the same model. In distributed development environments several developers may simultaneously work on the same model, and their updates need to be merged. Given updates δ_1 and δ_2 where $\delta_1.pre = \delta_2.pre$, we denote their union as $\delta_1 + \delta_2$, where $(\delta_1 + \delta_2).pre = \delta_1.pre = \delta_2.pre$ and $(\delta_1 + \delta_2).post$ is a model that is considered to have both δ_1 and δ_2 applied. For example, consider the update in Figure 2.2 changing the `name` attribute and the update in Figure 2.4 changing the `persistent` attribute. The union of the two updates may be an update that changes both attributes.

We require several properties on the union operation. First, the union operation should be commutative, that is, $\delta_1 + \delta_2 = \delta_2 + \delta_1$. Second, the union operation should be associative, that is, $(\delta_1 + \delta_2) + \delta_3 = \delta_1 + (\delta_2 + \delta_3)$. Third, the union of an update and the update itself results in the update, that is, $\delta + \delta = \delta$. Fourth, the union operation should be a partial function. If two updates, δ_1 and δ_2, contain conflict operations and cannot be merged, the union of the two updates, $\delta_1 + \delta_2$, should be undefined. Using this property, we can define whether two updates conflict. We say two updates *conflict* if $\delta_1 + \delta_2$ does not exist. If two updates do not conflict, we say the two updates are compatible, denoted as $\delta_1 \oplus \delta_2$. For example, the update in Figure 2.2 and the update in Figure 2.3 may be considered as conflicting updates because when one user is changing an attribute on a model element, another user deletes the model element. However, whether they conflict depends on the concrete approach [AP03, CRP08] used to represent updates and implement the union operation.

One special case of update is the identity update, which means nothing is changed. We use the notation void_m to indicate an identify update on m, where $\mathsf{void}_m.pre = \mathsf{void}_m.post = m$. We require that computing the union of arbitrary update δ with an identity update should result in δ. Formally, if $\delta.pre = m$, we have $\mathsf{void}_m + \delta = \delta$.

Definition 2.2. We say the partial function $+$ is a *union operation* over update set Δ_M if and only if $+$ is commutative and associative, $\delta + \delta = \delta$ for any δ in Δ_M, and $\mathsf{void}_m + \delta = \delta$ for any void_m and δ in Δ_M where $\mathsf{void}_m.pre = \mathsf{void}_m.post = m$.

2.3 Operation-based Synchronizer and Properties

We start by defining an operation-based synchronizer and express the requirement on the synchronizer. Suppose M is a meta model defining a set of models. An *operation-based synchronizer* consists of the following two components:

- a set of models that are consistent $R \in M$
- a partial synchronization function $sync : \Delta_M \to \Delta_M$, where $\forall \delta \in \Delta_M, sync(\delta).pre = \delta.pre$.

The first component defines the consistency relation over the model. We define the consistency relation as part of the synchronizer so that we can allow different consistency relations over the same meta model, and different synchronizers may synchronize according to different consistency relations. The second component takes an update on the model and produce new update where the new update is expected to be applied on the model to make it consistent according to the relation R. Because we assume each update has a unique pre-model, by taking the update as input, the synchronizer may not only use the information of update operations but also the current value of the model.

To keep the definition simple, we define the synchronizer on one meta model, and the synchronization function only synchronizes updates on one model. However, for situations like off-site synchronization that synchronizes multiple models, we can assume that each model in the meta model M is a tuple of models. That is, $M = M_1 \times M_2 \times \ldots \times M_n$ where M_i defines the ith model in the synchronization.

For example, Figure 2.6 shows an example of input update, where the `price` attribute is renamed to `bookPrice`. A possible output of the input is shown in Figure 2.7. The synchronizer propagates the renaming from the UML model to the database model to make all models consistent.

This definition already implies some requirements for model synchronization. First, the synchronization is a function, which means that the result of synchronization must be deterministic. An alternative approach would be to allow the synchronization to produce an update that does not fully depend on the input update. For example, an "intelligent" synchronizer may obtain information from the environment or invent information by itself. We reject this approach because we want to model predictable synchronization behavior. It is important for developers to fully control the synchronization behavior so that they can use the synchronizer in applications.

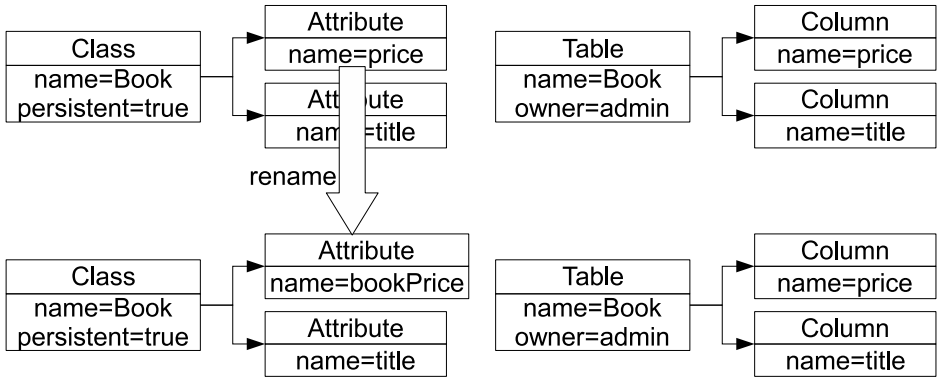

Figure 2.6: Input update of a synchronization function

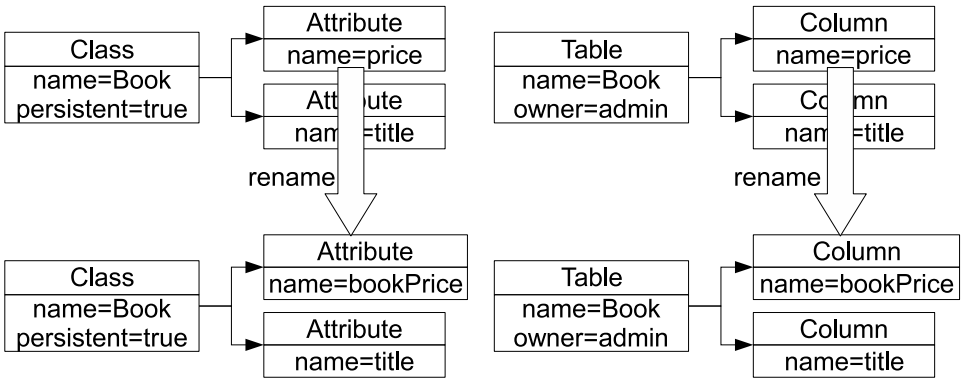

Figure 2.7: Output update of a synchronization function

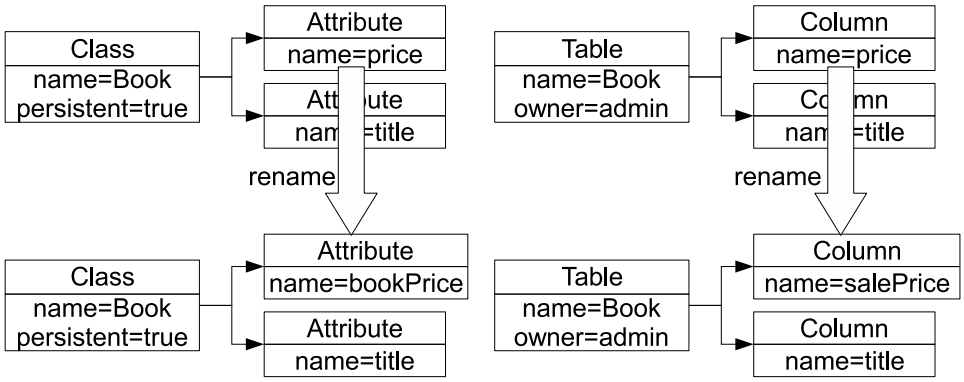

Figure 2.8: Conflicting update

Second, the function is partial, which implies detection of conflicts in updates. If the updates to the two models conflict, the function should be undefined for these input. This definition is similar to the definition of the union operation, but in this case, not only two updates applied to the same location may conflict, but also operations applied to different locations in one update may conflict according to the consistency relation over models. For example, Figure 2.8 shows an example of a conflicting update. The `price` attribute and the `price` column are renamed to different names and cannot be synchronized.

However, apart from the above two requirement, this definition alone does not impose much constraint on the behavior of the synchronization. We introduce three properties to ensure the synchronization procedure behaves in a reasonable way. These properties were motivated by previous studies on the semantics of bidirectional transformation [Ste07, FGM+07] and database view updating [BS81, DB82], and we extend the original studies to more general model synchronization.

Our first property, CONSISTENCY, requires that the synchronization procedure to do something useful. It ensures that consistency relation R is established after the synchronization.

Property 1 (CONSISTENCY)**.**
$sync(\delta)$ is defined $\Longrightarrow sync(\delta).post \in R$

The second property, STABILITY, prevents the synchronization procedure from doing something harmful. If neither of the two models has been updated, the synchronization procedure should update neither of them.

Property 2 (STABILITY)**.**
$\delta.pre \in R \wedge \delta.pre = \delta.post \Longrightarrow sync(\delta).post = \delta.post$

The last property, PRESERVATION, is more interesting. Consider the updates shown in Figure 2.6. The easiest way to achieve consistency is to change the attribute name from `"bookPrice"` back to `"price"`. However, this is not the behavior we want. What we want is that the updates are propagated from the modified parts to the unmodified parts, rather than changing back the modified parts. To prevent the unwanted behavior, we require that the user updates be preserved in the output models. If the user changes the name of the price attribute to `"bookPrice"`, the synchronization procedure should not change the attribute to any other value.

To do this, we define a preservation relation between updates. This relation can be defined using the union operation. Basically, we can consider that the union of two updates preserves each update. As a result, if the union of two updates is equal to one of the update, the other update is preserved in this update.

Definition 2.3. We say that δ_2 *preserves* δ_1, denoted as $\delta_1 \sqsubseteq \delta_2$, if and only if $\delta_1 + \delta_2 = \delta_2$.

We can see that the preservation relation is a partial order over updates. First, the preservation relation is reflexive. We have $\delta \sqsubseteq \delta$ because $\delta + \delta = \delta$. Second, the preservation relation is antisymmetry. If $\delta_1 + \delta_2 = \delta_2$ and $\delta_2 + \delta_1 = \delta_1$, we have $\delta_1 = \delta_2$ because the union operation is commutative. Third, the preservation relation is transitive. If $\delta_1 + \delta_2 = \delta_2$ and $\delta_2 + \delta_3 = \delta_3$, we have $\delta_1 + \delta_3 = \delta_1 + (\delta_2 + \delta_3) = (\delta_1 + \delta_2) + \delta_3 = \delta_2 + \delta_3 = \delta_3$.

After we have the preservation relation, we can define the preservation properties using the relation.

Property 3 (PRESERVATION)**.**
$sync(\delta)$ is defined $\Longrightarrow \delta \sqsubseteq sync(\delta)$

2.4 State-based Synchronizer and Properties

In this section we lift the operation-based synchronizer we defined in the last section into a state-based synchronizer. Different from the operation-based synchronizers, the state-based synchronizers takes a set of models and produces new models where the updates are synchronized. The key here is to use a model difference operation to find an update from two different versions of models.

Model difference has been intensively studied by many researchers [AP03, AAAN+06, MGH05, XS05]. In their approaches, they compare two versions of models and try to find what update operations can update the first model

into the second. We call this operation the *difference operation*. Formally, a difference operation is a function, $\mathit{diff} \in M \times M \to \Delta_M$, that takes two models, m and m', and produces update δ, where $\delta.\mathit{pre} = m$ and $\delta.\mathit{post} = m'$. We define a difference operation as a function to require the procedure to be deterministic.

As usually there are more than one update leading a model to another model, we require that the difference operation returns an update where no smaller update exists according to the preservation relation. Formally, if $\mathit{diff}(m, m') = \delta$, there is no δ' such that $\delta'.\mathit{pre} = m$, $\delta'.\mathit{post} = m'$ and not $\delta \sqsubseteq \delta'$. Because the preservation relation is a partial order, it is possible that there is no least element for the set of updates that updates m to m'. In such case a difference operation should choose one update from all possible updates using predefined criteria. For example, in Alanen et al.'s approach [AP03], the result is a set of insertions and deletions that preserve the longest common subsequence when comparing two ordered features.

One corollary about the difference operation is that differencing a model and the model itself results the void update, that is, $\forall m \in M, \mathit{diff}(m, m) = \mathtt{void}_M$. This is because void update a model to itself and is preserved in any update (for any update δ, $\mathsf{void} + \delta = \delta$). Since the difference operation should return an update where no smaller update exists, the only choice for the difference operation is to return void.

Now we proceed to define a state-based synchronizer. Same as the definition of operation-based synchronizer, we only define this synchronizer on one model. To synchronize multiple models, we consider the model as a tuple of models. Suppose M is a meta model. A *state-based synchronization function* consists of the following two components:

- a set of models that are consistent $R \in M$
- a partial synchronization function $\mathit{sync} : R \times M \to M$

A state-based synchronizer has the same form of an operation-base synchronizer but the input and output of the synchronization function is different. The state-based synchronization function takes two models as input, one is the model before update (which must be contained in the consistent model set R), and one is the model updated by users. This function returns a model depends on the two inputs, and in the model the updates are synchronized.

Similarly, we define the three properties on state-based synchronizers. We first define CONSISTENCY, which requires the output model to be consistent.

Property 4 (CONSISTENCY on state-based synchronizer)**.**
$\mathit{sync}(m, m')$ is defined $\implies \mathit{sync}(m, m') \in R$

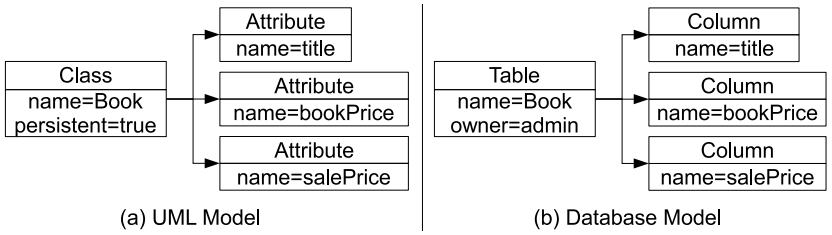

Figure 2.9: Updates to both models are preserved

Second, the STABILITY property on state-based synchronizers requires the output model be equal to the input if the two input models are equal.

Property 5 (STABILITY on state-based synchronizer)**.**
$m \in R \Longrightarrow sync(m, m) = m$

The PRESERVATION property is more complex. As a state-based synchronizer does not know the update operations, it is possible that when we consider different sets of updates, there may be different preservation result. For example, we have seen an example of conflicting update in Figure 2.8. However, as in the state-based synchronization we only consider the model before and after update, we may also regard the update as "delete the `price` attribute, insert the `bookPrice` attribute, delelte the `price` column, and insert the `salePrice` column". In this case the updates on the two models do not conflict and we may produce a synchronized model as shown in Figure 2.9.

As a result, we need to diminish the ambiguity to define the PRESERVATION property on state-based synchronizers. Because a difference operation chooses a unique update from a set of updates, we can use a difference operation to define the preservation relation on models. Given a difference operation *diff*, we say m'' *preserves* the update from m to m' according to *diff* if and only if there exists an update δ where $(diff(m, m') + \delta).post = m''$. We use $m \triangleright m' \sqsubseteq_{diff} m''$ to represent that m'' preserves the update from m to m'.

Using the preservation relation on models, we can define the PRESERVATION property. Given a difference operation *diff*, we say a state-based synchronizer satisfies PRESERVATION according to *diff* if and only if it satisfies the following conditions.

Property 6 (PRESERVATION on state-based synchronizer)**.**
$sync(m, m') = m'' \Longrightarrow m \rightarrow m \triangleright m' \sqsubseteq_{diff} m''$

2.5 Constructing a State-based Synchronizer from an Operation-based Synchronizer

Since the state-based synchronizer and the operation-based synchronizer are similar, one natural question to ask is whether we can construct one from the other so that we only have to define one type of synchronizer. The answer is positive. We can construct a state-based synchronizer from an operation-based synchronizer. The basic idea is to use a model difference operation to find the update between two models.

Given an operation-based synchronizer $(R, sync)$ and a difference operation $diff$, a state-based synchronizer $(R', sync')$ can be constructed using the following formulas.

$$\begin{array}{lcl} R' & = & R \\ sync'(m, m') & = & sync(diff(m, m')).post \end{array}$$

We can prove that the constructed synchronizer satisfies the three synchronization properties.

Theorem 2.4. *If* $(R, sync)$ *satisfies* CONSISTENCY, PRESERVATION *and* STABILITY, *the constructed synchronizer* $(R', sync')$ *satisfies* CONSISTENCY, PRESERVATION *and* STABILITY.

Proof. **Consistency** If $sync'(m, m')$ is defined, $sync(diff(m, m'))$ must be defined, and thus $sync(diff(m, m')).post \in R$.

Stability Because we have $m \in R$ and $diff(m, m) = \mathsf{void}_m$, we have the two preconditions, $diff(m, m).pre \in R$ and $diff(m, m).pre = diff(m, m).post$. Because $sync$ satisfy STABILITY, we have $sync(diff(m, m)).post = diff(m, m).post = \mathsf{void}_m.post = m$.

Preservation If we have $sync'(m, m') = m''$, we have $sync(diff(m, m'))$ is defined. From the preservation property we have $diff(m, m') \sqsubseteq sync(diff(m, m'))$. From the definition of preservation we can construct $diff(m, m') + sync(diff(m, m')) = sync(diff(m, m'))$. As we know $sync(diff(m, m')).post = m''$, we have $m \triangleright m' \sqsubseteq_{diff} m''$.

□

2.6 Related Work

Many people have discussed the properties of synchronization based on the context of asymmetric bidirectional transformation[Ste07], symmetric bidirectional transformation [FGM+07] and database view updating [BS81]. Our

requirements are inspired from their work, but as far as we know, we are the first to apply these properties to model synchronization area. In particular, we are the first to consider the operation-based synchronizer and to allow parallel updates on models. We propose a new property, PRESERVATION, to capture these new issues.

Some researchers build frameworks for classifying synchronization approaches. Antkiewicz and Czarnecki [AC08] classifies synchronization approaches using different design decisions. Under their classification schema, our state-based synchronizer can be classified as a "bidirectional, original-target-dependent, and many-to-many synchronizer". Diskin [Dis08] builds a more formal framework for bidirectional model synchronization, in which bidirectional transformation is classified into lenses, di-systems, and tri-systems on the basis of the relation between models and the number of input models. Our definition of the state-based synchronizer can be considered a supplement to his framework, where we add quadruple-systems, in the sense that our synchronizer takes four models as input. Furthermore, our operation-based synchronizer is new to both classification frameworks.

Chapter 3

Representing Models and Updates

As mentioned in Section 1.1.1, models are defined by the MOF standard [OMG02]. MOF standard is a very large standard. The standard consists of 88 pages, defining an essential MOF model and a complete MOF model, where each model consists of dozens of classes and other primitive types. Such model definitions are easy for end user to use because these models define many concepts close to different application domains. However, the definitions are not easy for tool developers to develop tools, nor for researchers to study, because the developers and researchers have to keep many concepts in mind. Furthermore, the standard is not formally defined. The standard is described in a semi-formal way by presenting the concepts and definitions in the MOF notations itself, but the MOF notations are not given a clearly-defined formal foundation. This makes it further difficult for tool developers and researchers because the same concepts may be interpreted differently by different people, and tool developers and researchers have to deal with such ambiguity.

To solve the problem, many researchers propose formal notations and semantics for object models [BC95, BG98, Jac02, CK01]. These notations mainly share two features. One is small. These notations often map the large, complex MOF model to a small set of concepts, so that tool developers and researchers only need to consider the small set of concepts. The other one is formal. The small set of concepts is often formally defined on some widely-used concepts like the set theory [CK01] or Larch Shared Language [BC95].

However, although many formalizations have been given, as far as we know no formalization has considered the updates on the models. The existing approaches [AP03, CRE06] for representing updates on models are often

directly defined on MOF standard, and thus suffer from the two problems of the MOF standard, being large and being informal. Since the target of this thesis is to deal with updates on models, it is necessary to give small and formal notations for representing models so that we can easily work on.

This chapter presents our formal notations for representing models and updates. Concretely, the contributions of this section can be summarized as follows.

- We propose a lightweight notation for representing models. Our notation is based on a core concept: dictionary, which is a function mapping keys to values. We show that most frequently-used concepts in models can be represented in dictionaries by assigning unique identifiers to elements. As a result, our representation can be considered as a formalization of MOF models.
- We define the updates on dictionaries. The updates on dictionaries are defined by dictionary itself. We also show that updates on models can be represented by updates on dictionaries, and the representation enjoys a set of properties so that we can freely use the operators on updates to compose them.

The rest of this chapter is organized as follows. Section 3.1 briefly introduces the MOF standard and the main concepts in this standard. Section 3.2 describes our dictionary notation, and Section 3.3 describes how to represent models in dictionaries. After that, Section 3.4 describes updates on dictionaries, and Section 3.5 describes how to represent model updates as dictionary updates. Finally, Section 3.6 discusses related work.

3.1 MetaObject Facility

Figure 3.1 shows a simplified model of MOF types. This model defines what should be contained in a MOF meta model. This model also conforms to itself, so it is the meta meta model of all MOF meta models. The core element in this model is `Type`, which defines what types can be included in a MOF model. Two classes are derived from `Type`. One is `PrimitiveType`, whose child classes (not shown in the figure) define the primitive types. The other one is `Class`, for users to define their own types in the meta model. A class can have one or more super classes. A class can be abstract or not, where an abstract class cannot be instantiated as an object. A class contains a set of properties. The `Property` class is derived from `TypedElement` and `MultiplicityElement`. `TypedElement` has a reference to `Type`, and

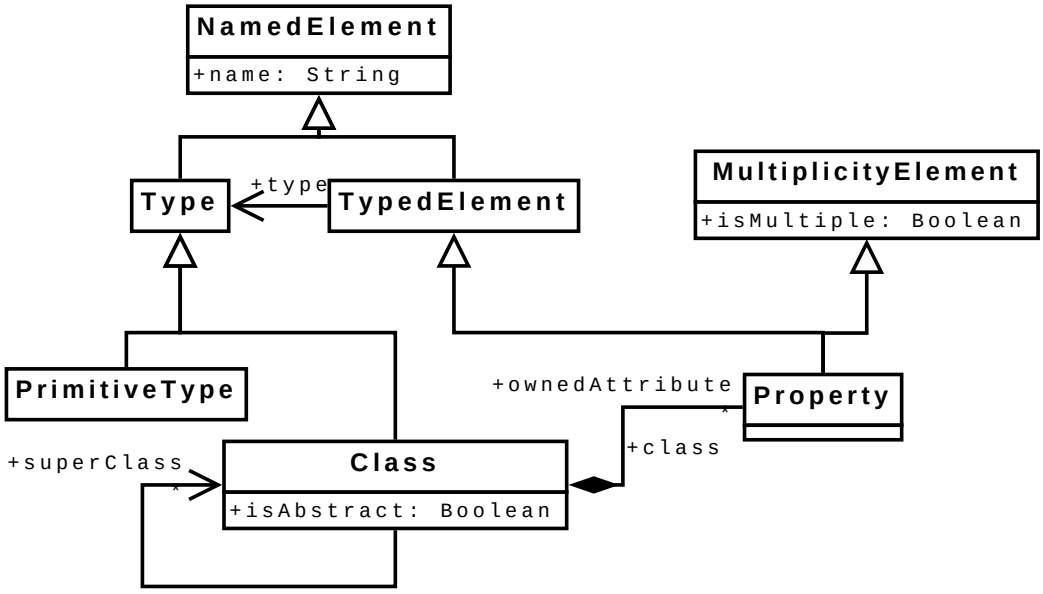

Figure 3.1: Simplified MOF types

thus a property has a type. If the type is a primitive type, we call the property attribute. If the type is a class, we call the property reference. `MultiplicityElement` allows the element to be single or multiple. A multiple property can have more than one value and thus is a multiset. In addition, both `Type` and `TypedElement` are derived from `NamedElement`, so they both have a name.

Let us review the example we have seen in Chapter 1. Figure 3.2 shows the example containing a meta model (left) and a model (right). The meta model is defined by the meta meta model in Figure 3.1. It contains three classes, two classes containing a single property and one class containing a multiple reference. These classes are instances of the `Class` class in the meta meta model, and we call such instances model elements. The model is defined by the meta model. It contains three model elements, one directory and two files. One thing to note here is that the semantics of instantiating the meta meta model and that of instantiating a meta model are the same. In this way the MOF standard defines three layers of abstractions (meta meta model, meta model and model) in a unique way using the same set of semantic rules.

It is worth remarking that this diagram is greatly simplified from the original MOF definition for simplicity. Two categories of elements are ignored in this diagram. One category is the elements that are not related

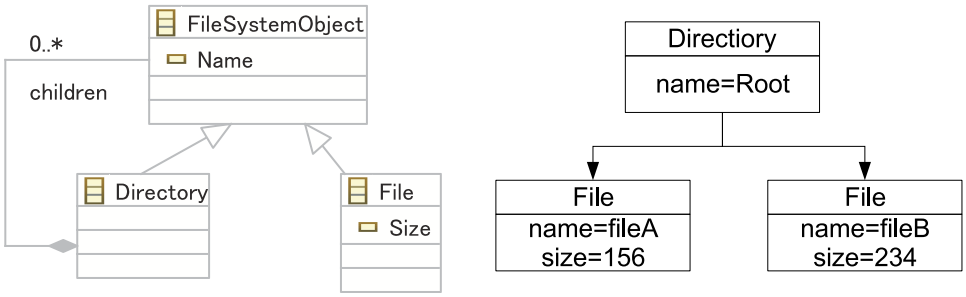

Figure 3.2: A MOF meta model and its model

to model updating. For example, according to the MOF standard, every class may contain a set of side-effect-free operations. These operations are only defined in the meta model but not in the model, so it is not possible for users to update these operations, and we do not include them for clarity. The second category is the elements that are currently not supported by the dictionary representation in this chapter. For example, the `Property` class has a `composite` property indicating that the object owning the property is contained in another object referred by the property. Other such elements include the `isOrdered` property and the `isUnique` property on `MultiplicityElement` and the `isID` property on `Property`. The unsupported properties are all considered `false` in this chapter. Nevertheless, some unsupported properties can be captured using synchronizers, and we shall discuss them in Chapter 6.

3.2 Dictionary-based Data Definition

We have seen what concepts are defined in the MOF standard. Let us see the definition of dictionary-based data. Dictionaries are functions mapping keys to values. Dictionaries can be used to represent many different kinds of data structures as studied by Benjamin et al. [FGK+05][1]. We first give the definition of dictionary-based data in this section, and describe how to map models to dictionaries in the next section.

Figure 3.3 shows the syntax of dictionary-based data. The *italic symbols* are non-terminals and the `Sans Serif` symbols are terminals. `STRING`, `NUMBER` and `BOOLEAN` are lexical tokens of common meanings. There are two types of data (values). One is *primitive* values including numbers, strings, booleans and a `null` value, and the other is *dictionaries* that map keys (prim-

[1] In [FGK+05], dictionaries are known as edge-labeled trees

value ::= *primitive* | *dictionary*
primitive ::= `NUMBER` | `STRING` | `BOOLEAN` | `null`
dictionary ::= {*entries*}
entries ::= *entry* | *entry*, *entries*
entry ::= *primitive* –> *value*

Figure 3.3: Syntax of dictionary-based data

itive values) to other values. A key-value pair is called an *entry*. Although keys can only be primitive values, the value part can be either primitive values or other dictionaries, forming a hierarchical structure. For example, `{1->{"name"->"Root"}, 2->{"name"->"fileA", "size"->156}}` is a dictionary that contains two embedded dictionaries.

We may consider all keys that do not exist in the definition are mapped to `null`. So, if a key is mapped to `null`, it means that the key does not exist in the dictionary. That is, `{"a"->null}` and {} are both empty dictionaries. This definition is very important for simplifying the definition of updates. When we want to delete an entry in a dictionary, we can just set the corresponding key to `null`. In this way we give a unique representation to both replacement and deletion. The set of keys that are not mapped to `null` by a dictionary d is called the domain of the dictionary, denoted as $dom(d)$. We write $d.k$ for the value to which the dictionary d maps the key k.

3.3 Representing Models

There are many ways to map MOF-based models into dictionary-based data. For example, one can encode a model into XML as defined in the XMI standard [Obj07], and map the XML file (which is a tree structure and can be easily converted to dictionaries) into dictionary-based format. In this chapter we describe a standard representation that will be used throughout the thesis and discuss several properties this representation holds.

The first step of our conversion is to assign a unique identifier (UID) to each model element. In many cases we need to identify a model element. For example, when one property refers to a model element, or when we update a model element. In implementation, these UIDs are often implicitly used in the implementation. For example, an in-memory model may use memory addresses to identify model elements, and an XMI model may use XPath [CD+99] to identify model elements. As we are representing models as dictionaries, we need to identify each model element so that we can make clear the correspondence between model elements and dictionaries. The UID for each

model may be randomly generated, incrementally assigned or transformed from the implicit UIDs in the implementation.

If a property of a model element is a multiple property, we also assign a UID to each value in the multiset. After we assign UIDs to all model elements and all values in multisets, we say that the model becomes a model with UIDs. In the following we will discuss the mapping between models with UIDS and dictionaries.

One requirement on UIDs is that each UID must be unique among the whole system, even if the system is distributed at different locations. This seems a strong requirement as we have to keep all distributed locations coordinated at all time. For example, if two programmers at distributed locations both insert a model element to the model, the UIDs of two model elements must be different. However, as we do not really care about the UIDs before we integrate the distributed models, we may leave the UIDs uncoordinated and coordinate them when we merge the models. Every distributed model may use their own local UIDs, and when we merge these models, we generate a new set of global UIDs and keep bijective mappings between the global UIDs and the local UIDs. The bijective mappings can be stored and be used every time we merge the models.

After we convert models into models with UIDs, we can represent models into dictionaries. A model, its meta model and the meta meta model are represented as three dictionaries, respectively. The model is represented as a dictionary mapping UIDs to the model elements. Each model element is represented as a dictionary mapping property names to property values. The property values are different between single properties and multiple properties. For a single property, if the type of the property is a primitive type, the property value is the primitive value. If the type of the property is a class, the property value is the UID of referred model element. For a multiple property, we represent its value as a dictionary mapping UIDs to the members in the feature.

The meta model and the meta meta model is also represented in the same way. For each model element, we also insert a special key `"__type"` mapping to the UID of its meta model element in the meta model. For example, the model in Figure 3.2 is mapped into the following dictionary, where the numbers starting with "id" are generated IDs.

```
{id1->{"__type"->id10,
     "name"->"Root",
     "children"->{id4->id2, id5->id3}},
 id2->{"__type"->id20,
     "name"->"fileA",
     "size"->156},
```

```
id2->{"__type"->id20,
     "name"->"fileB",
     "size"->234}}
```

Reversely, we can convert a dictionary back into a model with UIDs. For each entry in the top dictionary, we check its `__type` key, and create a new model element according to the key. After we create all model elements, we set the properties of the dictionary. If the name of a property is not in the dictionary, we set it to `null`.

Using the above two algorithms, we can map a model with UIDs to a dictionary and vice versa. One property of the conversions is that if we convert a model with UIDs to a dictionary and then convert the dictionary back into a model, the model should be the same as the original model. In other words, if we consider the two conversions as functions, either function is the inverse of the other.

As our data definition allows more free form of dictionaries than dictionaries converted from models, one question to ask is whether a dictionary can be converted back into a model or not. Because a model must conform to its meta model, we need also check whether the dictionary conforms to its meta model dictionary. We use the following set of rules to do the checking. Given a dictionary m representing a model, a dictionary mm representing a meta model and a dictionary mmm representing a meta meta model, the dictionary m can be converted back into a model conforming to mm according to mmm, denoted as $m \in_{mmm} mm$, if it satisfies the following rules.

1. mm can be converted to a model conforming to mmm and mmm can be converted to a model conforming to itself according to the rules below.

2. All keys in m must be mapped to dictionaries (each representing a model element):

$$\forall k \in dom(m) : m.k \in dictionary$$

3. Each dictionary representing a model element should include a `__type` key which pointed to a class in the meta model:

$$\forall k \in dom(m) : mmm.(mm.(m.k.\texttt{"\_\_type"}).\texttt{"\_\_type"}).\texttt{"name"} = \texttt{"Class"}$$

4. Each key in a model element dictionary (a property name) should be equal to the name of a property in the class or a super class of the class:

$$\begin{aligned}&\forall k \in dom(m) : \forall n \in dom(m.k) : \\ &\quad n == \texttt{"\_\_type"} \vee \\ &\quad (\exists p \in allProperties(mm.(m.k.\texttt{"\_\_type"}), mm) : \\ &\qquad mm.p.\texttt{"name"} = n),\end{aligned}$$

$$
\begin{array}{ll}
\mathit{update} & ::= \mathit{pupdate} \mid \mathit{dupdate} \mid \texttt{void} \\
\mathit{pupdate} & ::= !\mathit{primitive} \\
\mathit{dupdate} & ::= \{\mathit{update\_entries}\} \\
\mathit{update\_entries} & ::= \mathit{update\_entry} \mid \mathit{update\_entry}, \mathit{update\_entries} \\
\mathit{update\_entry} & ::= \mathit{primitive} \texttt{ -> } \mathit{update}
\end{array}
$$

Figure 3.4: Syntax of dictionary-based updates

where *allProperties* returns all property names in the class and all its super classes.

5. Each value mapped by a property name in a model element dictionary should conform to its type:

$$
\begin{array}{l}
\forall k \in \mathit{dom}(m) : \forall n \in \mathit{dom}(m.k) : \\
\mathit{isPrimitive}(m.k.\texttt{"\_\_type"}, n, mm, mmm) \rightarrow \\
\quad \mathit{conform}(m.k.n, \mathit{getType}(m.k.\texttt{"\_\_type"}, n, mm, mmm)) \wedge \\
\mathit{isClass}(m.k.\texttt{"\_\_type"}, n, mm, mmm) \rightarrow \\
\quad m.(m.k.n).\texttt{"\_\_type"} \in \mathit{getAllSubTypes}(m.k.\texttt{"\_\_type"}, n, mm, mmm),
\end{array}
$$

where *isPrimitive* determines whether the type of n is a primitive type, *isClass* determines whether the type of n is a class type, *conform* checks whether the value conforms to the primitive type and *getType* returns the set of keys mapped to the type class and all its sub classes.

3.4 Dictionary-based Update Definition

Dictionaries not only allow us to represent many data structures but also enable us to uniquely identify every location in a dictionary. We make use of this feature to represent updates. An update on a dictionary is represented by the location of updates and the updated value. Figure 3.4 gives the syntax of updates. An update can be either *pupdate* – an update on primitive values, *dupdate* – an update on dictionaries, or void – indicating that nothing has been changed. An update on primitive values just contains a new value and an update on dictionaries maps keys to updates. If a key does not exist in the *dupdate* definition, we assume the key is mapped to void. The set of keys that are not mapped to `void` is the domain of the dictionary-update, denoted as *dom*(*dupdate*). We also use the notation *dupdate.k* on dictionary-updates. Its meanings are the same as that defined on dictionaries.

Figure 3.5 shows the denotational semantics of updates. The denotation of an update u, represented by $U[\![u]\!]$, is a function mapping between values. The denotation of void changes nothing. The denotation of *pupdate*

$$
\begin{array}{ll}
U[\![\texttt{void}]\!](v) & = v \\
U[\![!primitive]\!](v) & = primitive \\
U[\![dupdate]\!](d) & = \begin{cases} U[\![dupdate]\!](\{\}) & d \notin dictionary \\ d' & d \in dictionary \end{cases} \\
\multicolumn{2}{l}{\text{where } d' = \bigcup_{\forall k \in dom(dupdate) \cup dom(d)} \{k\texttt{->}U[\![dupdate.k]\!](d.k)\}}
\end{array}
$$

Figure 3.5: Semantics of dictionary-based updates

Table 3.1: The result of $u_2 \circ u_1$

	$u_2 = \mathsf{void}$	$u_2 \in pupdate$	$u_2 \in dupdate$
$u_1 = \mathsf{void}$	void	u_2	u_2
$u_1 \in pupdate$	u_1	u_2	u_2
$u_1 \in dupdate$	u_1	u_2	u_3

where $u_3 = \bigcup_{\forall k \in dom(u_1) \cup dom(u_2)} \{k\texttt{->}(u_2.k \circ u_1.k)\}$

maps everything to the new value, e.g., $U[\![\texttt{!3}]\!](2) = 3$. The denotation of *dupdate* applies every update in the domain of the dictionary-update to the value mapped by the same key, e.g., $U[\![\{\texttt{2->!"a"}\}]\!](\{\texttt{2->"m"}\})=\{\texttt{2->"a"}\}$. One property of the semantics is that the denotation of an update is always idempotent, i.e., $\forall u \in update, \forall v \in value : U[\![u]\!](U[\![u]\!](v)) = U[\![u]\!](v)$.

Two updates can be merged (composed). Sometime users may perform a sequence of updates before the modeling tool can perform a fix, e.g., in a distributed environment. In such case we need to merge a sequence of updates into a single update. We use $u_2 \circ u_1$ to denote merging two updates u_1 and u_2 where u_1 is considered earlier than u_2. Table 3.1 shows the rules for merging two updates. For example, merging {`1->"a", 2->"b"`} with {`1->"c", 3->"d"`} results in {`1->"c", 2->"b", 3->"d"`}. A requirement on merging is that merging should preserve the semantics of updates. In other words, $U[\![u_2 \circ u_1]\!] = U[\![u_2]\!] \circ U[\![u_1]\!]$. We can easily prove that the rules in Table 3.1 satisfy the requirement by checking the definitions. Another property of merging is associative. That is, $(u_1 \circ u_2) \circ u_3 = u_1 \circ (u_2 \circ u_3)$.

However, if two users change the same location to different values, we say that the two updates by the two users conflict with each other. If two updates do not conflict, we say the two updates are compatible, denoted as $u_1 \oplus u_2$. We use merging to define compatibility. Formally, $u_1 \oplus u_2$ iff $u_1 \circ u_2 = u_2 \circ u_1$. For example, {`1->!"a"`} and {`1->!"b"`} conflict but {`1->!"a"`} and {`2->!"b"`} are compatible.

A partial order can be defined over updates. We may want to know if one update u_1 is completely included in another update u_2. In other words, the locations changed by u_1 are all changed to the same values by u_2. This can

Table 3.2: The result of $\mathit{find\_update}(v_1, v_2)$ when $v_1 \neq v_2$

	$v_2 \in \mathit{primitive}$	$v_2 \in \mathit{dictionary}$
$v_1 \in \mathit{primitive}$	$!\, v_2$	$\mathit{find\_update}(\{\}, v_2)$
$v_1 \in \mathit{dictionary}$	$!\, v_2$	u

where $u = \bigcup_{\forall k \in \mathit{dom}(v_1) \cup \mathit{dom}(v_2)} \{k\texttt{->}\mathit{find\_update}(v_1.k, v_2.k)\}$

also be formally defined by merging. If $u_1 \circ u_2 = u_2$, we say u_1 is included in u_2, denoted as $u_1 \sqsubseteq u_2$.

One property of dictionary-based updates is that we can always find a minimal updates from a value v_1 to another value v_2 where all other updates from v_1 to v_2 include the update. We use a function *find_update* to get the minimal update from two values. Formally, $\forall u \in \mathit{update}, U[\![u]\!](v_1) = v_2 \Longrightarrow \mathit{find\_update}(v_1, v_2) \sqsubseteq u$. The function *find_update* can be defined as follows: 1) it returns `void` if the two values are equal, and 2) it follows the rules in Table 3.2 for other inputs.

3.4.1 From Dictionary-based Update to General Update

You may have noticed that dictionary-based updates do not conform to our definition of updates in Chapter 2. To distinguish, we call the update defined in Chapter 2 edit update because it records the edits on a model. In this section we show how dictionary-based updates can be converted to edit updates.

The basic idea of converting a dictionary-based update to an edit update is to make a pair containing the update and a model. As an edit update describing how a pre-model changes to a post-model, we may consider that the model in the pair corresponds to the pre-model and the update describing the update process. In addition, we can get the post model by applying the update to the model.

Theorem 3.1. *The set* $(\mathit{value} \times \mathit{update})$ *is an update set on values.*

Proof. Let us define the two functions, *pre* and *post*, as follows.

$$\begin{aligned} \mathit{pre}(\langle v, u \rangle) &= v \\ \mathit{post}(\langle v, u \rangle) &= U[\![u]\!](v) \end{aligned}$$

Then for any v_1, v_2 in *values*, we have an update $\delta = \langle v_1, \mathit{find\_update}(v_1, v_2) \rangle$ where $\delta.\mathit{pre} = v_1$ and $\delta.\mathit{post} = v_2$. □

In addition, the union operation can be defined by merging compatible updates, as follows. In the definition, $\perp$ indicate the operation is undefined at the input.

$$\langle v, u_1\rangle + \langle v, u_2\rangle = \begin{cases} \langle v, u_1 \circ u_2\rangle & u_1 \oplus u_2 \\ \perp & \text{otherwise} \end{cases}$$

Theorem 3.2. *The operation +is a union operation over the set* (*value* $\times$ *update*).

Proof. To prove +is a union operation, we need to prove the four properties required for the union operation.

- First, +is commutative. This is because it is only defined when $u_1 \oplus u_2$.
- Second, +is associative. This is because merging is associative.
- Third, for any $\langle v, u\rangle \in$ (*values* $\times$ *update*), we have $\langle v, u\rangle + \langle v, u\rangle = \langle v, u \circ u\rangle = \langle v, u\rangle$.
- Fourth, for any $v \in$ *value*, $\mathsf{void}_v = \langle v, \mathit{find\_update}(v, v)\rangle = \langle v, \mathsf{void}\rangle$. Because merging void with any update u results in u, we have for any $\delta \in$ (*values* $\times$ *update*), $\mathsf{void}_v + \delta = \delta$.

□

3.5 Representing Model Updates

After we define the updates on dictionaries, we proceed to consider how to represent updates on models with UIDs. We use the UIDs to locate elements and items in a model when specifying updates. Concretely, we consider an update on model is a sequence of update operations, where the update operations fall into nine types, as follows.

1. Insert a model element with UID `e`.
2. Delete a model element with UID `e`.
3. Replace the value of single attribute `p` in model element `e` with a new value, `v`.
4. Replace an item at the location `i` in multiple attribute `p` in model element `e` with a new value, `v`.

5. Insert a new value, `v`, at the location `i` in multiple attribute `p` in model element `e`.

6. Delete an item at the location `i` in multiple property `p` in model element `e`.

7. Change a single reference `p` in model element `e` to refer to model element `e'`.

8. Change an item at the location `i` in multiple reference `p` in model element `e` to refer to model element `e'`.

9. Insert an item referring to model element `e'` at the location `i` in multiple attribute `p` in model element `e`.

When we convert a model with UIDs into a dictionary, we can represent the nine types of updates into updates on the dictionary accordingly. Concretely, the nine types of updates can be represented by the following six updates.

1. {e->{}}
   A dictionary representing the new model element is inserted, where all properties of the new model element are left `null`.

2. {e->!null}
   A dictionary at `e` is deleted.

3. {e->{p->!v}}
   The value mapped by p in the dictionary representing model element `e` is replaced by `v`.

4. {e->{p->{i->!v}}}
   The value mapped by i in the dictionary representing the value of property `p` in model element `e` is replaced by `v`.

5. {e->{p->{i->!v}}}
   Insertion has the same form as replacement because the UID distinguishes insertion from replacement.

6. {e->{p->{i->null}}}
   The value mapped by i in the dictionary representing the value of property `p` in model element `e` is removed.

7. {e->{p->!e'}}
   This is similar to replacing an attribute except that the referred model element is represented by its UID.

8. {e->{p->{i->!e'}}}
   This is similar to replacing an item in a multiple attribute except that the referred model element is represented by its UID.

9. {e->{p->{i->!e'}}}
   This is similar to inserting an item in a multiple attribute except that the referred model element is represented by its UID.

Using the above rules, we can represent model updates as dictionary-based updates. On the other hand, we would like to see whether it is possible to convert dictionary-based updates back into the nine types of model updates. From the representation rules we can see that the updates converted from model updates is a subset of all dictionary-based updates, and thus we need to first figure out what subset can be converted back. We have the following theorem.

Theorem 3.3. *Suppose u be a dictionary-based update, m, mm and mmm be three dictionaries, and $m \in_{mmm} mm$. If $U[\![u]\!](m) \in_{mmm} mm$, u can be converted back into a sequence of updates on the model converted from m.*

This theorem can be proved by defining algorithms to convert such a dictionary-based update back. The algorithms are outlined below. One key part of the algorithm is that we first insert/delete all model elements and then set the properties of all model elements. In this way we can ensure when we change a reference, the value of the reference is valid.

1. Update u must be a member in *dupdate*, otherwise $U[\![u]\!]$(m) cannot be converted into a model.

2. For each entry in u, the entry must have the form `e->du` or `e->!null`, where `du` is another member in *dupdate*. If the entry is of `e->!null` and `e` exists in m, we append "delete element `e`" to the generated update. If the entry is of `e->du` and `e` does not exist in m, we append "insert element `e`" to the generated update. After we converted all entries, we use the next rule to convert all *du*s in the second form.

3. For each entry in `du`, the entry must have the form `p->v` or `p->du'`, where `p` is a property name existing in mm, `v` is a member in *pupdate* and `du'` is a member in *dupdate*. If the entry is of `p->v`, p must be a

single property and we append "change a single attribute" or "change a single reference" to the generated update according to the type of `p`. If the entry is of `p->du'`, p must be a multiple property and we use the next rule to convert `du'`.

4. For each entry in `du'`, the entry must have the form `i->!v`, where `v` is a member of *primitive*. If `v=null` and `i` exists in $m.k.p$, we append "delete an item in a multiple attribute" or "delete an item in a multiple reference" to the generated update. If `v`$\neq$`null` and `i` does not exist in $m.k.p$, we append "insert an item in a multiple attribute" or "insert an item in a multiple reference". If `v`$\neq$`null` and `i` exists in $m.k.p$, we append "replace an item in a multiple attribute" or "replace an item in a multiple reference".

From the above discuss we know that not all dictionary-based updates can be converted back into model updates. As a result, when we merge two updates, we would like to know whether the merged result is still an update on the same model. This property can be ensured by the following theorem.

Theorem 3.4. *If* $m \in_{mmm} mm$, $U[\![u_1]\!](m) \in_{mmm} mm$ *and* $U[\![u_2]\!](m) \in_{mmm} mm$, *we have* $U[\![u_1 \circ u_2]\!](m) \in_{mmm} mm$.

Proof. This theorem can be proved by checking all the five rules are not violated during the merging. In the following we prove two rules, and the other rules can be similarly proved.

1. The meta model and the meta meta model, mm and mmm, are not modified, so they still can be converted back.

2. Because we have $U[\![u_1]\!](m) \in_{mmm} mm$, u_1 does not change the value part of any entry in m into non-dictionaries. Similarly, u_2 does not change the value part of any entry in m into non-dictionaries. In other words, the value part of all entries in u_1 and u_2 are members in *dupdate*. From the merging rule we know that merging two updates in *dupdate* results in an update in *dupdate*, so $u_1 \circ u_2$ does not change the value part of any entry into non-dictionaries.

□

3.6 Related Work

There are many approaches provide lightweight formalization to object models. One notable approach is Alloy [Jac02], which provide a lightweight object

modeling notation that is noticeably more concise than MOF. Moreover, the Alloy language has a kernel, which is defined on three types, and the other parts of the language are mapped to the kernel. Other approaches include providing formal semantics to UML diagrams [BC95, BG98] and providing formal semantics to OCL [CK01]. However, all these formalizations do not take into updates into consideration. Compared to them, our notation mainly focuses on updates and captures data uniquely using one concept: dictionary.

Another branch of work focusing on representing updates on models. This branch of work includes update-representing work [CRP07, CRP08] and model difference work [AP03, AAAN+06, MGH05, XS05] which also needs to represent update. However, those representations are directly defined on MOF models, and the representation is relatively large and complex due to the complexity of MOF standard. For example, Alanen and Porres [AP03]'s representation contains of seven different update operations though many features in MOF, like multiple attributes, are ignored in their models. Compare to their representations, our representation is much simpler but can represent most frequently-used concepts in MOF models.

Chapter 4

Off-Site Synchronization from Unidirectional Transformation

4.1 Motivating Example

Model transformations play an important role in Model-driven architecture(MDA), an approach to software development, which provides a way to organize and manage software artifacts by automated tools and services for both defining models and facilitating transformations between different model types. Writing model transformations is becoming a common task in software development.

ATL [JK06] is a practical batch model transformation language that has been designed and implemented by INRIA to support specifying model transformations that can cover different domains of applications [Tea]. As a simple running example which will be used throughout this chapter, consider the following `UML2Java` transformation in ATL.

```
module UML2Java;
create OUT : Java from IN : UML;
rule Class2Class {
        from u : UML!Class (
                not u.name.startsWith('__draft__')
        )
        to j : Java!Class (
                name <- u.name,
                fields <- u.attrs
        )
}
rule Attribute2Field {
        from a : UML!Attribute
        to f : Java!Field (
                name <- '_' + a.name,
```

```
                type <- a.type
        )
}
```

It uses two rules to transform a simple Unified Modeling Language (UML) model to a simple Java model. Roughly speaking, it maps each UML class whose name does not start with "__draft__" to a Java class with the same name, and each attribute of the class to a field of the corresponding Java class where the field name is the attribute name with an additional prefix "_". For instance, this transformation maps the UML model (in XMI [Obj07])

```
<Class name="Book" description="a demo class">
        <attrs name="title" type="String"/>
        <attrs name="price" type="Double"/>
</Class>
<Class name="__draft__Authors"/>
```

to the following Java model (in XMI):

```
<Class name="Book">
        <fields name="_title" type="String"/>
        <fields name="_price" type="Double"/>
</Class>
```

As mentioned in Chapter 1, despite a bunch of interesting applications of model transformations in software development, there is little work on a systematic method to maintain models at different stages of the software development. Models may be changed in both source and target sides after transformation. For the above example, suppose a group of designers and a group of programmers are working on the models at the same time. The designers may want to add a new attribute `authors` to the `Book` class on the UML model

```
<Class name="Book" description="a demo class">
        <attrs name="title" type="String"/>
        <attrs name="price" type="Double"/>
        <attrs name="authors" type="String"/>
</Class>
<Class name="__draft__Authors"/>
```

while at the same time the programmers may change the field name `_title` to `_bookTitle`, delete the field `_price` from the Java model, and add a new comment to the `Book` class.

```
<Class name="Book">
        <fields name="_bookTitle" type="String"/>
        <comment text="_bookTitle cannot be null"/>
</Class>
```

Now the UML model and the Java model become inconsistent and need to be synchronized. Simply performing the `UML2Java` transformation again is not adequate because the modifications on the Java model will be lost.

There are many challenges in automatically synchronizing these two models related by a model transformation. First, we need an automatic way to derive from a given transformation enough necessary information, forward and backward, such that not only modifications on the source model can be automatically propagated to the target model, but also modifications on the target model can be automatically reflected back to the source model.

Second, the method should be able to deal with general model transformations described in general transformation languages. In fact, the more restriction we impose on a model transformation, the easier but less useful the derived model synchronization process will be. Therefore, we should target a class of practically useful model transformations in order to obtain a useful model synchronization system.

In this chapter, we describe our attempt towards automatically constructing a model synchronization system from a given batch model transformation described in ATL. The main contributions of this chapter can be summarized as follows.

- We propose a new model synchronization approach that can automatically synchronize two models related by a transformation described in ATL, without requiring users to write extra synchronizing code. The model synchronization process satisfies the required properties and ensures correct synchronization of models. Different from the existing bidirectional tree transformations working on high level functional programs [FGM+05, LHT07], our approach works on low level byte codes, which allows us to target more general transformation programs and cover the full ATL.

- We have implemented a model synchronization system by extending the ATL Virtual Machine (VM), the interpreter of ATL byte-code, and have successfully tested several ATL transformation examples in the ATL web site [Tea]. The current prototype system is available at our web site [Xiob].

The rest of this chapter is organized as the follows. We start by clarifying the requirement of model synchronization of two models that are related by a model transformation in Section 4.2. We then show how to automatically synchronize models from a model transformation in Section 4.3 and Section 4.4. We give a case study to illustrate the feasibility of our system in Section 4.5. Finally, we summarize this chapter in Section 4.6.

4.2 Problem Definition

Before we describe our synchronization approach, we must first precisely define the problem so that we know what requirements the approach is going to satisfy.

Our goal is to derive a synchronizer for off-site synchronization of two models from a model transformation related the two models. To well-support off-site synchronization, we would like to generate a state-based synchronizer. To ensure the synchronizer works correctly, the generated synchronizer should satisfy the three synchronization properties. Regarding this, several issues need to be clarified.

First, to avoid the complexity of MOF models, we would like to base our approach on the dictionary-based data (cf. Chapter 3). Because the dictionary-based data and models can be mutually converted, the original transformation can also be executed on the dictionary-based data. The signature of the transformation thus becomes $f : \mathit{value} \longrightarrow \mathit{value}$. From this transformation f, the goal of this chapter is to derive a state-based synchronizer $\mathit{sync}_f : (\mathit{value} \times \mathit{value}) \times (\mathit{value} \times \mathit{value}) \to (\mathit{value} \times \mathit{value})$. The synchronizer takes two original models, two updates models and produces two synchronized models. The type of the synchronizer is specialization of the type in Chapter 2 on two models.

During the conversion from models to dictionaries, we need to assign UIDs to model elements and items in multiple properties. When there are different versions of models, such UIDs show the correspondence between the model elements and items in different versions. However, as recording correspondences is not required by MOF standard, it is possible the correspondences are lost after user edit the model, either directly or through some other applications. In such cases a user of our approach may use model difference approaches [AAAN+06, XS05] to recover the correspondence between model elements and between items in multiple properties.

Second, to define a state-based synchronizer, another important component is a consistency relation R over models. Because the transformation defines a function over two sets of models, a basic idea is to use the model transformation f to define R, where $R = \{\langle m, f(m)\rangle\}$. However, this definition does not allow us to change additional attributes in the target model. For instance, in the running example a `comment` attribute is added to the target model which is not related to the source model by the transformation.

To allow such modification, we define an inclusion relation "$\sqsubseteq$" over values, where $v \subseteq v'$ indicates that v' is only expanded from v. The concrete rule for this relation is shown in Figure 4.1. Then the consistency relation R is defined as $R = \{\langle m, m'\rangle \mid f(m) \sqsubseteq m'\}$.

$$\frac{v, v' \in \mathit{primitive} \wedge v = v'}{v \sqsubseteq v'} \qquad \frac{v, v' \in \mathit{dictionary} \wedge \forall k \in \mathit{dom}(v), v.k \sqsubseteq v'.k}{v \sqsubseteq v'}$$

Figure 4.1: Rules for the inclusion relation

Note this definition of consistency relation allows the target model not only to have additional attributes but also to have new model elements or new items in a multiple property. In other words, the synchronizer is considered correct even if it does not propagate the inserted model elements and inserted item in multiple properties back to the source model. We define the consistency relation in this way because currently there is a limitation in our approach: we cannot reflect inserted items back on the target side into the source model. We define a loose consistency relation so that we can judge the correctness of the synchronizer even if it does not reflect back inserted items.

Third, the PRESERVATION property is defined by a difference operation. Here we define the difference operation using the *find_update* function, as follows. The dictionary-based update is augmented with a model to be an edit update.

$$\mathit{diff}(\langle v, u_1\rangle, \langle v, u_2\rangle) = \langle v, \mathit{find\_update}(u_1, u_2)\rangle$$

To sum up, the goal of this chapter is to derive a synchronizer $\mathit{sync} : (\mathit{value} \times \mathit{value}) \times (\mathit{value} \times \mathit{value}) \to (\mathit{value} \times \mathit{value})$ from a forward transformation $f : \mathit{value} \longrightarrow \mathit{value}$, where the synchronizer satisfies the following three properties.

consistency $\mathit{sync}_f(s, t, s', t')$ is defined $\Longrightarrow \mathit{sync}_f(s, t, s', t') \in R$, where $R = \{\langle m, m'\rangle \mid f(m) \sqsubseteq m'\}$.

stability $\langle s, t\rangle \in R \Longrightarrow \mathit{sync}_f(s, t, s, t) = \langle s, t\rangle$.

preservation

$\mathit{sync}_f(s, t, s', t') = \langle s'', t''\rangle \Longrightarrow$
$\quad \exists u \in \mathit{update} : u \oplus \mathit{find\_update}(s, s') \wedge U[\![u \circ \mathit{find\_update}(s, s')]\!](s) = s''.$
$\mathit{sync}_f(s, t, s', t') = \langle s'', t''\rangle \Longrightarrow$
$\quad \exists u \in \mathit{update} : u \oplus \mathit{find\_update}(t, t') \wedge U[\![u \circ \mathit{find\_update}(t, t')]\!](t) = t''.$

4.3 Backward Propagation of Modifications

To synchronize two models related by a model transformation, we need to propagate modifications between the source model and the target model. The

Table 4.1: The core instructions of ATL byte-code

instructions	description
`push`	push a constant to the stack
`pop`	pop the top of the stack
`store`	store a value into a local variable
`load`	load value from local variable
`if`	branch if the top of the stack is `true`
`iterate`	delimitate the beginning of iteration on collection elements
`enditerate`	delimitate the end of iteration on collection elements
`call`	call a method
`new`	create a new model element
`get`	fetch an attribute of a model element
`set`	set an attribute of a model element

propagation of modifications from the source model to the target model, i.e., the forward propagation, can be carried out by running the model transformation again. However, the propagation of modifications from the target model to the source model, i.e., the backward propagation, cannot get direct help from this transformation.

In this section, we will propose a technique to implement the backward propagation by extending the ATL Virtual Machine (VM). Specifically, we make two extensions to the ATL VM. First, we extend ATL VM so that it works on dictionaries. It takes a dictionary representing the source model and produces a dictionary representing the target model. Second, rather than a normal dictionary, the synchronizer produces an extended dictionary, where the value part of each entry also contains a putting-back function. If later users modify the dictionary, we can use the function on the modified value to reflect back the modification.

4.3.1 ATL Byte-code

An ATL transformation program is first compiled into ATL byte-code and then executed on the ATL VM. The ATL VM, like the Java virtual machine, contains a stack to hold local variables and partial results. An ATL byte-code program consists of a sequence of instructions. A summary of the core ATL instructions is given in Table 4.1. The full specification of ATL byte-code and the ATL virtual machine can be found at the ATL web site [Tea].

As a simple example, the rule `Attribute2Field` in the UML2Java trans-

```
1  push "UML!Attribute"
2  push "IN"
3  call "S.allInstancesFrom(S):QJ"
4  iterate
5  store "a"
6  push "Java!Field"
7  new
8  store "f"
9  load "f"
10 push "_"
11 load "a"
12 get "name"
13 call "S.Concatenate(S):S"
14 set "name"
15 load "f"
16 load "a"
17 get "type"
18 set "type"
19 enditerate
```

Figure 4.2: Byte-code for `Attribute2Field`

formation in Section 4.1 can be written in byte-code, as shown in Figure 4.2. The first three lines return a list containing all `UML!Attribute` instances in the source model. Then instructions between Line 4 and Line 19 iterate on the list. Each instance is stored in a variable `a` (Line 5) and for each instance, a `Java!Field` model element is created (Line 6-7) and stored in a variable `f` (Line 8). Then the `name` attribute of the variable `a` is concatenated with "_" (Line 10-13) and set to the `name` attribute of the variable `f` (Line 9 and 14). The `type` attribute of the variable `a` is retrieved (Line 16 and 17) and set to the `type` attribute of the variable `f` (Line 15 and 18).

4.3.2 Extending the ATL Virtual Machine (VM)

The first extension we made is to execute ATL bytecode instructions on dictionaries. Because models and dictionaries representing models have one-to-one correspondence, the ATL byte-code instructions can be executed on dictionaries by only a few modifications. Table 4.2 shows the modified byte-code instructions. The main change from the original version is to replace collections by dictionaries, replace new model element by empty dictionaries and replace attribute access by accessing entries in dictionaries.

One extra task of executing on dictionaries is that when we create a model element or add an item to a collection, we need to find a proper UID so that we have a key to insert the item. To make the transformation a function,

Table 4.2: The ATL byte-code instructions on dictionaries

instructions	description
`push`	push a constant to the stack
`pop`	pop the top of the stack
`store`	store a value into a local variable
`load`	load value from local variable
`if`	branch if the top of the stack is `true`
`iterate`	delimitate the beginning of iteration on a dictionary
`enditerate`	delimitate the end of iteration on a dictionary
`call`	call a method
`new`	create an empty dictionary
`get`	fetch a value in a dictionary at a predefined key
`set`	replace a value in a dictionary at a predefined key

$$
\begin{array}{ll}
\textit{e-value} & ::= (\textit{e-singlevalue}, \textit{put}, \textit{val}, \textit{loc}) \\
\textit{e-singlevalue} & ::= \textit{primitive} \mid \textit{e-dictionary} \\
\textit{primitive} & ::= \texttt{NUMBER} \mid \texttt{STRING} \mid \texttt{BOOLEAN} \mid \texttt{null} \\
\textit{e-dictionary} & ::= \{\textit{e-entries}\} \\
\textit{e-entries} & ::= \textit{entry} \mid \textit{entry}, \textit{entries} \\
\textit{entry} & ::= \textit{primitive} \texttt{ -> } \textit{e-value}
\end{array}
$$

Figure 4.3: Syntax of dictionary-based data

the UIDs should solely depend on the source model. We find a proper UID based on the UID of the corresponding value in the source model. We shall introduce the detailed technique when we discuss the `new` instruction.

The second extension is to extend dictionaries with putting-back functions. The extended dictionary definition is shown in Figure 4.3. In this definition, each *value*, either the top one or these inside a dictionary, is extended into *e-value* by adding three functions. The function *put* is to be called when the value at the location is changed by user, the function *val* is used to reevaluate the value from the source dictionary and the function *loc* is to return the location of the related source value in the source dictionary. A location is a tuple of keys used to read the value from the top dictionary. For example, in a dictionary {`a->{b->1}`}, the location of `1` is `a->b`.

We convert an update into a sequence of pairs containing a location and

a primitive update. For example, a dictionary update {a->!1, b->{c->!2, d->!null}} is converted into a sequence of three pairs, as follows.

```
   (a,   !1)
(b->c,   !2)
(b->d,   !null)
```

When there is an update u to the target dictionary, we call all *put* functions at the changed locations to put back the update. For each pair, we find the *e-value* at the corresponding location in the target dictionary, pass the primitive update to its *put* function to get a reflected update, and merge all updates to form the result update on the source side. If there are conflicting updates, we report the conflict to the users.

Specifically, we made three extensions to the ATL VM to produce the extended dictionary correctly. The first is that the source dictionary is extended with the three functions when we load the model. The second extension is to extend the semantics of each ATL byte-code instruction, which, if generating new values, also associates the generated values with appropriate extensions. In addition, each `if` instruction also generates a validity-checking function to ensure that its condition is still satisfied after propagating modifications into the source. The third extension is made on the ATL library methods, such as `Concatenate` and `startsWith`, such that the values returned by these methods are also associated with extensions. In most methods and some instructions, new values are created by composing existing values. In those cases, the putting-back functions of new values are built by composing the putting-back functions of existing values. In this way, a call to a putting-back function of a new value will invoke a series of calls to functions of existing values, and will eventually call putting-back functions of values in the source dictionary to update the source dictionary if necessary.

Extending the Source

The model elements and values in the source are extended before transformations. This is done when the ATL VM loads the source dictionary into its runtime environment.

Suppose v is a value at the location of l, and v' is a new value to the original one. Then the extended value is a tuple (v, *put*, *val*, *loc*, where *put*, *val* and *loc* are defined as follows. The functions *put* applies the update to the location in the source model. The `val` function returns the value at the location in the source dictionary. The expression $connect(l, u)$ returns an update that applies u to the location l. The expression $get(d, l)$ returns the value at location l in the dictionary d.

$put(u) = connect(l, u)$
$val(d) = get(d, l)$
$loc = l$

In this extension we extend not only primitive values but also dictionaries. This is because users may apply a primitive update to a dictionary, e.g., applying `!null` to delete a dictionary.

Extending ATL Byte-code Instructions

Some instructions of ATL byte-code do not change or create values or model elements, but move values among different parts (e.g. from a local variable to the stack) of the running environment. The instructions `pop`, `store`, and `load` in Table 4.1 belong to this case. We extend these instructions so that they not only move the original value but also the extensions. Although instructions `get` and `set` decompose dictionaries, we treat them as instructions moving a value from/to an entry of a dictionary and extend the two instructions in the same way.

In the following, we explain how to extend the instructions `iterate`, `enditerate`, `new`, `push` and `if`. The `call` instruction is discussed in the next subsection.

`new`, `iterate` and `enditerate`

The `new` instruction creates new target model elements. However, this instruction provides no information of what source model element or source value corresponds to the new target model element.

To create a collection of target model elements, usually we have to traverse a collection of values or model elements, and create a target model element for each item in the collection. Thus items in the collection can be considered as sources of the target model elements. In the example in Figure 4.2, a set of `Field` model elements is created when traversing the set of `Attribute` model elements in the source. In ATL byte-code the only way to traverse a collection is the `iterate` and `enditerate` instructions.

Based on the above observation, we create a stack called `IterObjs` in the runtime environment to remember the item being iterated. As collections are represented as dictionaries in the extended ATL VM, `iterObjs` store the value parts of the entries being iterated. The `iterate` and `enditerate` instructions are extended to manage this stack. The `iterate` instruction pushes the value being iterated onto the `IterObjs` stack, while `enditerate` pops off the top value from the `IterObjs` stack. If the current value at the

top of the `IterObjs` stack is $\{v', put', val', loc'\}$, the model element created by a `new` instruction has the following extension.

$$put(u) = \begin{cases} put'(\texttt{!null}) & u = \texttt{!null} \\ \mathsf{void} & u = \mathsf{void} \\ \bot & \text{otherwise} \end{cases}$$
$$val(d) = \{\}$$
$$loc = loc'$$

The *put* function checks whether the updated value is `!null` or void. If it is `!null`, i.e., delete the model element, the original value is deleted. If it is void, the function returns void to indicate nothing needs to be modified. If it is neither update, the function is undefined. Here we use $\bot$ to indicate a function is not defined. In such cases, the system should report an error message to the user.

If `IterObjs` is empty, the created model element is considered as a constant that could not be modified.

$$put(u) = \begin{cases} \mathsf{void} & u = \mathsf{void} \\ \bot & \text{otherwise} \end{cases}$$
$$val(d) = \{\}$$
$$loc = \texttt{""}$$

When a new element is created, we need to insert it into a target dictionary at a proper UID. The UID is obtained by concatenating the values returned by `loc` of all values in `IterObjs`, as well as the line number of current instruction. If `IterObjs` is empty or `IterObjs`.*loc* is empty, we incrementally assign a number to replace the location of `IterObjs`, staring from 1. In this way we can ensure each target model element has a unique UID which does not change between different transformations if the `goto` instruction only jumps forward. The code generated by the ATL compiler always ensures that the `goto` instruction only jumps forward.

`push` *cst*

The original semantics of this instruction is to push the constant *cst* onto the top of the operand stack. For example, the instruction at line 10 in Figure 4.2 pushes a constant string `'_'` to the stack. The extended `push` works in a similar way like `new`. When `IterObjs` is empty, the pushed constant is not allowed to be modified.

$$put(u) = \begin{cases} \mathsf{void} & u = \mathsf{void} \\ \bot & \text{otherwise} \end{cases}$$
$$val(d) = cst$$
$$loc = \texttt{""}$$

If the `IterObjs` is not empty, deletion of the pushed constant will result in a deletion of the source value. Let $\{v', put', val', loc'\}$ be the top item in `IterObjs`. The extension on the pushed constant is defined below.

$$put(u) = \begin{cases} put'(\texttt{!null}) & u = \texttt{!null} \\ \mathsf{void} & u = \mathsf{void} \\ \bot & \text{otherwise} \end{cases}$$
$$val(d) = cst$$
$$loc = loc'$$

`if` l

The `if` instruction jumps to the instruction with label l if the value at the top of the operand stack is `true`, otherwise it falls through to the next instruction. We call the value at the top of the stack the *condition value* of the `if` instruction. If we execute the transformation again after backward propagation of modifications, some condition values may become different from their values before backward propagation. This will change the execution paths of the transformation, and probably generate target models in which the user modifications are lost. In our synchronization algorithm, this will violate the PRESERVATION property.

In our running example, a `Java!Class` model element is generated only when the `name` attribute of the `UML!Class` model element does not start with `__draft__`. Suppose a user happens to change the `name` attribute of a `Java!Class` model element to a value starting with `__draft__`. After propagating modifications backward and transforming again, this model element will disappear on the target model.

To prevent such cases, we require that modifications by users should not cause a condition value to be different before and after backward propagation. Our solution is that when executing an `if` instruction, the system will generate a validity-checking function `sat`, and store the function into a set Θ. After backward propagation, this validity-checking function is used to recompute the condition value of this `if` instruction and check whether it is the same as before backward propagation. If not, the system reports an error.

Suppose when executing an `if` instruction, its condition value is (v, put, val, loc). Then the function `sat` generated for this `if` instruction is: $\texttt{sat}(d) =$ if $val(d) = v$ then `true` else `false`.

After backward propagation, the system calls all validity-checking functions in Θ and reports a failure if a function returns `false`.

Extending ATL Library Methods

The `call` instruction is to call ATL library methods. These methods are implemented in Java, not ATL byte-code, so we need to extend them to return extended model elements or extended values. In the following, we will explain how to extend ATL library methods `concatenate` and `startsWith` as examples.

The methods `concatenate` and `startsWith` both take as arguments the first two strings at the top of the operand stack. Suppose the two arguments for both `concatenate` and `startsWith` methods are $(str_1, put_1, val_1, loc_1)$ and $(str_2, put_2, val_2, loc_2)$. For the concatenated string returned by `concatenate`, its extension is defined below:

$$
\begin{aligned}
&put(\mathsf{void}) = \mathsf{void}\\
&put(!v) = \begin{cases} put_1(!head(v,i)) \circ put_2(!tail(v, len(v)-i)) & \\ \qquad\qquad \exists i : put_1(!head(v,i)) \oplus put_2(!tail(v, len(v)-i)) & \\ \bot \qquad \text{otherwise} & \end{cases}\\
&val(d) = \texttt{concatenate}(val_1(d), val_2(d))\\
&loc = loc_1
\end{aligned}
$$

The function `tail`(v', l) extracts the tail substring of string v' of length l; the function `head`(v', l) extracts the leading substring of string v' of length l. The function *concatenate* concatenates two strings. For a modified string, we try to find a proper position to split the string into two parts. If put_1 and put_2 are defined at the two parts, respectively, and the results are compatible, we merge the results and return. As long as strings are separated with constants, we can ensure a reasonable putting-back behavior.

For boolean value b returned by the `startsWith` method, its extension is defined as below.

$$
\begin{aligned}
&put(u) = \begin{cases} \mathsf{void} & U[\![u]\!]b = b \\ \bot & \text{otherwise} \end{cases}\\
&\texttt{val}(d) = \texttt{substr}(\texttt{val}_1(d), \texttt{val}_2(d))\\
&loc = loc_1
\end{aligned}
$$

Boolean values returned by the `startsWith` method cannot be modified, but these values can be reevaluated by calling the `val` function. The `substr` checks whether the first argument is the leading substring of the second argument.

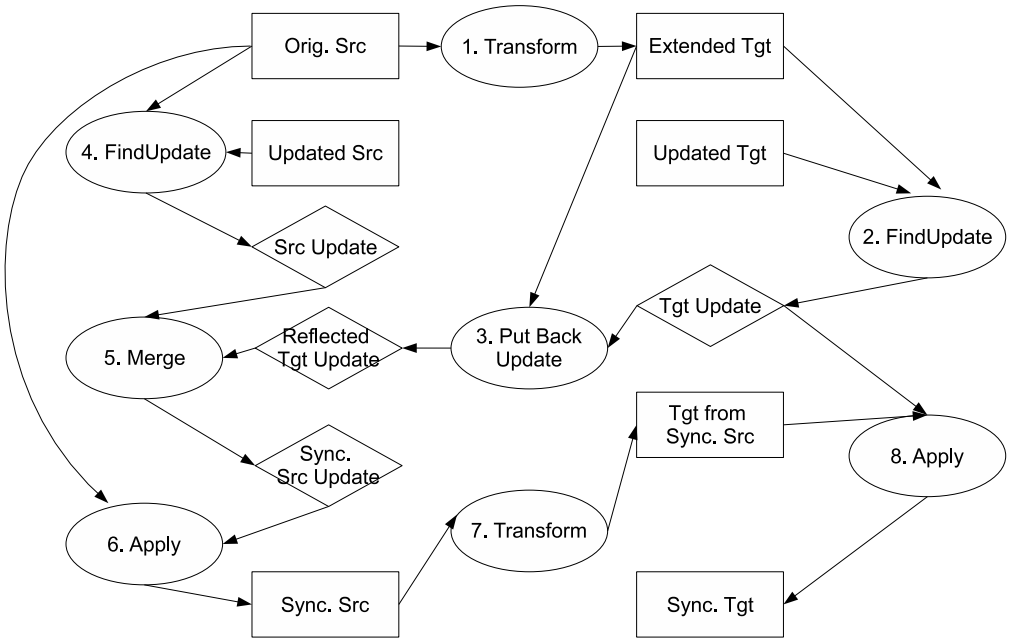

Figure 4.4: Overview of synchronization algorithm

4.4 Synchronization

In this section we show how to realize our model synchronization process (as defined in Section 4.2)

$$sync_f : (value \times value) \times (value \times value) \to value \times value$$

based on a given transformation $f : value \to e\text{-}value$ which shows how to map a source dictionary to a extended target dictionary, where the extension (in Section 4.3) shows how to reflect updates to the target dictionary back to the source dictionary. We shall illustrate our synchronization algorithm by our running example, and explain why our synchronization satisfies the properties in Section 4.2.

4.4.1 Synchronization Algorithm

An overview of our synchronization algorithm is shown in Figure 4.4. The synchronization algorithm takes as input

- the original source model `Orig.Src`(represented as a dictionary. The below is the same),
- the modified source model `Updated Src`,

- the modified target model `Updated Tgt`, and
- the transformation f that can generate a extended target model from a source model

and returns as output

- the synchronized source model `Sync.Src`, and
- the synchronized target model `Sync.Tgt`.

The algorithm does not use the original target model, because the needed information can be reproduced from the source model.

The basic idea of the algorithm is: first put back the modifications on the target into the source and merge with modifications on the source, then reproduce the target model. The synchronization process in all has seven steps, which will be informally illustrated through our running example of UML2Java in Section 4.1, where all inputs have been given.

Step 1: Generating the extended target model

This step simply applies the transformation to the original source model to obtain the extended target model `Tgt0`. For our UML2Java example, we first have the following source dictionary. For simplicity, we represent the type by strings.

```
{1->{name->"Book",
     description->"a demo class",
     attrs->{5->3, 6->4},
     __type->"Class"},
 2->{name->"__draft__Authors"
     __type->"Class"},
 3->{name->"title",
     type->"String",
     __type->"Attribute"},
 4->{name->"price",
     type->"Double",
     __type->"Attribute"}}
```

Then we transform it into the an extended target model. The following code shows the model, where the extensions are omitted and the concatenated UIDs are replaced by unique numbers for clarity.

```
{7->{name->"Book",
     attrs->{10->8, 11->9},
     __type->"Class"},
 8->{name->"_title",
     type->"String",
     __type->"Field"},
 9->{name->"_price",
     type->"Double",
     __type->"Field"}}
```

Step 2: Compare the two target models to get the target update

We use the function `findUpdate` we defined in Chapter 3 to find the updates from one dictionary to another dictionary. Given the following dictionary representing the updated target model,

```
{7->{name->"Book",
     attrs->{10->8},
     comment->12,
     __type->"Class"},
 8->{name->"_bookTitle",
     type->"String",
     __type->"Field"},
 12->{text->"_bookTitle cannot be null",
                  __type->"Comment"}}
```

the function `findUpdate` returns the update as follows.

```
{7->{attrs->{11->!null},
     comment->!12},
 8->{name->!"_bookTitle"},
 9->!null,
 12->{text->!"_bookTitle cannot be null",
                  __type->!"Comment"}}
```

It should be noted that adding the comment to the class changes two locations. First a new `Comment` model element needs to be inserted. Second the `comment` attribute of the class needs to be modified from `null` to the UID to the comment. Same as this, the deletion of the `__price` attribute needs modifying two locations.

Step 3: Use the extension to reflect back updates

We apply the technique described in Section 4.3 to put back updates on the target side back to the source model.

```
{1->{attrs->{6->!null}},
 3->{name->!"_bookTitle"},
 4->!null}
```

Note that the inserted comment on the target model is not reflected to the source model. This is because the given transformation does not write the `comment` attribute nor does it create any `Comment` model elements. There is no putting-back function generated for the inserted `comment` model element and the changed `comment` attribute.

Step 4: Compare the source models to get the source update

This step is similar to Step 2 except that it is applied to the source side instead of the target side. In our example, the updated source model is represented by the following dictionary.

```
{1->{name->"Book",
     description->"a demo class",
     attrs->{5->3, 6->4, 14->13},
     __type->"Class"},
 2->{name->"__draft__Authors"
     __type->"Class"},
 3->{name->"title",
     type->"String",
     __type->"Attribute"},
 4->{name->"price",
     type->"Double",
     __type->"Attribute"},
 14->{name->"authors",
     type->"String",
     __type->"Attribute"}}
```

After comparing, the result update is as follows.

```
{1->{attrs->{14->!13}},
 14->{name->!"authors",
     type->!"String",
     __type->!"Attribute"}}
```

Step 5: Merging the two updates

Now we have the updates to the source and the updates reflected from the target side, we can merge them to get the synchronized update that contains updates from both side. We first determine whether the two updates are compatible. We report an error message to the user when the updates conflict, and merge the updates when they are compatible.

Since the two updates in our example are compatible, we can merge them, and the result is as follows.

```
{1->{attrs->{6->null, 14->!13}},
 3->{name->!"_bookTitle"},
 4->!null,
 14->{name->!"authors",
     type->!"String",
     __type->!"Attribute"}}
```

Step 6: Apply the merged update to the original source model

As we have the synchronized source update, we can apply it to original source model to get the synchronized source. Note here applying to the original source and applying to the updated source should have the same effect because the user update on the source side is included in the synchronized update.

```
{1->{name->"Book",
     description->"a demo class",
```

```
     attrs->{5->3, 14->13},
     __type->"Class"},
 2->{name->"__draft__Authors"
     __type->"Class"},
 3->{name->"booktitle",
     type->"String",
     __type->"Attribute"},
 14->{name->"authors",
     type->"String",
     __type->"Attribute"}}
```

Step 7: Reproduce the target model from the synchronized source

The result source model contains synchronized update from both side, and we would like to also propagate the synchronized update to the target side. To do that, we perform the transformation again to produce a target model from the synchronized source model.

```
{7->{name->"Book",
     attrs->{10->8, 16->15},
     __type->"Class"},
 8->{name->"_bookTitle",
     type->"String",
     __type->"Field"},
 15->{name->"_authors",
      type_>"String",
      __type->"Field"}
```

Step 8: Apply the target update to the reproduced target model

The target model produced in the last step now should contain the update on the source model and the update that have been reflected from the target model to the source. Yet the update not reflected from the target model to the source model is missing. To merge such modifications, we further apply the target update on the model to get the synchronized target model.

```
{7->{name->"Book",
     attrs->{10->8, 16->15},
     comment->12,
     __type->"Class"},
 8->{name->"_bookTitle",
     type->"String",
     __type->"Field"},
 15->{name->"_authors",
      type_>"String",
      __type->"Field"}
 12->{text->"_bookTitle cannot be null",
                  __type->"Comment"}}
```

4.4.2 Properties

It is worth remarking that our synchronization system satisfies the properties we defined. In the following we discuss why these properties are satisfied.

First, our synchronization system satisfies CONSISTENCY. In Step 6, we produce a target model from the synchronized source model. If no existing location in the target model is modified in Step 7, the two models are consistent according to the relation R. Because all condition expressions will be evaluated to the same value, all reflected updates will be produced again following the same path. Consequently for any existing locations, the value at the location and the value to be updated should be the same, and thus no existing locations will be changed in Step 7.

Second, our synchronization system satisfies PRESERVATION. This is because on either side, the user update is eventually applied to the synchronized models before they are produced.

Third, our synchronization system satisfies STABILITY. If no model is modified, no update will be discovered by *find_update*, and thus no update will be reflected. Consequently, the result models are the same as input.

4.5 A Case Study

Our system has been successfully applied to several ATL examples listed at ATL web site [Tea]. In this section, we will use one of them to help demonstrate our approach described before. This example is about a transformation from class models to relational database models and is widely used in the literature of model transformations [LDGR04]. By this case study, we can see after users write an ATL transformation, the consistency of the source and target models can be automatically maintained by our system when they are evolved, and the synchronization procedure exhibits some interesting properties.

To run this example, we need the ATL code, the source model as well as the source and target metamodels. Due to space limitation, only the source model is shown in Figure 4.5, and other files can be found at ATL web site [Tea]. This source model includes two classes `Person` and `Family`, and two Datatypes `String` and `Integer`. Each class has a collection of attributes `attr`, which can be single-valued or multi-valued. The attribute `ID` in each model element is added by us to identify model elements.

In this example, a class will be transformed into a table, and a datatype into a type in the relational table model. Each attribute in a class, if it is single-valued, will lead to a column in the corresponding table, otherwise a

```
 0: <?xml version="1.0" encoding="ISO-8859-1"?>
 1: <xmi:XMI xmi:version="2.0" xmlns="Class"
        xmlns:xmi="http://www.omg.org/XMI" >
 2:   <Class name="Person" ID="1">
 3:     <attr name="firstName" ID="5" type="3"/>
 4:     <attr name="closestFriend" ID="6" type="1"/>
 5:     <attr name="emailAddresses" ID="7"
 6:          multiValued="true" type="3"/>
 7:   </Class>
 8:   <Class name="Family" ID="2">
 9:     <attr name="name" ID="8" type="3"/>
10:     <attr name="members" ID="9"
11:          multiValued="true" type="1"/>
12:   </Class>
13:   <DataType name="String" ID="3"/>
14:   <DataType name="Integer" ID="4"/>
15: </xmi:XMI>
```

Figure 4.5: A source model in XMI

new table will be generated for it. And each table generated from a class also includes a key column. The ATL web site has the detailed description for this transformation. The target model generated by this transformation is given in Figure 4.6.

In the following, we will give several experiments to show the synchronization results of our system. Each experiment is to demonstrate some properties that our approach has.

In the first experiment, we invoke the synchronization procedure without changing the source model and the target model. After synchronization, the resulting source and target models are still the same as the original ones, embodying the property of STABILITY.

In the second experiment, change `Person_emailAddresses` in Line 14 to `Individual_emailAddresses` and change the type of `emailAddresses` in Line 17 from `"3"` to `"4"`, that is, the type changes to `Integer`. In addition, we change the source model by removing the line 4, that is, the attribute of `closestFriendId` in class `Person` is deleted. After synchronization, the result source model keeps the attribute of `closestFriendId` deleted while the class name in line 2 changes from `Person` to `Individual` and the type of `emailAddresses` changes to `"4"`, that is, changes to type `Integer`; the result target model has `closestFriend` originally in Line 10 deleted, the type of `emailAddresses` remaining `Integer` and all occurrences of the string "Person" changing to "Individual", in other words, the table name in Line 7 changes to `Individual`, the table name in Line 14 remains `Individual_emailAddresses`, and the column name in Line 15 changes to

```
 0: <xmi:XMI xmi:version="2.0" xmlns="Relational"
 1:          xmlns:xmi="http://www.omg.org/XMI" >
 2:   <Table name="Family" ID="2" key="1002">
 3:     <col name="objectId" ID="1002" keyOf="2"
 4:          type="4"/>
 5:     <col name="name" ID="8" type="3"/>
 6:   </Table>
 7:   <Table name="Person" ID="1" key="1001">
 8:     <col name="objectId" ID="1001" keyOf="1"
 9:          type="4"/>
 9:     <col name="firstName" ID="5" type="3"/>
10:     <col name="closestFriendId" ID="6"
11:          type="4"/>
11:   </Table>
12:   <Type name="String" ID="3"/>
13:   <Type name="Integer" ID="4"/>
14:   <Table name="Person_emailAddresses" ID="7">
15:     <col name="PersonId" ID="1007" type="4"/>
16:     <col name="emailAddresses" ID="1008"
17:          type="3"/>
18:   </Table>
19:   <Table name="Family_members" ID="9">
20:     <col name="FamilyId" ID="1009" type="4"/>
21:     <col name="membersId" ID="1010" type="4"/>
22:   </Table>
23: </xmi:XMI>
```

Figure 4.6: A target model in XMI

`IndividualID`. This experiment demonstrates PRESERVATION and CONSISTENCY.

In the third experiment, we change the string `objectId` in the line 8 into `objId`. This string comes from the transformation code, not from the source model. The system reports a failure during synchronization. This shows that our system has the ability to detect and report inappropriate modifications.

The fourth experiment is to demonstrate an interesting property of our system: applying and synchronizing two updates in sequence is the same as composing them and synchronizing once. We first change the table name in the target model and delete the attribute in the source model, synchronize, then change the type of `emailAddresses` in the target model and synchronize again. After the two synchronization processes, we get the same result as the second experiment.

4.6 Summary

In this chapter we have reported our attempt to automatic construction of model synchronization systems under the condition that the models to be synchronized are related by model transformations. In our approach, if a model transformation from one model to another is given, these two models can be synchronized for free without writing extra code. The key contributions of our approach are two folds: an automatic derivation of putting-back functions from execution of a model transformation, and a new synchronization algorithm with clear synchronization semantics. We have implemented all the ideas as a system, SyncATL, for synchronizing models related by ATL transformations. The experimental results are encouraging; several nontrivial examples in the ATL Web site have been successfully tested.

One limitation of our current system is that it cannot deal well with insertions on the target side; although the system works well when the inserted value is not related to the source, it cannot reflect insertion when the inserted value should also cause source changes. This is one of our future work.

Chapter 5

Off-Site Synchronization from Bidirectional Transformation

In Chapter 4 we derive a synchronizer from a unidirectional transformation. In practice, two models are not always related by a unidirectional transformation. Sometimes they are related by a bidirectional transformation. Bidirectional model transformation approaches [Obj08, SK08b] provide bidirectional model transformation languages, which are used to describe the relation between the two models symmetrically. Programs in these languages are used not only to transform models from one format into another, but also to update the other model automatically when a model is updated by users.

Stevens [Ste07] formalizes a bidirectional model transformation as two functions. If M and N are meta models and $R \subseteq M \times N$ is the consistency relation to be established between them, a *bidirectional model transformation* consists of two functions:

$$\overrightarrow{R} : M \times N \to N$$
$$\overleftarrow{R} : M \times N \to M$$

Given a pair of models $(m, n) \in M \times N$, function $\overrightarrow{R}$ changes n to make it consistent with m. Similarly, $\overleftarrow{R}$ changes m in accordance with n. Many bidirectional model transformation languages fall into this model; typical languages include Query/View/Transformation relations (QVT-R) [Obj08] and TGGs [SK08b].

Bidirectional transformation synchronizes two models by propagating updates from one to the other and vice versa. However, in some cases, models m and n may be simultaneously updated before a bidirectional transformation can be applied. For example, a designer could be working on the design model at the same time a programmer is working on the implementation

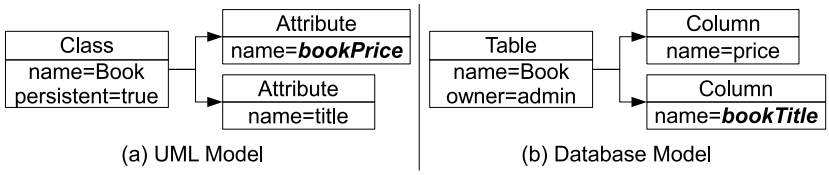

Figure 5.1: Non-conflicting parallel updates

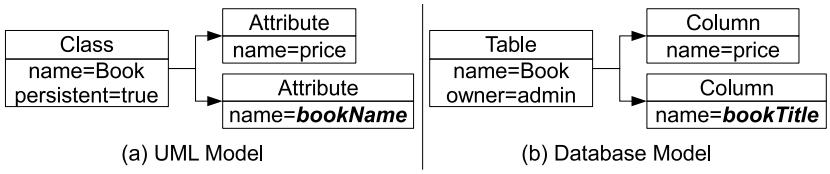

Figure 5.2: Conflicting parallel updates

model. Applying the transformation in either direction will result in the loss of updates on the target side.

Because of the large number of available bidirectional transformation languages and existing transformation programs, it would be preferable if we could synchronize parallel updates using existing bidirectional transformations. One basic idea is to sequentially apply the two updates and interleave them with two transformations. Let us consider the running example we used in Chapter 2. Suppose a user changes the `price` attribute into `"bookPrice"` in the UML model and another user changes the `title` column into `"bookTitle"` in the database model at the same time, as shown in Figure 5.1. We can assume that the `title` column in the database model is changed first and perform a backward transformation to change the `title` attribute in the UML model. Then, we change the `price` attribute into `"bookPrice"` in the UML model and perform a forward transformation to change the `price` column in the database model.

However, there are two problems in implementing this idea. First, as with bidirectional transformation, we do not want to require users to track updates. We thus need to identify which part of the updated UML model was changed so that we can later apply the update to the result of the backward transformation. Second, the sequential application of updates does not deal with conflicts. Figure 5.2 shows an example of conflicting updates where the `title` attribute and the `title` column are changed to different values. If we transform backward and then go forward again, we will lose the update to the database model. A preferable synchronization procedure would detect such conflicts and advise the user.

In this chapter we propose a new approach based on the idea of sequen-

tially applying parallel updates. We use commonly used model difference approaches [AP03, MGH05, AAAN+06] to solve the two problems above. We design an algorithm that use model difference approaches to wrap any bidirectional transformation into a synchronizer for parallel updates. The synchronizer takes the two original models and two updated models as input and produces two new models in which the updates are synchronized. The idea of this algorithm is inspired by the algorithm in Section 4.4, but it is significantly modified to wrap a bidirectional transformation.

The main contributions of this chapter can be summarized as follows.

- We propose an algorithm that can wrap any bidirectional model transformation and any model difference approach into a synchronizer supporting parallel updates. It treats the bidirectional model transformation and the model difference approach as black boxes and does not require the user to write additional code. In addition, this algorithm is adaptable in the handling of conflicts and updating failures.

- We prove that, for any bidirectional transformation satisfying the CORRECTNESS and HIPPOCRATICNESS properties [Ste07], the synchronizer satisfies CONSISTENCY, STABILITY, and PRESERVATION, ensuring correct and predictable synchronization behavior.

- We have used our algorithm to design an architecture-based runtime management framework. The application not only shows that our algorithm works well in practical cases but also is a significant contribution to architecture-based runtime management.

5.1 Background: Properties of Bidirectional Model Transformation

The definition of bidirectional transformation describes only the input and output of a transformation; it does not constrain the behavior of the transformation. Stevens [Ste07] proposes three properties that a bidirectional transformation should satisfy to ensure that models are transformed in a reasonable way. In this paper, however, we require only that a bidirectional transformation satisfies two of them (CORRECTNESS and HIPPOCRATICNESS) because the last property, UNDOABILITY, would prohibit many practical transformations.

The first property, CORRECTNESS, ensures that a bidirectional transformation does something useful. Given two models, m and n, the forward

and backward transformations must establish consistency relation R between them.

Property 7 (CORRECTNESS)**.**
$\forall m \in M, n \in N : \quad R(m, \overrightarrow{R}(m, n))$
$\forall m \in M, n \in N : \quad R(\overleftarrow{R}(m, n), n)$

The second property, HIPPOCRATICNESS, prevents a bidirectional transformation from doing something harmful. Given two consistent models m and n, if neither model is modified, the forward and backward transformations should modify neither model.

Property 8 (HIPPOCRATICNESS)**.**
$R(m, n) \Longrightarrow \overrightarrow{R}(m, n) = n$
$R(m, n) \Longrightarrow \overleftarrow{R}(m, n) = m$

The last property, UNDOABILITY, means that a performed transformation can be undone. Suppose there are two consistent models, m and n. A user, working on the M side, updates m to m' and performs a forward transformation to propagate the updates to the N side. Immediately after the transformation, he realizes that the update is a mistake. He modifies m' back to m and performs the forward transformation again. If the bidirectional transformation satisfies UNDOABILITY, the second transformation will produce the exact n to cancel the previous modification on the N side.

Property 9 (UNDOABILITY)**.**
$\forall m' \in M : \quad R(m, n) \Longrightarrow \overrightarrow{R}(m, \overrightarrow{R}(m', n)) = n$
$\forall n' \in N : \quad R(m, n) \Longrightarrow \overleftarrow{R}(\overleftarrow{R}(m, n'), n) = m$

While UNDOABILITY makes sense in some situations, here we do not require bidirectional transformations to satisfy this property because UNDOABILITY imposes a strong requirement on the consistency relation, R, and prohibits many useful transformations. One example is the UML-to-database transformation. If we change the `persistent` property of a class to `false` in the UML model, a forward transformation will delete the corresponding table in the database model. However, if we modify the property back to `true`, it is not possible for the forward transformation to recover the original table because the value of the `owner` property has been lost. This problem cannot be solved from the transformation alone. To satisfy UNDOABILITY, we must change the meta model of the database to store all deleted `owner` properties, which would be impossible and unnecessary in many cases.

5.2 Approach

In this section we introduce our approach. We first discuss some needed techniques for our algorithm. We shall create a three-way merger using a model difference operation, and discuss how to test the preservation relation. Based on them, we present our algorithm to wrap a bidirectional transformation into a synchronizer.

5.2.1 Three-Way Merger

With the model difference function and the union operator, we can construct a three-way merger of models. A *three-way merger* takes one original model and two independently updated copies of the model and produces a new model in which the updates to the two copies are merged. Three-way mergers are widely used in many distributed systems, like the Concurrent Versions System (CVS), and in the `diff3` command [KKP07] in Unix. Given an original model m_o and two independently modified copies, m_a and m_b, a three-way merger is a partial function defined as the following.

$$\mathit{merge}(m_o, m_a, m_b) = (\mathit{diff}\,(m_o, m_a) + \mathit{diff}\,(m_o, m_b)).\mathit{post}$$

If $(\mathit{diff}\,(m_o, m_a) + \mathit{diff}\,(m_o, m_b))$ is not defined, *merge* is not defined, indicting there are conflicts between m_a and m_b.

One natural result is that a three-way merger will always preserve the updates in both models.

Theorem 5.1. *If $m_c = \mathit{merge}(m_o, m_a, m_b)$, then m_c preserves the update from m_o to m_a and the update from m_o to m_b.*

Proof. From the definition of *merge* we get $(\mathit{diff}\,(m_o, m_a) + \mathit{diff}\,(m_o, m_b)).\mathit{post} = m_c$. From the commutativity of $+$, we get $(\mathit{diff}\,(m_o, m_b) + \mathit{diff}\,(m_o, m_a)).\mathit{post} = m_c$. Because th ere exists $\mathit{diff}\,(m_o, m_b)$, from the first formula, we have that m_c preserves the update from m_o to m_a. Similarly, from the second formula, we have that m_c preserves the update from m_o to m_b. □

5.2.2 Testing Preservation

The definition of preservation relation on models (see Section 2.4) gives us a basic method for testing whether model m_c preserves an update from model m_o to model m_o. However, to actually test it, we must iterate all possible updates starting from m_o, which is not possible in practice. What we need is an efficient procedure for quickly testing the preservation of three models.

Table 5.1: Modification operations

Operation	Description
new(e, t)	create a new element e of type t
delete(e, t)	delete element e of type t
set(e, f, v_o, v_n)	set an attribute f of element e from v_o to v_n
insert(e, f, e_t)	add a link from $e.f$ to e_t for an unordered reference f
remove(e, f, e_t)	remove a link from $e.f$ to e_t for an unordered reference f
insertAt(e, f, e_t, i)	add a link from $e.f$ to e_t at index i for an ordered reference f
removeAt(e, f, e_t, i)	remove a link from $e.f$ to e_t at index i for an ordered reference f

Such an efficient testing procedure is difficult to find in general. However, given a specific model difference approach, it is often possible to define an efficient testing procedure in accordance with the update operations considered in the difference approach. In the following we show how to efficiently test preservation for Alanen et al.'s [AP03] model difference approach as an example.

Alanen et al. consider an update as a sequence of update operations, and they define seven types of operations, as shown in Table 5.1. In their work, they assume that each element has a universally unique identifier (UUID) that does not change across versions. Under this assumption, we can easily identify and match model elements in different versions of objects. In addition, they consider limited types of features on the models. Features can be classified as attributes that store primitive values and references that store links to other model elements. They assume that all attributes are single features (can contain only one value) and that all references are multiple features (can contain more than one feature, either ordered or unordered).

To test whether an update from m_o to m_a is preserved in m_c, we first use the difference operation to get the update $\delta_{oa} = \mathit{diff}(m_o, m_a)$. Then we examine m_c for each update operation in δ_{oa}. If we find that an operation such that the union of any operation and this operation cannot reach m_c from m_o, we report a violation of preservation. The detailed rules for examining the update operations can be found in Table 5.2.

For example, suppose the `price` attribute in Figure 2.1(a), the `bookPrice` attribute in Figure 5.1(a), and the `price` attribute in Figure 5.2(a) share UUID e_p. The difference of Figure 2.1(a) and Figure 5.1(a) is thus an update containing one update operation: set(e_p, `name`, `"price"`, `"bookPrice"`). This

Table 5.2: Testing of preservation

Operation in δ_{oa}	Preservation condition
new(e, t)	e exists in m_c, and all features of e are the same as m_a
delete(e, t)	e does not exist in m_c
set(e, f, v_o, v_n)	e exists in m_c, and $e.f$ is the same value as v_n
insert(e, f, e_t)	e exists in m_c, and a link to e_t exists in $e.f$
remove(e, f, e_t)	e does not exist in m_c, or a link to e_t does not exist in $e.f$
insertAt(e, f, e_t, i)	e exists in m_c, a link to e_t exists in $e.f$, and the inserted links have their order in m_a preserved in m_c for all insertAt operations on the feature
removeAt(e, f, e_t, i)	always preserved (as deleted links can be inserted back)

update is not preserved in Figure 5.2(a) because the rule for `set`(e, f, v_o, v_n) is violated: e_p.`name` has a value of `"price"` and is different from `"bookPrice"` in Figure 5.2(a).

5.2.3 Algorithm for Wrapping Bidirectional Transformation

Now we have a three-way merger and can test the preservation of updates. Let us use them to wrap a bidirectional transformation into a synchronizer for parallel updates. The basic idea is first to convert the model from the N side to the M side using backward transformation, then to use the three-way merger to reconcile the updates, and transform back using the forward transformation. The detailed algorithm is shown in Figure 5.3.

We explain the algorithm using the UML-to-database example. Initially, we have the two models in Figure 2.1, which correspond to m_o and n_o in our algorithm. Users modify the two models into the models in Figure 5.1, which correspond to m_a and n_b in our algorithm. We use different subscripts to show different updates, where a represents the update on m_o and b represents the update on n_o. The four models together comprise the algorithm input.

The first step of our algorithm is to invoke backward transformation $\overleftarrow{R}$ to propagate the updates made to n_b to m_o, resulting in m_b. The result is shown in Figure 5.4(a). The attribute name is changed from `"title"` to `"bookTitle"`.

Now we have model m_a containing update a and model m_b containing

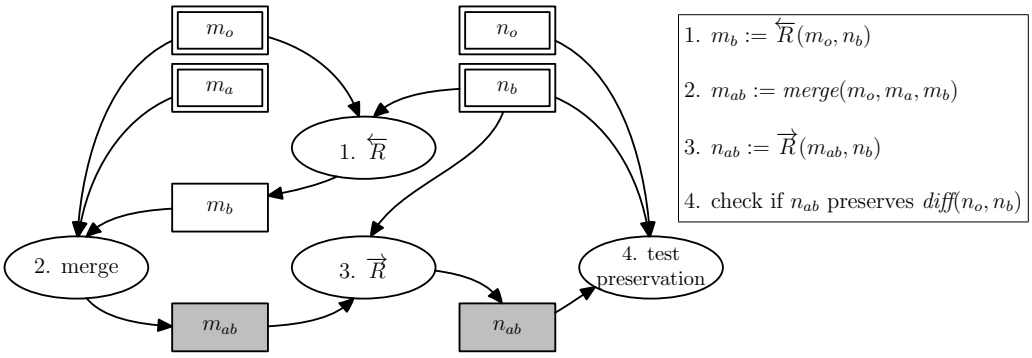

Figure 5.3: Synchronization algorithm

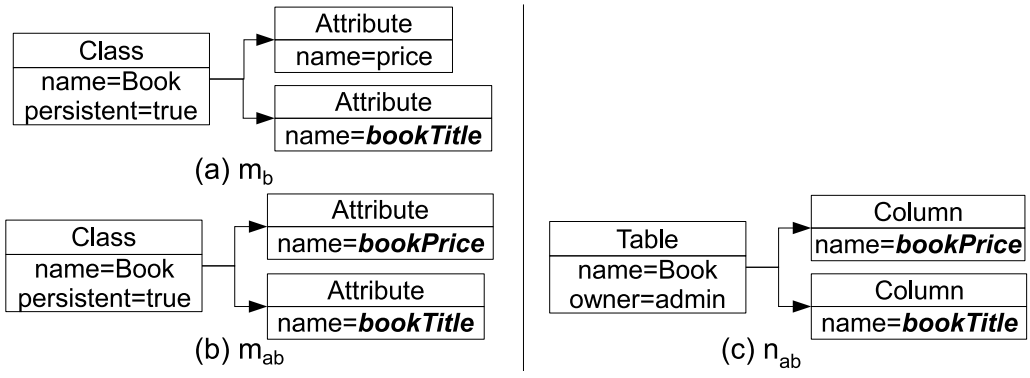

Figure 5.4: Execution of algorithm

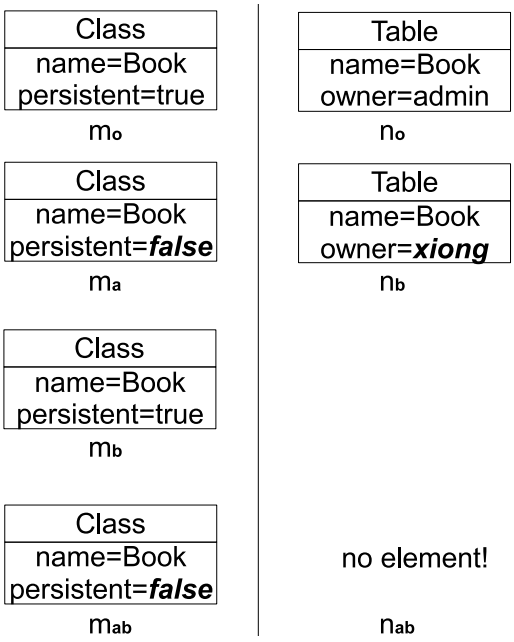

Figure 5.5: Violating PRESERVATION

update b. The second step is to use the three-way merger we constructed in the last section to merge the two updates and produce synchronized model m_{ab} on the M side. The result is shown in Figure 5.4(b). The model has both attributes changed; i.e., it contains updates from both sides. If the updates to the two models conflict, the three-way merger detects the conflict and reports an error.

The third step is to use forward transformation $\overrightarrow{R}$ to produce synchronized model n_{ab} on the N side. The result is shown in Figure 5.4(c). This model also contains updates from both sides, with both columns changed.

Now we have two synchronized models to which the updates have propagated. It looks as if we have performed enough steps to finish the algorithm. However, the above steps do not ensure the detection of all conflicts and may lead to violation of PRESERVATION due to the heterogeneousness of the two models.

To see how this can happen, let us consider the example in Figure 5.5. Initially we have only one class and one table, and they are consistent. Then suppose that a user changes the `persistent` feature of the class to `false` and changes the owner of the table to `"xiong"`. Because the `owner` feature is not related to the UML model, the backward transformation changes nothing, and m_b is the same as m_o. The three-way merger detects no updates in m_b and produces a model that is the same as m_a. Finally, we perform the forward transformation, and the table is deleted because of the change to the `persistent` feature. However, as the user has modified a feature of the

table, so he or she will expect to see the existence of the table in the final result. The input models contain conflicting updates, but the synchronization process does not detect them.

This kind of violation is caused by the heterogeneity of M and N. Due to the heterogeneity, not all updates to N are visible on the M side. As the three-way merger only works on the M side, it cannot detect such invisible conflicts.

To capture such conflict, we add an additional step, preservation testing, to the end of the algorithm. It is shown as the fourth step in Figure 5.3. This step uses the preservation testing procedure described in Section 5.2.2 and checks whether the update from n_o to n_b is preserved in n_{ab}. If not, the algorithm reports an error.

The models used in Figure 5.4 and Figure 5.5 are simply examples. The actual execution depends on the bidirectional transformation and the model difference approach used in the synchronization and may differ from the above execution. Nevertheless, whatever bidirectional transformation and model difference approach we choose, our algorithm ensures the three synchronization properties: CONSISTENCY, STABILITY, and PRESERVATION.

Theorem 5.2. *If the bidirectional transformation satisfies* CORRECTNESS*, the synchronization algorithm satisfies* CONSISTENCY*.*

Proof. Because $\overrightarrow{R}(m_{ab}, n_b) = n_{ab}$, we have $R(m_{ab}, n_{ab})$. □

Theorem 5.3. *If the bidirectional transformation satisfies* HIPPOCRATICNESS*, the synchronization algorithm satisfies* STABILITY*.*

Proof. If we have $m_o = m_a$ and $n_o = n_b$, we have $R(m_o, n_b)$. Because of HIPPOCRATICNESS, we have $m_b = \overleftarrow{R}(m_o, n_b) = m_o$. Because of STABILITY OF MODEL DIFFERENCE, $m_{ab} = merge(m_o, m_a, m_b) = (\mathit{diff}(m_o, m_a) + \mathit{diff}(m_o, m_b)).\mathit{post} = (\mathit{diff}(m_o, m_o) + \mathit{diff}(m_o, m_o)).\mathit{post} = m_o$. On the other hand, $n_{ab} = \overrightarrow{R}(m_{ab}, n_b) = \overrightarrow{R}(m_o, n_o) = n_o$, and the preservation testing always passes because of the existence of identity update. □

Theorem 5.4. *The synchronization algorithm always satisfies* PRESERVATION*.*

Proof. Because of Theorem 5.1, the update on the M side is preserved. Because of the last preservation test, the update on the N side is preserved. □

It is worth noting that our algorithm works even if the bidirectional transformation does not satisfy CORRECTNESS or HIPPOCRATICNESS. This has practical value because many bidirectional transformation languages in practice do not guarantee the properties [Ste07]. In such cases, the algorithm still

produces output but does not guarantee the corresponding synchronization properties (CONSISTENCY or STABILITY).

Bidirectional transformations are symmetrical, so we can also implement this algorithm in the opposite direction. We can start a forward transformation first, merge models on the N side, perform a backward transformation, and check preservation on the M side. Implementing the algorithm in both directions can guarantee the three properties. However, due to the heterogeneity of M and N, it is possible that different directions may produce different results for some input. The difference is related to the specific bidirectional transformation approach and the difference approach used in the algorithm, and we do not discuss it in this paper.

5.3 Extending the Algorithm

In the previous section we discuss the basic algorithm for wrapping a bidirectional transformation into a synchronizer, but several issues are needed to be discussed to put the algorithm into piratical use. In this section we discuss two issues and extend the algorithm to deal with the two issues.

The first issue is about handling conflicts. The algorithm presented in the previous section only reports the existence of a conflict but provide little support in resolving the conflict. A desirable support of conflict resolution should represent the location of conflicts and the reason of conflicts to users and provide options for users to choose from. However, this kind of support requires us to treat the bidirectional transformation and the updates as white boxes to locate the location and the reason of conflicts, which is different from our goal of treating them as black boxes. However, although it is difficult to provide full support of conflict resolution, some simplified strategies can be provided for specific situations. Here we consider one simplified strategy: overwriting all conflicting updates on one model with updates on the other model. In many cases, the two models being synchronized are not symmetric. One model may have high priority than the other and the updates on one model can override the updates on the other. One example is a management system where the updates from some human administrator on the management model can overwrite the system changes on the system being managed.

As the union operator is undefined when the input updates conflict, we need other operators to deal with conflicts. In this section we define a sequential composition operator over updates. The operator is similar to the merge operator on the dictionary update. A sequential composition operator defined on meta model M is a total function $\circ : \Delta_M \times \Delta_M \to \Delta_M$, where a

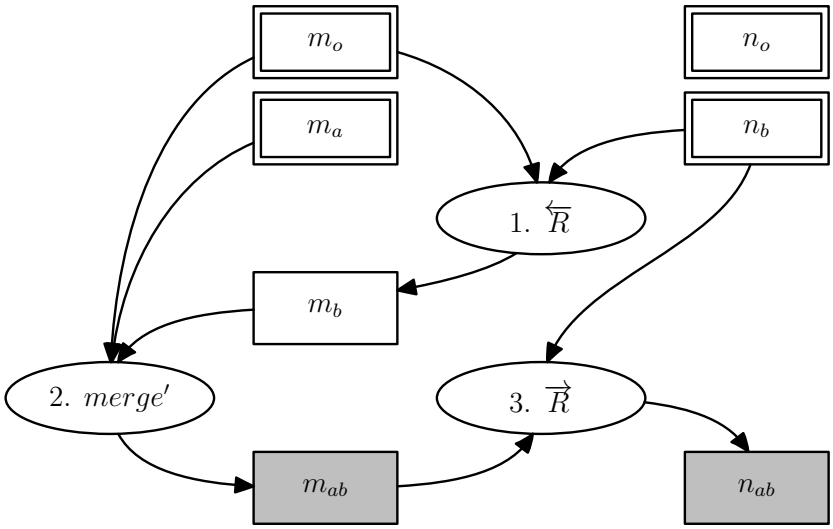

Figure 5.6: The synchronization algorithm for conflict resolving

sequential composition of m_1 and m_2, denoted as $m_2 \circ m_1$, is an update that is considered to have the same effect of first applying m_1 and then applying m_2. Many model difference representation approaches [AP03] provide this operation. When we sequentially apply two updates, the later one will overwrite the early one if there are conflicts. In this way we can use this operator to overwrite conflicts by the update from one side.

Using the sequential composition operator, we can define a three-way merge operation that deals with conflicts. This three-way merger, denoted as $merge'$, is defined below. The updates in m_a will overwrite the updates in m_b.

$$merge'(m_o, m_a, m_b) = (\mathit{diff}\,(m_o, m_a) \circ \mathit{diff}\,(m_o, m_b)).\mathit{post}$$

Using $merge'$, we construct the synchronization algorithm for conflict resolving, as shown in Figure 5.6. This algorithm is similar to the original one, except that we use $merge'$ instead of $merge$ and we do not post-check preservation. We can similarly prove that the algorithm ensure STABILITY, CONSISTENCY. However, PRESERVATION is not satisfied because the updates may be overwritten by the updates from the other side.

The second issue is about update failures. There are various reasons that an update to a model may fail. For example, there may be constraint rules that some updates is not allowed on the model, or the system represented by the models runs into some state where certain types of updates are not allowed. In such cases, it is possible that some updates reflected from the other side cannot be applied to the model, and we need to undo these updates

on the other side to ensure the two models are consistent. This can be handled by adding an extra step after the synchronization.

Before we describe the extra step, we need to clarify how the synchronizer knows the update failure. As the synchronizer interacts with the system in a state-based way, taking models and producing models, we abstract the procedure of updating a model in meta model M as a function $write_M : M \times M \to M$, which takes the current model, a model to be updated to, and produce the updated result. If $write_M(m_o, m) = m$, then the system model is successfully changed to m and there is no update failure. If $write_M(m_o, m) \neq m$, some part of the update is not successfully applied and $write_M(m)$ satisfies that $\exists u_1, u_2 \in \Delta_M : u_1.pre = u_2.pre = m_o \wedge u_1.post = m \wedge u_2.post = write_M(m_o, m) \wedge u_2 \sqsubseteq u_1$.

The goal of handling update failure is that when some part of the update cannot be applied on one side, we need to undo the corresponding part of update in the other side to ensure the consistency of the two models. As in the end of the synchronization we already have two consistent models, m_{ab} and n_{ab}, the $write_M(m, m_a)$ can be considered as an update on m_{ab}, where m_{ab} is updated into $write_M(m, m_{ab})$. In this way we can use the bidirectional transformation to undo the update.

Suppose on the M side there may be update failures, the step for handling update failures will produce a new model on the N side, which is defined as $n'_{ab} = \overrightarrow{R}(write_M(m_a, m_{ab}), n_{ab})$. Similarly, when the N side may have update failures, we produce a new model on the M side $m'_{ab} = \overleftarrow{R}(m_{ab}, write_M(n_n, n_{ab}))$. We currently do not consider the situation where both sides may have update failures.

In the handling of both conflicts and update failures, part of the update may not be preserved on the model, and we should inform users of the loss of updates. We can first compare m_o and m_a to find the original update, and then compare m_o with the result model, and compare the two updates to find which part of user update is not preserved. In an application system, the comparison of updates can be done by white-box analysis of updates.

5.4 Application: Architecture-based Runtime Management

In this section we use our algorithm to construct a general framework for architecture-based runtime management. When software systems are running, their environments and user requirements are constantly changing, which calls for an effective way to manage the systems during runtime to

find and fix defects, adapt to the changed environments, or meet new requirements [KM07, FR07]. This has attracted many researchers to investigate *architecture-based runtime management* [KM90, OMT98, GCH+04, BCRP98], which uses architecture models to represent the running system states, and supports management agents (human administrators or automated software agents [FR07]) to monitor and control the systems by directly reading and editing the architecture models. In general, researchers realize such architecture-based runtime management by developing *synchronizers* to synchronize the architecture modifications and system changes, and to "maintain the correspondence" [OMT98] between architecture models and system states. Such synchronizers are also known as "runtime architecture infrastructures" [OMT98], "reflection infrastructures" [BBF06], "translation layers" [GCH+04], etc.

An important issue of architecture-based runtime management is how to support a wider scope of systems and management activities [OMT98, GCH+04]. As a matter of fact, there are many kinds of systems that provide low-level mechanisms and APIs for manipulating their running states [SBP08], and many kinds (styles) of architecture models that are proper for different high level management activities [OMT08]. Since the types of architecture models and systems are rich and diverse and different pairs of them require different synchronizers, it is valuable to provide support for the development of synchronizers.

In this section we use our algorithm to build a framework for generating such synchronizers. As the architecture model and the system state are usually heterogeneous, we need to use model synchronization approaches. As architecture modifications and system changes might happen at the same time, we need to handle parallel updates. Consequently, our algorithm is suitable to work here. In this framework, developers write a bidirectional transformation in the QVT language [Obj08], and the framework wraps it into a synchronizer between an architecture model and a running system using our algorithm.

In addition, to manage a running system, the usual way is through management APIs rather than modifying models. To make model synchronization work on management API, we need first to wrap the management APIs into a model. As APIs vary from system to system and there is no universe standard about management APIs, a general solution for wrapping APIs is needed. Our framework also allows users to specify how to wrap management APIs into models. User specify a meta model and an "access model" describing the relation between the meta model and the management APIs, and our framework automatically generates an adapter relating models in the meta model with the running system. These adapters are also integrated into the

synchronizer to make synchronizers work on management APIs. As a result, a synchronizer consists of two parts, adapters for invoking management APIs through model modification, and the model synchronizer generated by our algorithm to synchronize models. To distinguish, we call the model synchronizer synchronization engine.

We have implemented the framework as a toolset on Eclipse platform, which can automatically generate a synchronizer from a specification given by synchronizer developers. We have successfully applied this toolset to generate many practical synchronizers, including the one to support typical C2-based runtime management [OMT98] on practical JOnAS-based JEE systems [JOn]. The toolset, the specifications and the generated synchronizers are available in our open source project[1].

Related work

Architecture-based runtime management is a hot topic in the recent decade, and lots of work has been devoted to the high-level architecture styles to support management activities, the low-level mechanisms to manipulate running systems.

For high-level management activities, Kramer et al.[KM90] first propose to represent system states as *Nodes* and *Links* to support intuitive runtime management. Oreizy et al.[OMT98] use C2 architecture style to support runtime evolution of GUI-centric systems. Garlan et al.[GCH+04] use design-time information carried by architecture models to support self-adaptation. Oreizy et al.[OMT08] surveyed many relevant approaches, and summarized several typical architecture styles.

For the low-level mechanisms, researchers have investigated on utilizing reflective middleware [BCRP98, HMY06], inserting "probes" and "executors" [GCH+04], capturing system events [CC03], and re-writing the configuration files [BGF+08]. They also try to maintain the system integrity during the reconfiguration process [ZC06]. Currently, many mainstream platforms [PLA, JOn] have built-in management mechanisms, and provide management APIs for external programs to utilize these mechanisms. Sicard et al.[SBP08] surveyed several typical management APIs.

All these previous researchers focus on management activities or mechanisms, but only develop synchronizers proper for their specific cases. Different from them, we focus on the development of synchronizers. We assume that the developers have already decided the proper architecture style and the target system with capable management API, and we provide automated

[1] `http://code.google.com/p/sandtablist/`

support for them to develop the synchronizer.

There has been much effort made in automated support for the development of architecture-based runtime management. Rainbow [GCH+04] is a framework for reusing existing work on runtime architecture, but for a new architecture style and a new type of system, developers still have to write much code. DiscoTECH [SAG+06] provides a high-level, domain specific language for specifying relation between architecture and system, but it only supports propagating changes from system to architecture. Genie [BGF+08] supports automated transformation from architecture modifications to the changes on configuration files, but currently this tool is specific to the Gridkit platform.

Our work is related with the research on self-management. Kramer and Margee [KM07] proposed a reference architecture for self-management, where management plans are executed in an abstract level. Our synchronizers could help bridge the abstraction gap between the plan execution and the raw system state. But in this paper, we do not emphasize the ability of *self*-management.

Recently, many researchers propose to utilize modeling technologies for runtime management [FR07, BBF06]. We adopt standard XMI files, so that developers can also utilize modeling technologies on our architecture models.

Antkiewicz et al. [AC06] prove that it is feasible to specify the framework APIs as models, and to generate code for using these APIs. Based on a similar idea, we designed the access model for specifying the management API, and implement the tool for generating adapters that "use" this API to manage the system state.

5.4.1 Framework Overview

In this section, we give an overview about our approach with the help of a small example, which will be used throughout this paper.

The example is shown in Figure 5.7. The right part shows a simple mobile computing system, where three mobile devices communicate with a desktop computer via PLASTIC Multi-radio Networking Platform [PLA]. The left part shows a Client/Server architecture model representing the current state of this system. We would like to develop a synchronizer between them that can enable the administrator to use this architecture model to intuitively *monitor* and *control* the system. Specifically, *monitor* means that if the system state changes, the synchronizer will refresh the architecture model to let the administrator know this change intuitively, and *control* means that if the administrator modifies the architecture model, the synchronizer will reconfigure the system state according to his modification.

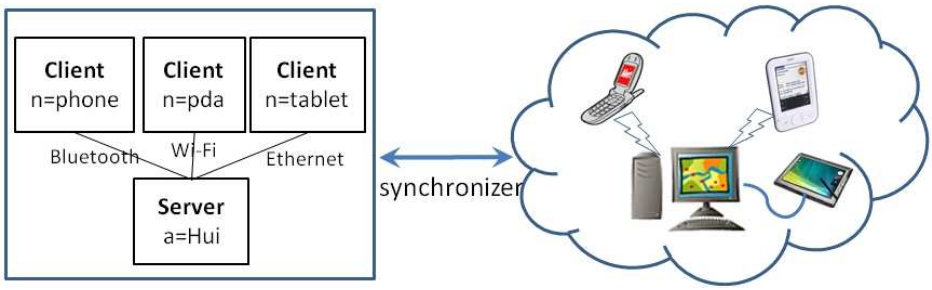

Figure 5.7: The running example

Before discussing how to develop this synchronizer, let us have a deep inspect about the runtime management case for which this synchronizer should work. 1) The architecture model is in Client/Server style. It contains a `Server` and several `Client`s and `Link`s. The `Server` records the name of current administrator ("`a`" for `admin`). The `Clients` have unique `names`, and the `Link`s have specific `types`. The architecture model is stored as XMI [Obj07] files. 2) The system state is constituted of a local desktop computer, some remote devices and devices' active networks. The synchronizer can manipulate them by invoking PLASTIC management API. For example, it can inspect the active devices by broadcasting a request and collecting the devices' responses. 3) The architecture model and the system state have a specific relation. The `Server` represents the desktop computer, the `Client`s represent the active devices, and the `Link` between a client and a server represents an available network provided by the corresponding device.

In our framework, we only require developers to provide the above information about *what* the runtime management case is, and our toolset will generate the synchronizer which works for this case. Figure 5.8 is an overview of our approach from a developer's perspective. Developers can specify the architecture style as an MOF meta-model, specify the management API as another MOF meta-model (what can be manipulated) and an access model (how to invoke the management API), and specify the relation between them as a QVT transformation. And our system automatically generates the required synchronizer. Detailed discussion about the specification work will be given in Sections 5.4.2.

Let us use a simple scenario to illustrate *how* the generated synchronizer supports this runtime management case. Suppose an administrator named "Yingfei" changed all the link types to "Wi-Fi", and changed the server's `admin` from "Hui" to "Yingfei". In the meantime, the running system itself also changed: the tablet computer stopped.

If the synchronizer is launched at this time, it will do the following tasks. 1) It identifies the above architecture modifications and system changes by

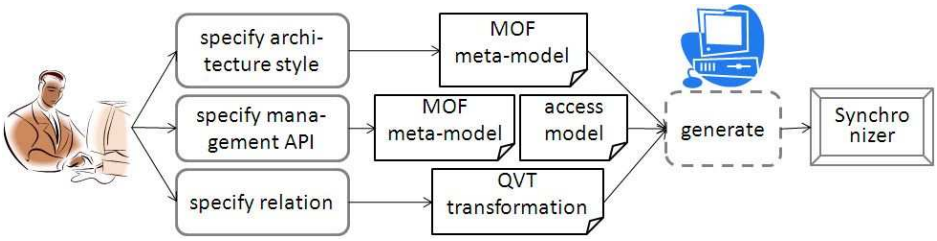

Figure 5.8: Approach overview

loading the XMI file and invoking the PLASTIC API. 2) It figures out that modifying the link type from "Bluetooth" to "Wi-Fi" *means* reconfiguring the network of the "phone" device, and thus it invokes the PLASTIC API to send a "network reset" request to this device. We assume this request succeeds and the network becomes "Wi-Fi" 3) It ignores the other modification of link type because of conflict. Specifically, the modification means a system change for reconfiguring the network of the "tablet" device. This is an *illegal system change*, because the device is stopped, and thus if the synchronizer sends a request to the tablet computer, it has to wait for the response until timeout. 4) It deletes the "tablet" client and the link in the architecture model, according to the system change. In the meantime, it preserves the modification of the server's `admin`, so that the subsequent administrator knows her predecessor. In this way, the synchronizer reconfigures the system for the administrator, and returns a new architecture model to inform him the system changes and the reconfiguration result. But the synchronizer's work is still not over, because the invocation to management API may fail. For example, if the phone currently does not support "Wi-Fi", then the request in task 2 will not have any effect. This time, the synchronizer returns an architecture model where the first link is still "Bluetooth", and it also returns a warning about this failure.

In section 5.4.3, we will present how we realize the automated generation of the synchronizer that carries out the above tasks for this specific case.

5.4.2 Specification

This section presents how to specify the architecture style, the system's management API, and the relation between them, as shown in Figure 5.8.

An architecture style defines what *kinds of element*s may exists in an architecture model (usually, but not necessarily, some kinds of components and connectors), the *properties* of each kind of elements, and the possible *containments* and *configurations* between model elements [GCH+04]. Table 5.3 shows how to specify these concepts using MOF meta-models. Following

Table 5.3: Concept mapping

Architecture style	*MOF meta-model*
Element type	Class
Property	Attribute
Composition	Containment Reference[2]
Configuration	Reference
Type of managed states	*MOF meta-model*
Managed Element Type	Class
Local State	Attribute
Composition	Containment Reference
Functional Link	Reference
Connection	Reference

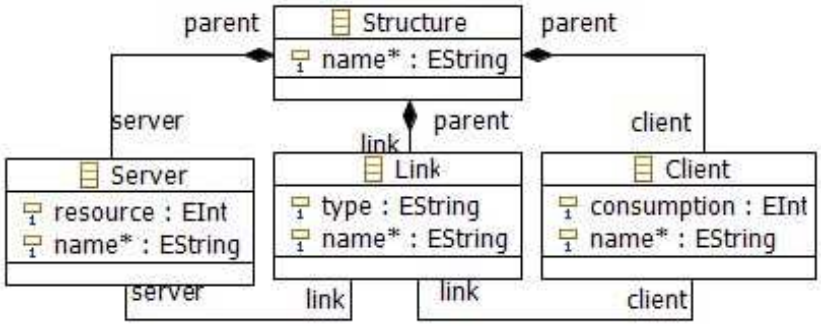

Figure 5.9: The meta-model for Client/Server style

this guidance, we can specify the architecture style in our running example as Figure 5.9. A `Structure` could contain several `Clients` and `Servers`, and they are connected by `Links`. The `Server` provides a number of `resources`, and each `Client consumes` part of these resources (which are not illustrated in Figure 5.7).

According to Sicard et al. [SBP08], the system state that can be manipulated through a management API is constituted of *managed elements* (like the `Local` computer, the `Remote` devices and their `Networks`). Managed elements have *local states* (like the maximal number of threads (`maxThreads`) required by a device or provided by the local computer). They could be *composed* by other managed elements (like a device contains networks), and they could have *functional links* or *connections* between each other. Table 5.3 shows how to define these concepts using MOF meta-models, and Figure 5.4.2 shows the sample MOF meta-model we defined for PLASTIC. As a simple example, this specification does not involve functional links and connections.

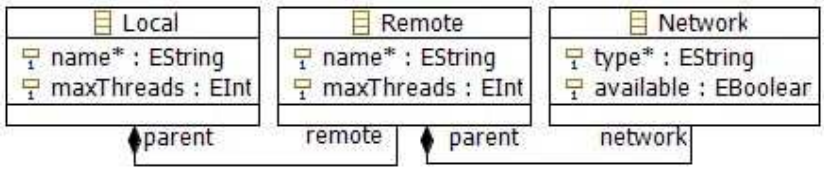

Figure 5.10: The Meta-model for PLASTIC states

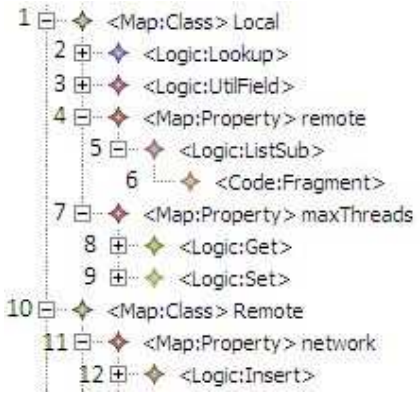

Figure 5.11: Access model for PLASTIC

We designed a domain-specific modeling language named "access model" for specifying how to invoke management APIs. The relation between the system meta-model and this access model is similar to the relation between the "syntax pattern" and the "semantic action" in Yacc. Developers could *decorate* each of the classes, attributes and references in the meta-model with some pieces of code for invoking the management API.

Figure 5.11 shows part of the access model for our running example. We defined a `ClassMap` (Element 1) and a `PropertyMap` (4) to decorate the `Local` class and its `remote` aggregation (see Figure 5.4.2). Under the `PropertyMap`, we add a `ListSub` (5) to specify how to *list* all the active remote devices. The corresponding Java code is directly specified by the `Code:Fragment` (6). The code is shown in List 1, where we launch a listening thread (Lines 2-3), and then broadcast a request (Lines 4-7). During the sleep time (Line 8), the listening thread collects the active devices' responses and stores the information of each device under a hash map. The logic for broadcasting and listening under PLASTIC API is specified inside the two Java classes `Broadcast` and `Listen`, which are defined under the `UtilField` (3). The two variables `mncListner` and `mncBroadcast` are also declared in `UtilField` (3) and the logic for instantiating and configuring them is defined in `Lookup` (2). In the same way, we also specified how to get and set the local computer's maximal thread number (8,9), and how to active a new network for a device (12).

Listing 5.1: Code for listing remote devices

```
1 HashMap res = new HashMap();
2 Listen lt = new Listen(mncListner, res);
3 new Thread(lt).start();
4 Broadcast bt = new Broadcast(mncBroadcast,
5       name+"Group",
6       mncListner.getMyPlasticAddress());
7 new Thread(bt).start();
8 Thread.sleep(1000);
9 return new ArrayList(res.values());
```

When developers define the architecture style and the state type as MOF meta-models, they implicitly regard the architecture models and system states as MOF-compliant models. Therefore, it is natural to specify the relation between them using a QVT transformation. List 2 is the sample QVT transformation for the running example.

This QVT transformation contains three `top relations`, specifying the relation as described in Section 5.4.1. For example, the third `relation`, `Link2Network` (Line 5), specifies that "the link between a client and a server represents an available network provided by the corresponding device". We can read this rule as follows. For each `link` in the architecture model (Line 7), if its parent is `strctr` (Line 8), its type is `ltype`, and it connects `clnt` and `srvr`, then there must be a `network` in the system model (Line 12). This `network` is available (Line 14) and its type equals `ltype` (Line 15). Moreover, the `network`'s grandparent `lcl` (Line 13) satisfies the `StrServer2Local` relation with `strctr` and `srvr` (Line 16), and its parent `rmt` (Line 13) satisfies the `Client2Remote` relation with `clnt` (Line 17). These `when` clauses locate the network under the correct client. This QVT transformation is bidirectional, and thus the above rule also means that for each `network` there must be a `link`.

5.4.3 Generation

This section presents how we achieve the automated generation of synchronizers. Figure 5.12 is a layered illustration of our generation approach. From the developer's specifications (Layer 3), our generation toolset (Layer 2) could automatically generate the synchronizer (Layer 1) between the architecture model and the system state(Layer 0).

We divide the synchronizer into three components (which is a common structure for synchronizers [OMT98, GCH+04, SAG+06]). The *architecture*

Listing 5.2: Sample QVT transformation

```
1  transformation CS2PLA(arc:CS,sys:Plastic){
2    key Structure{name};...key Network{type};
3    top relation StrServer2Local{ ... }
4    top relation Client2Remote{ ...  }
5    top relation Link2Network{
6     ltype:String;
7     enforce domain arc link:Link{
8      parent=strctr:Structure{},
9      client=clnt:Client{},
10     server=srvr:Server{},
11     type=ltype };
12    enforce domain sys network:Network{
13     parent=rmt:Remote{parent=lcl:Local{}},
14     available=true,
15     type=ltype};
16    when{StrServer2Local(strctr,srvr,local) and
17         Client2Remote(clnt,rmt); }  }  }
```

adapter manipulates architecture models by reading and writing XMI files, and the *system adapter* manipulates system states by invoking the management API. The *synchronization engine* interacts the adapters through a standard MOF reflection interface, so that it can manipulates the various architecture models and system states in a unique way. This enables the engine to use a general solution to address the issues like propagating changes, handling conflicts, etc. We developed two generation tools, which generate Java code that implements MOF reflection interface, loads and saves XMI files, and invokes the management API. Such code constitutes the two adapters. We also designed a synchronization algorithm to address the common issues, and implement it as a general synchronization engine. When we generate a specific synchronizer, we just configure this general engine with the specific QVT transformation and meta-models.

Generating the two adapters

We utilize the EMF (Eclipse Modeling Framework) code generation engine [BBM03] to generate the architecture adapters. EMF generates a Java class for each of the MOF classes defined in the architecture meta-model. These Java classes contain the *standard methods* that conform to the MOF reflection interface. External programs could use these *standard methods* to manipulate

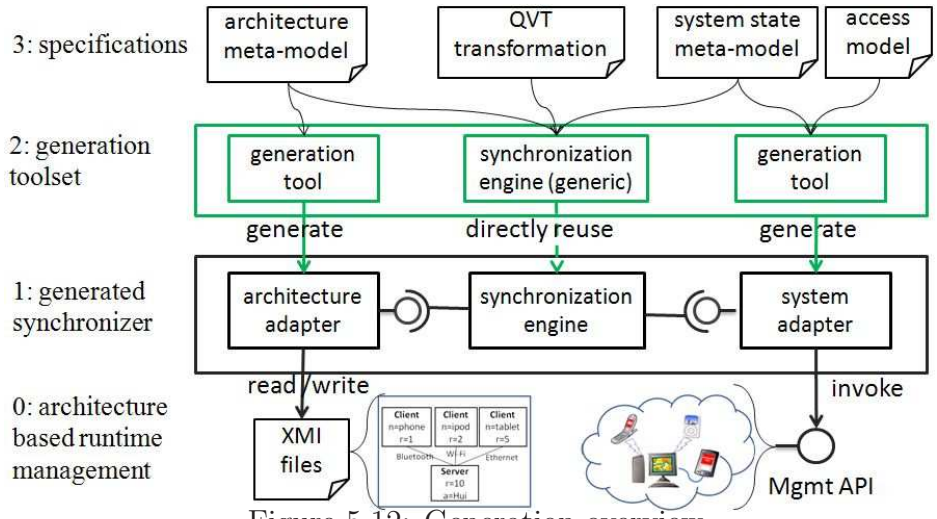

Figure 5.12: Generation overview

a set of member variables, whose values are bound with a XMI file by an XMI parser.

We extend the EMF generation engine to generate the system adapters. The generated adapters are also constituted of Java classes that implement the MOF reflection interface. But besides the *standard methods*, we also generate some *specific methods* from the access model for invoking the management API. When an external program invokes a standard method, the generated code inside the standard methods will forward the invocation to the proper specific methods.

We use an example to illustrate the above presentation. List 3 is a sample Java Class in the system adapter for PLASTIC. It is generated from the `Local` class (Figure 5.4.2) and its `ClassMap`(Element 2 in Figure 5.11). Lines 2-4 are generated from the `UtilField` (4). Lines 5-6 are generated according to the "remote" aggregation, where `EListSubAdapter` is a utility class dealing with multiple properties. Line 7 is generated from `Lookup` and contains the instantiation logic. `listSubCores` (Line 8) is a *specific method* generated from `ListSub` (5), which lists all the active devices. `eGet` (Line 18) is a *standard method* for getting property values.

The generated adapters hide the variance of architecture models and system states. External programs could manipulate the different architecture models and system states by invoking the same set of *standard methods*. We use one example to illustrate how the adapters work. The tool in this example will be used later in the synchronization algorithm.

The "model clone" tool in EMF [BBM03] provides a `copy(mod1, mod2)` method to clone the content from `mod2` to `mod1`. If we invoke `copy(mod1,`

Listing 5.3: Sample of generated system adapter

```
1  public class LocalImpl extends EObjectImpl{
2    class Listen implements Runnable {...}
3    private static MNClient mncListener =
4      new MNClient(name + "Listener", maxThreads);
5    private EListForAdapt<Remote> remote=
6      new EListForAdapt<Remote>(...);
7    public Object lookupCore() {...}
8    public List listSubCores(int featureID) {
9      switch(featureID){
10     case PlasticSystemPackage.LOCALREMOTE:
11       HashMap remotes = new HashMap();
12       Listen lt = new Listen(mncListner, remotes);
13         new Thread(lt).start();
14       /*The same lines in List 1 */
15       return new ArrayList(remotes.values());
16     }
17   }
18   public Object eGet(int featureID...) {
19     switch (featureID){
20     case PlasticSystemPackage.LOCALREMOTE:
21       remote.refresh();
22       return remote;
23     }   }   }
```

`sysAdapter)`, the clone tool will load the current system state into a model. Let us see how this tool interacts with the system adapter. Suppose now this clone tool wants to copy the children of the root `Local` element. It first checks the meta-model (Figure 5.4.2) and find out that `Local` has an aggregation named "remote", and so it invokes `eGet('remote')` on this root element. The invoked `eGet` (Line 18) refreshes the `remote` list (Line 21). This `refresh()` method of `EListForAdapter` invokes back the `listSubCores` (Line 8) to collect the information of all the active remote devices (Lines 12-16, as we have introduced in Section 5.4.2), and instantiate a `RemoteImpl` for each of the devices. In this way, the `copy` method obtained a list with instances of `RemoteImpl`.

Synchronization Algorithm

With the above preparations, we can construct our synchronization algorithm. The algorithm is constructed using the extensions in Section 5.3 to handle conflicts and update failures. Because the updates to the architecture model representing the management behavior of users, we use the update to the architecture to overwrite that to the system when they conflict. Because updates to the system sometimes may fail, we handle update failure on the system side.

To construct the algorithm, we need to provide some core components used in the algorithm. First, the bidirectional transformation is obtained from the QVT bidirectional transformation program. Although QVT enables rapid development of bidirectional transformations, it does not always guarantee CORRECTNESS and HIPPOCRATICNESS. If a program has complex interaction with the constraints on the meta models, it may produce inconsistent result. As a result, it is up to programmers to carefully check their programs to ensure the two properties. Nevertheless, as synchronization in architecture-based runtime management mainly involves attribute-mapping, the two properties are often satisfied.

Second, the model difference operation is obtained from our dictionary representations (c.f. Chapter 3). We first convert models into dictionaries, and then use *find_update* to compare dictionaries. To assign the UIDs across different version of models, we assume each model elements has a special attribute that uniquely identifies this model element and will not be changed when user modify the models. If model elements in two versions of models have the same value in their identifying attributes, we assign the same UID to them.

Third, the `write` operation is constructed use the `copy` tool that we constructed in the end of the last section. We first copy the target model to

the system adapter, and then read the adapter again. The model obtained by reading the adapter is the result of the `write` operation, where the applicable updates are all applied, and the inapplicable updates, if any, are discarded away.

Finally, because the updates on the architecture model may not be successfully applied, we need to report the failed updates to users. This is achieved by first comparing models to find the user updates and the applied updates, then converting updates into a set of pairs containing locations and primitive updates for comparison, and reporting the pairs missing in the applied updates.

5.4.4 Properties of the Synchronizer

Applying our algorithm not only helps construct the synchronizer, but also ensures many important properties on the synchronizer, which ensure the synchronizer works correctly in the runtime management.

Property 1. The result architecture model and system state are *consistent* according to the specified relation, even though there are conflicts or failures. This directly follows the CONSISTENCY property of our algorithm. In addition, this is also known as a basic property of runtime management [OMT98]. It ensures that the management agents always get the right representation of system state.

Property 2. Unchanged architecture model does not cause system change. This is a reduced version of STABILITY and can be proved similarly. This property prevents the synchronizers from *polluting* the running system without the management agents' intention.

Property 3. Conflicts do not harm the system. This is because we overwrite the system updates with architecture updates when there are conflicts. This property liberates management agents from worrying about conflicts when they modify the architecture.

Property 4. "Irrelevant" architecture modifications remain in the result architecture model. "Irrelevant" modifications are the ones that do not have meanings on the system state, like the change of server's administrator name (see Section 5.4.1). This property allows management agents to record irrelevant information on architecture models to assist further management activities.

Property 5. The synchronization result is unambiguous. From the synchronizers' perspective, they will *try* to propagate *every* change without conflict, according to the behavior of the three-way merger. From the management agents' perspective, if their modifications remain in the final architecture model, then these modifications must have been propagated successfully,

otherwise the agents could receive warnings.

Note that we require some premises on the running system. For Property 2, we require that the API method for *retrieving* states does not have side-effects. For Property 3, we require that the system does not change during the process of three way merge. For some kinds of systems that do not satisfy these premises, developers have to do some extra work when defining the access model, e.g. utilizing state-locks.

5.5 Summary

In this chapter we have proposed an approach that wraps a bidirectional transformation program and a model difference approach into a synchronizer for parallel updates. Our approach is general and predictable. It is general in the sense that it allows the use of any bidirectional transformation and any model difference approach, and it is predictable because it satisfies three model synchronization properties: CONSISTENCY, STABILITY and PRESERVATION. We also use algorithm to construct an architecture-based runtime management framework, which not only certifies the usefulness of our algorithm, but also greatly facilitate the development of architecture-based runtime management systems.

Chapter 6

Beanbag: An On-Site Synchronization Language

6.1 Motivation Example

In the previous two Chapters we have seen our support for off-site synchronization. In this chapter we introduce our support for on-site synchronization. Chapter 1 has discussed the difference between off-site synchronization and on-site synchronization and Figures 1.1-1.3 has shown an example of on-site synchronization.

Although there is currently no high-level language for writing on-site synchronizers, there are languages for defining consistency relations over models [Obj06, Jac02]. One of such languages is the Object Constraint Language (OCL) [Obj06] that is to define and check consistency relations over models. As an example, we give two OCL relations describing the consistency relation over the UML model in Figure 1.1.

```
C1: context Message
    inv let rec = self.receiver in
        let ops = rec.base.operations in
        ops->exists(oper | oper.name = self.name)
C2: context Message
    inv self.sender <> null and self.receiver <> null
```

C1 requires every message in a sequence diagram to be declared as an operation in the receiver's class. The `context` keyword states that it will be applied to every `Message` object. The concrete relation definition starts from a message (`self`), finds the receiver object (`rec`), finds the operation set in the class of the receiver object (`ops`) and check if there exists an operation having the same name of the message. C2 requires every message to have

a sender object and a receiver object by disallowing related features to be `null`.

As there are many existing OCL programs, we would like to ask whether we can automatically derive synchronizers from OCL programs. As a matter of fact, if we examine the definitions of consistency relations, we would find that some relations have already implicitly included the definitions of synchronizers. Consider a simple relation that two primitive values `a` and `b` are equal: `a=b`. If a user changes `a`, the only reasonable action to take is to change `b` accordingly. The same strategy also works when `b` is changed. However, it is not easy to automatically derive synchronizers beyond the simplest relations. First, many consistency relations, like C1, have multiple, even infinite numbers of actions to take for some updates. How to choose one among them is unknown. Second, consistency relations may be composed by operators like `and`, `or` and quantifiers. It is unclear how to compose synchronizers accordingly while ensuring a correct synchronization behavior.

In this chapter we suggest a compromising approach. Instead of deriving synchronizers purely from consistency relations, we ask users to provide some extra information so that we can determine a unique synchronization behavior from a consistency relation. In this way we can greatly reduce the effort in implementing a synchronizer (because the amount of extra information is usually small) and we can ensure the correctness of the synchronizer from the derivation process.

To achieve this, we design a language, Beanbag, for users to define a consistency relation and a synchronization behavior at the same time. The Beanbag language defines consistency relations in OCL-like syntax, but every relation in Beanbag also has a synchronization semantics describing when some parts of the data are changed by users, how to change the other parts to ensure consistency. For relations with multiple synchronization behavior, Beanbag provides users with more than one way to construct one relation, where a different way indicates a different synchronization behavior.

We evaluate the expressiveness and usability of Beanbag by developing Beanbag programs for consistency relations in MetaObject Facility (MOF) [OMG02] and UML [Egy07, ELF08] models. The evaluation shows that Beanbag greatly eases the development and can support many useful scenarios in practice.

To have a concrete idea of Beanbag, let us write a Beanbag program for relation C1. A Beanbag relation describing C1 can be defined as follows.

```
def C1(msg, model) :=
 let rec = model.(msg."receiver") in
 let opRefs = model.(rec."base")."operations" in
```

```
 opRefs->exists(opRef | model.opRef."name"=msg."name")
```

We can see that the Beanbag program is very similar to the OCL expression. One small difference is that Beanbag is built on dictionaries. Therefore, instead of writing `msg."receiver"`, we need to write `model.(msg."receiver")`. This difference is not important for now.

To apply this relation to all instances of `Message`, we further write the following code:

```
def C1onAll(model, meta) :=
 model->forall(obj |
  isTypeOf(obj, "Message", meta) and C1(obj, model)
  or not isTypeOf(obj, "Message", meta)
)
```

The `C1onAll` relation works in two modes. In the checking mode, it runs as a normal OCL expression, takes the current model and the meta model as input and produces a boolean value indicating whether the relation is satisfied. In the synchronization mode, it takes as input the current model, the meta model and the updates that users try to apply to the model, and produces new updates representing actions to take to make the model consistent. The input updates can be a single update, changing a single feature or inserting/deleting an object. It can also be a combination of several updates performed by different users in a distributed environment, changing several locations or inserting/deleting several objects.

Putting it more concretely, the synchronization mode of `C1onAll` proceeds in a similar way to the checking mode, but propagates updates when it encounters one. Suppose a user has renamed an operation to a new name. `C1onAll` will invoke `C1` on all `Message` objects and `C1` will check if there exists an operation with the same name. For the renamed message, such an operation cannot be found. Then the `exists` statement will insert a new `null` reference in the collection and proceed to the inner relation. The expression `model.opRef` will create a new operation and replace the `null` reference with the actual reference. Finally, the equality relation will assign the changed name to the newly created operation. We will see the precise semantics in Section 6.2.3.

As discussed in Chapter 1, when an operation is renamed, we can either insert a new operation or rename an existing operation. To rename an existing operation, what we need to do is to change `exists` to `exists!` in the last line. The new Beanbag program runs the same in the checking mode, but in the synchronization mode it will rename the operation that originally corresponded to the message.

Note that the current program for C1 can be improved; the current version

c	::=	`v=v`
	|	c `and` c
	|	c `or` c
	|	`d->forall(v|`c`)`
	|	`d->exists(v|`c`)`
	|	`d->exists!(v|`c`)`
	|	`let v=`e `in` c
	|	`protect v in` c
	|	`test` c
	|	`not` c
e	::=	*const*
	|	`d.k`
	|	`let v=`e `in` e
`v`	::=	any variable
`d`	::=	a dictionary variable
`k`	::=	a key variable

Figure 6.1: Core synatx

will insert a new operation even when we change the receiver of a message or when we delete/rename an operation in the class diagram. We can describe a more natural synchronization behavior by extending C1 to allow the message name to be `null`. We will see a full featured program in Section 6.2.4.

6.2 The Beanbag Language

6.2.1 An Overview

Figure 6.1 shows the syntax of the core Beanbag language. The Beanbag language has similar syntax as OCL [Obj06]. It has the primitive constraint "`=`" to describe equal relation between two variables, uses logic operators of `and`, `or` and `not`, and quantifiers of `forall` and `exists` on keys of dictionaries to construct complex constraints, and binds variables with expressions with the `let` construct. An expression may be a constant value, a dictionary key indexing $d.k$, or a local binding expression with `let`. With these constructs, Beanbag is powerful to describe various kinds of constraints; we have seen several examples in the introduction, and will see more examples in Section 6.2.4.

As we have seen in Section 6.1, a Beanbag program can be executed in either the checking mode or the synchronization mode. We use two functions to describe the semantics in the two modes. In the two functions, we

use two sets of bindings to pass the information. The variable-value bindings, often denoted as σ : VAR $\rightarrow$ *value*, bind variables to data values. The variable-update bindings, often denoted as τ : VAR $\rightarrow$ *update*, bind variables to updates. We write $\texttt{var}^\sigma$ or $\texttt{var}^\tau$ for the value or the update bound to variable var in binding set σ or τ. We write $dom(\sigma)$ for the set of all variables in σ. We also write $\tau(\sigma)$ to denote applying all updates in τ to the corresponding variables in σ and returning a new set of variable-value bindings.

Suppose c is a constraint (an instantiated relation) defined by Beanbag. The checking function evaluates c according to a set of variable bindings.

$$E[\![c]\!] : (\texttt{VAR} \rightarrow \mathit{value}) \rightarrow \texttt{BOOLEAN}$$

For example, $E[\![\texttt{a=b}]\!]$ returns true for an input σ where $\texttt{a}^\sigma = \texttt{b}^\sigma$.

The synchronization function is a partial function takes the current values and the updates on the variables and produces new updates on the variables to satisfy the constraint.

$$R[\![c]\!] : (\texttt{VAR} \rightarrow \mathit{value}) \times (\texttt{VAR} \rightarrow \mathit{update}) \rightarrow (\texttt{VAR} \rightarrow \mathit{update})$$

The function is partial (returning $\perp$ on some input) because the updates may conflict with each other or may not be allowed by the program. In such cases the modeling tool should report an error message to users. For example, given an input (σ, τ) where $\texttt{a}^\sigma = \texttt{b}^\sigma = 1$, $\texttt{a}^\tau = \texttt{void}$, and $\texttt{b}^\tau = \texttt{!2}$, $R[\![\texttt{a=b}]\!]$ returns τ' where $\texttt{a}^{\tau'} = \texttt{!2}$ and $\texttt{b}^{\tau'} = \texttt{!2}$. If $\texttt{a}^\tau = \texttt{!3}$ and $\texttt{b}^\tau = \texttt{!2}$, $R[\![\texttt{a=b}]\!]$ returns $\perp$.

Different from OCL, the Beanbag provides the following declarative ways for people to define synchronization behavior for reestablishing the consistency relation after an update happens.

- *Each standard constraint operator is equipped with a specific synchronization operation.* For example, the primitive equation constraint "$v_1 = v_2$" will propagate updates from one to the other while treating v_1 with higher priority (so $v_2 = v_1$ has different synchronization behavior from $v_1 = v_2$). The conjunction "c_1 `and` c_2" synchronize updates using the synchronization functions of both c_1 and c_2. The disjunction "c_1 `or` c_2" will synchronize updates by first trying the synchronization function of c_1 if c_1 was satisfied before updates happen, and that of c_2 otherwise, and if this fails we try the synchronization function of the other. The forall qualifier "`d->forall(v|`c`)`" will synchronize updates by synchronizing each dictionary entry with the synchronization function of c if necessary.

- *New constraint constructors are introduced to describe different synchronization functions.* Two forms of the existence constraint are provided to dealing with flexible synchronization of dictionary structures. For instance, the two constraints

```
model->exists(class|class."name"=x)
model->exists!(class|class."name"=x)
```

  describe the same consistent relation that there exist a `class` in the `model` whose name is equal to `x`. But they behave differently when the consistency is destroyed by, for example, a change of `x`. The former will create a new class with its name equal to the changed `x`, while the later will rename the existing class whose name is equal to `x` before `x` is changed.

- *New constructs are introduced to restrict synchronization behavior.* The construct "`protect v in` c" describes the same constraint as c but does not allow its synchronization function to update `v`, while the test construct "`test` c" describes the same constraint as c and allows no update on any variable.

In the following, after briefly explaining the common checking semantics, we focus on a detailed and formal definition of our new synchronization semantics of the language.

6.2.2 Checking Semantics

Figure 6.2 shows the checking semantics of the Beanbag language. $E[\![c]\!](\sigma)$ and $E[\![e]\!](\sigma)$ evaluate a constraint c to a boolean value and an expression to a value under a set of variable-value bindings, respectively.

We write $\sigma[\texttt{var} \mapsto v]$ to indicate a new set of bindings that maps a variable `var` to value v and maps all other variables to the same values as σ. We will also use this notation for dictionaries and updates on dictionaries.

We can see that the checking semantics of Beanbag is the same as what we can expect from the syntax. In the following part we will focus on the fixing semantics.

6.2.3 Synchronization Semantics

One of our major contributions is a natural and correct synchronization semantics for Beanbag, an extended constraint language. Our idea is to propagate updates through equality constraints, control the propagation order

$$
\begin{array}{lcl}
E[\![\texttt{v}_1\texttt{=v}_2]\!](\sigma) & = & \begin{cases}\texttt{true} & {\texttt{v}_1}^\sigma = {\texttt{v}_2}^\sigma \\ \texttt{false} & {\texttt{v}_1}^\sigma \neq {\texttt{v}_2}^\sigma\end{cases} \\
E[\![c_1 \texttt{ and } c_2]\!](\sigma) & = & E[\![c_1]\!](\sigma) \wedge E[\![c_2]\!](\sigma) \\
E[\![c_1 \texttt{ or } c_2]\!](\sigma) & = & E[\![c_1]\!](\sigma) \vee E[\![c_2]\!](\sigma) \\
E[\![\texttt{d->forall(v|}c\texttt{)}]\!](\sigma) & = & \forall k \in dom(\texttt{d}^\sigma) : E[\![c]\!](\sigma[\texttt{v} \mapsto \texttt{d}^\sigma.k]) \\
E[\![\texttt{d->exists(v|}c\texttt{)}]\!](\sigma) & = & \exists k \in dom(\texttt{d}^\sigma) : E[\![c]\!](\sigma[\texttt{v} \mapsto \texttt{d}^\sigma.k]) \\
E[\![\texttt{d->exists!(v|}c\texttt{)}]\!](\sigma) & = & E[\![\texttt{d->exists(v|}c\texttt{)}]\!](\sigma) \\
E[\![\texttt{let v=}e\texttt{ in }c]\!](\sigma) & = & E[\![c]\!](\sigma[\texttt{v} \mapsto E[\![e]\!](\sigma)]) \\
E[\![\texttt{protect v in }c]\!](\sigma) & = & E[\![c]\!](\sigma) \\
E[\![\texttt{test }c]\!](\sigma) & = & E[\![c]\!](\sigma) \\
E[\![\texttt{not }c]\!](\sigma) & = & \neg E[\![c]\!](\sigma) \\
E[\![const]\!](\sigma) & = & const \\
E[\![\texttt{d.k}]\!](\sigma) & = & \texttt{d}^\sigma.\texttt{k}^\sigma \\
E[\![\texttt{let v=}e_1\texttt{ in }e_2]\!](\sigma) & = & E[\![e_2]\!](\sigma[\texttt{v} \mapsto E[\![e_1]\!](\sigma)])
\end{array}
$$

Figure 6.2: Checking semantics

by logic operators, derive structural updating through logic quantifiers, restrict synchronization behavior through special constructs, and introduce recursion for describing more involved synchronization strategies. We will use $R[\![\texttt{c}]\!](\sigma, \tau)$ to describe the synchronization for the constraint c under the variable-value binding set σ and an update described by the variable-update binding set τ. Its result is a new variable-update binding set showing how to update variables in such a way that c is satisfied again. We will define $R[\![\texttt{c}]\!](\sigma, \tau)$ by induction on the construction of c. For simplicity, we assume all bound variables introduced by `forall`, `exists` and `let` have different names from the free variables.

Update Propagation based on Equality Constraints Propagating updates from one part to another to synchronize updates can be reduced to dealing with the following three equality constraints in our framework. This reduction will be explained in the synchronization semantics for the `let` construct.

- $R[\![\texttt{v}_1\texttt{=v}_2]\!](\sigma, \tau)$. To establish the equality constraint between two variables, we propagate updates on $\texttt{v}_1$ and $\texttt{v}_2$ to each other, and compute a new variable-update binding set τ' such that ${\texttt{v}_1}^{\tau'(\sigma)} = {\texttt{v}_2}^{\tau'(\sigma)}$. Let us first consider a simple case, where the values of $\texttt{v}_1$ and $\texttt{v}_2$ are equal before updating τ, that is, ${\texttt{v}_1}^\sigma = {\texttt{v}_2}^\sigma$. In this case, we simply merge the input updates on $\texttt{v}_1$ and $\texttt{v}_2$ when they are compatible, and return

$\perp$ when they conflict.

$$R[\![\mathtt{v_1}{=}\mathtt{v_2}]\!](\sigma,\tau) = \begin{cases} \tau[\mathtt{v_1} \mapsto u][\mathtt{v_2} \mapsto u] & {\mathtt{v_1}}^{\tau} \oplus {\mathtt{v_2}}^{\tau} \\ \perp & \text{otherwise} \end{cases}$$

where u is an update merging the updates on $\mathtt{v_1}$ and $\mathtt{v_2}$, i.e., $u = {\mathtt{v_1}}^{\tau} \circ {\mathtt{v_2}}^{\tau}$. Generally, the values of $\mathtt{v_1}$ and $\mathtt{v_2}$ may be unequal before updating, i.e., ${\mathtt{v_1}}^{\sigma} \neq {\mathtt{v_2}}^{\sigma}$. In this case, we first apply both updates to $\mathtt{v_2}$ to get a new value *new_v*, and then we calculate the update u by merging the two updates on $\mathtt{v_1}$ and $\mathtt{v_2}$ with the update *find_update*$({\mathtt{v_1}}^{\sigma}, \mathit{new\_v})$ that are used to change the value of $\mathtt{v_2}$ to *new_v*.

$$\begin{array}{l} R[\![\mathtt{v_1}{=}\mathtt{v_2}]\!](\sigma,\tau) = \begin{cases} \tau[\mathtt{v_1} \mapsto u][\mathtt{v_2} \mapsto u] & {\mathtt{v_1}}^{\tau} \oplus {\mathtt{v_2}}^{\tau} \\ \perp & \text{otherwise} \end{cases} \\ \text{where } u = ({\mathtt{v_1}}^{\tau} \circ {\mathtt{v_2}}^{\tau}) \circ \mathit{find\_update}({\mathtt{v_1}}^{\sigma}, \mathit{new\_v}) \\ \quad \mathit{new\_v} = U[\![{\mathtt{v_1}}^{\tau} \circ {\mathtt{v_2}}^{\tau}]\!]({\mathtt{v_2}}^{\sigma}) \end{array}$$

It is worth noting that after the synchronization, both $\mathtt{v_1}$ and $\mathtt{v_2}$ are equal to *new_v*.

- $R[\![\mathtt{v}{=}\mathit{const}]\!](\sigma,\tau)$. To establish the equality constraint between a variable and a constant, we calculate an update over $\mathtt{v}^{\tau}$ by finding an update to change the updated $\mathtt{v}$ (i.e., $\mathtt{v}^{\tau(\sigma)}$) to the constant.

$$\begin{array}{l} R[\![\mathtt{v}{=}\mathit{const}]\!](\sigma,\tau) = \begin{cases} \tau[\mathtt{v} \mapsto u \circ \mathtt{v}^{\tau}] & u \oplus \mathtt{v}^{\tau} \\ \perp & \text{otherwise} \end{cases} \\ \text{where } u = \mathit{find\_update}(\mathtt{v}^{\tau(\sigma)}, \mathit{const}) \end{array}$$

- $R[\![\mathtt{v}{=}\mathtt{d.k}]\!](\sigma,\tau)$. To establish the equality constraint between a variable $\mathtt{v}$ and a dictionary key indexing $\mathtt{d.k}$, we first check the key k. If k is original *null* with no update, denoted by *isNull*(k), we create a new key using the function *newID*(σ,τ) and do synchronization. If k is deleted, i.e., $\mathtt{k}^{\tau} = \mathtt{!null}$, we have no way to do synchronization and return $\perp$. Otherwise, we do synchronization on $R[\![\mathtt{v}{=}\mathtt{v'}]\!](\sigma,\tau)$ where $\mathtt{v'}$ is a fresh variable referring to the same value and update that $\mathtt{k}^{\tau(\sigma)}$ is mapped to by $\mathtt{d}$. When $\mathtt{v'}^{\sigma}$ or $\mathtt{v'}^{\tau}$ changes, the $\mathtt{d}^{\sigma}.\mathtt{k}^{\tau(\sigma)}$ or $\mathtt{d}^{\tau}.\mathtt{k}^{\tau(\sigma)}$ changes accordingly. This is denoted by $R[\![\mathtt{v}{=}\mathtt{v'}]\!](\sigma[\mathtt{v'}{=}\mathtt{d.k}^{\tau(\sigma)}], \tau[\mathtt{v'}{=}\mathtt{d.k}^{\tau(\sigma)}])$.

$$\begin{array}{l} R[\![\mathtt{v}{=}\mathtt{d.k}]\!](\sigma,\tau) = \\ \begin{cases} R[\![\mathtt{v}{=}\mathtt{d.k}]\!](\sigma, \tau[\mathtt{k} \mapsto \mathtt{!}\mathit{newID}(\sigma,\tau)]) & \mathit{isNull}(k) \\ \perp & \mathtt{k}^{\tau} = \mathtt{!null} \\ R[\![\mathtt{v}{=}\mathtt{v'}]\!](\sigma[\mathtt{v'}{=}\mathtt{d.k}^{\tau(\sigma)}], \tau[\mathtt{v'}{=}\mathtt{d.k}^{\tau(\sigma)}]) & \text{otherwise} \end{cases} \end{array}$$

Propagation Order Control based on Logic Operators We assign a synchronization semantics to the logic operators of `and` and `or` to control the order of update propagation.

- The synchronization function $R[\![c_1 \texttt{ and } c_2]\!]$ is to establish both c_1 and c_2. To do this, we call $R[\![c_1]\!]$ and $R[\![c_2]\!]$ one by one in this order to propagate updates. Since $R[\![c_2]\!]$ may propagate new updates to variables used in c_1, we may need to call $R[\![c_1]\!]$ again to satisfy c_1. Similarly, a call of $R[\![c_1]\!]$ may require a call of $R[\![c_2]\!]$. Hence in the synchronization function we repeatedly call $R[\![c_1]\!]$ and $R[\![c_2]\!]$ until we reach a fixed point where no new update is propagated.

  $$R[\![c_1 \texttt{ and } c_2]\!](\sigma,\tau) = \begin{cases} \tau' & \tau' = \tau \\ R[\![c_1 \texttt{ and } c_2]\!](\sigma,\tau') & \tau' \neq \tau \end{cases}$$
  $$\text{where } \tau' = R[\![c_2]\!](\sigma, R[\![c_1]\!](\sigma,\tau))$$

  Beanbag does not guarantee the existence of such a fixed point, and thus $R[\![c_1 \texttt{ and } c_2]\!]$ does not always terminate. However, this will not be a problem in practice because most programs will terminate. We will discuss more on this issue in Section 6.3.

  It is worth noting a different order of c_1 and c_2 sometimes leads to different synchronization behavior. We can write "c_1 `and` c_2" or "c_2 `and` c_1" to customize the behavior in such cases.

- The synchronization function $R[\![c_1 \texttt{ or } c_2]\!]$ is to make either c_1 or c_2 be satisfied. As a result, in the synchronization mode we can choose to use either $R[\![c_1]\!]$ or $R[\![c_2]\!]$ to propagate updates. However, to satisfy STABILITY, we must first use the one that is previously established on the data, otherwise the other constraint may produce updates to change the data and violate STABILITY. For example, let us consider $R[\![$`a=b."x" or a=b."y"`$]\!]$. Suppose in the input data bindings `a` is equal to `b."y"` and the update bindings map both `a` and `b` to `void`. If we choose the first constraint, `a` will be changed to `b."x"` and STABILITY is violated.

  In $R[\![c_1 \texttt{ or } c_2]\!]$, we first find out the constraint that is previously established by calling $E[\![c_1]\!]$ and $E[\![c_2]\!]$, and use the constraint to propagate updates. If the constraint fails to propagate updates, we use the other constraint. Because here we switch from one constraint to the other, the data that is previously consistent for the former may not be consistent for the latter. That is why we require synchronization functions to handle inconsistent data bindings. When both $E[\![c_1]\!]$ and $E[\![c_2]\!]$ evaluates to `false`, that is, when the input data bindings are not consistent,

we first try $R[\![c_1]\!]$ and then try $R[\![c_2]\!]$. This strategy is very useful in customizing the synchronization behavior: programmers can assign a higher priority to a constraint by writing it first.

$$R[\![c_1 \texttt{ or } c_2]\!](\sigma,\tau) = \begin{cases} R[\![c_1]\!](\sigma,\tau) & \text{if } E[\![c_1]\!](\sigma) \wedge R[\![c_1]\!](\sigma,\tau) \neq \bot \\ R[\![c_2]\!](\sigma,\tau) & \text{elif } E[\![c_2]\!](\sigma) \wedge R[\![c_2]\!](\sigma,\tau) \neq \bot \\ R[\![c_1]\!](\sigma,\tau) & \text{elif } R[\![c_1]\!](\sigma,\tau) \neq \bot \\ R[\![c_2]\!](\sigma,\tau) & \text{otherwise} \end{cases}$$

Derivation of Structural Updating based on Logic Quantifiers The `forall` and `exists` statements both involve invoking an inner constraint on values in a dictionary. We will assign a synchronization semantics to them to deal with updating on dictionary structures.

- The `forall` constraint is satisfied only if the inner constraint is satisfied by all entries in the domain of the dictionary. Consequently we can call the synchronization function of the inner constraint on all entries in the domain. Since sometimes we need to propagate from the deletion of an entry, we also call on the deleted entries. Because updates may be propagated to variables other than `v`, we recursively call the synchronization function until we reach a fixed point, the same as the `and` operator. In the definition we use an union operator on update bindings to construct the result. The union $\tau_1 \cup \tau_2$ is a set of bindings where the updates on the same variable in τ_1 and τ_2 are merged, and is $\bot$ when some updates conflict, or any of its operands is $\bot$.

$$R[\![\texttt{d->forall(v|}c\texttt{)}]\!](\sigma,\tau) = \begin{cases} \tau' & \tau' = \tau \\ R[\![\texttt{d->forall(v|}c\texttt{)}]\!](\sigma,\tau') & \tau' \neq \tau \end{cases}$$
$$\text{where}$$
$$\tau' = \bigcup_{\forall k \in dom(\texttt{d}^{\sigma}) \cup dom(\texttt{d}^{\tau(\sigma)})} . R[\![c]\!](\sigma[\texttt{v=d.}k], \tau[\texttt{v=d.}k])$$

- The `exists` constraint is satisfied if one entry in the dictionary satisfies the inner constraint. Therefore, we can reestablish the relation by inserting a new entry in the dictionary that satisfies the inner constraint. When we insert a new entry, we generate a new ID as the key of the entry with the function $newID(\sigma,\tau)$. The value part of the new entry is generated by invoking the synchronization function of the inner constraint on the new key. Because the key does not in the domain of the dictionaries, `v` will be mapped to `null` and `void` in the two binding

sets. The inner constraint must change `v` to some value different from `null` otherwise the synchronization function will return $\perp$.

$$R[\![\texttt{d->exists(v|}c\texttt{)}]\!](\sigma,\tau) = \begin{cases}\tau & \text{if } E[\![\texttt{d->exists(v|}c\texttt{)}]\!](\tau(\sigma)) \\ R[\![c \texttt{ and not (v=null)}]\!](\sigma[\texttt{v=d.}k],\tau[\texttt{v=d.}k]) & \\ & \text{otherwise}\end{cases}$$
where $k = newID(\sigma,\tau)$

- The `exists!` construct locates an entry that previously satisfies the inner constraint and updates the entry. However, such an entry may not be found because the input value bindings may not be consistent. In this case it just proceed as `exists`.

$$R[\![\texttt{d->exists!(v|}c\texttt{)}]\!](\sigma,\tau) = \begin{cases}R[\![c \texttt{ and not (v=null)}]\!](\sigma[\texttt{v=d.}k],\tau[\texttt{v=d.}k]) & \\ \qquad \text{if } \exists k \in dom(\texttt{d}^\sigma) : E[\![c]\!](\sigma[\texttt{v} \mapsto \texttt{d}^\sigma.k]) & \\ R[\![\texttt{d->exists(v|}c\texttt{)}]\!](\sigma,\tau) & \\ \qquad \text{else}\end{cases}$$

Restricting Fixing Behavior The constructs `protect`, `test` and `not` restrict its inner constraint from taking some fixing actions. These constructs are needed because sometimes we may want to reduce the fixing behavior. For example, it is possible that in `a=b`, `a` is considered as a source while `b` is considered as a read-only view where only source updates can be propagated to views and view updates cannot affect source. In this case we want to protect `a` from being modified by the fixing function of `a=b`.

- The "`protect v in` c" statement protects a variable from being modified by c. If c changes the variable, the `protect` statement will return $\perp$.

$$R[\![\texttt{protect v in } c]\!](\sigma,\tau) = \begin{cases}R[\![c]\!](\sigma,\tau) & U[\![\texttt{v}^{R[\![c]\!](\sigma,\tau)}]\!](\texttt{v}^\sigma) = \texttt{v}^{\tau(\sigma)} \\ \perp & \text{otherwise}\end{cases}$$

- The `test` construct protects all variables in the inner constraint. This construct is useful when we build an `or` constraint and we want to test some condition without changing anything.

$$R[\![\texttt{test } c]\!](\sigma,\tau) = \begin{cases}\tau & E[\![c]\!](\tau(\sigma)) \\ \perp & \neg E[\![c]\!](\tau(\sigma))\end{cases}$$

- The operator `not` reverses a constraint. A constraint containing `not` is usually unfixable because we may face infinite choice of actions. For example, if "`not a=b`" is violated, we can change `a` and `b` to any pair of values that is not equal, and a fixing function cannot decide one. Nevertheless, `not` is still useful in testing conditions, so in Beanbag we define `not` in a similar way to `test`, where the fixing function simply returns $\perp$ when the constraint is not satisfied.

$$R[\![\texttt{not } c]\!](\sigma,\tau) = \begin{cases} \tau & \neg E[\![c]\!](\tau(\sigma)) \\ \perp & E[\![c]\!](\tau(\sigma)) \end{cases}$$

The `let` construct In previous part we have mentioned the fixing semantics of expressions can be reduced to an equality constraint in the "`v=`e" form. This reduction is done by the two `let` constructs when the constructs connect expressions and constructs together.

- The constraint "`let v=`e `in` c" is similar to "`v=`e `and` c" because it establishes the relations of both e and c. In the latter e becomes an equality constraint `v=`e. Since all expressions will eventually connect to a constraint by `let`, all expressions can be reduced to a equality constraint in this way.

  However, one notable difference between the above two constraint is that the `let` constraint has an inner variable `v` that initially has no bounded value. We must first set a proper value on `v` so that we can invoke the fixing functions of e and c. If e can be evaluated under the input value bindings, we produce the value by just evaluating e. If e cannot be evaluated (e.g., `k` is bound to `null` in `d.k`), we simply set the value of `v` to `null` to indicate an unknown value. After the value of `v` is properly set, we proceed to use the fixing function of `and`.

$$R[\![\texttt{let v=}e\texttt{ in } c]\!](\sigma,\tau) = \\ \begin{cases} R[\![\texttt{v=}e\texttt{ and } c]\!](\sigma[\texttt{v} \mapsto val], \tau[\texttt{v} \mapsto \texttt{void}]) & val \neq \perp \\ R[\![\texttt{v=}e\texttt{ and } c]\!](\sigma[\texttt{v} \mapsto \texttt{null}], \tau[\texttt{v} \mapsto \texttt{void}]) & \text{otherwise} \end{cases} \\ \text{where } val = E[\![e]\!](\sigma)$$

- The statement "`let v=`e `in` e" will also be reduced to the "`v=`e" form and we define its fixing semantics using the previous `let` construct.

$$R[\![\texttt{v}_1\texttt{=(let v}_2\texttt{=}e_1\texttt{ in } e_2\texttt{)}]\!](\sigma,\tau) = \\ R[\![\texttt{let v}_2\texttt{=}e_1\texttt{ in v}_1\texttt{=}e_2]\!](\sigma,\tau)$$

Recursion for More Involved Fixing Recursion is important to the description power of a language as it allows us to iterate over a recursive

structure of a language. Beanbag supports recursion by allowing us to define named constraints (called relations) and named expressions (called functions). We have seen `C1` and `C1onAll`, which are two examples of relations. Relations and functions can both be recursively called. For example, we can check if a class is not inherited from a particular class using the following code.

```
def check(class, parentRef, model) :=
    test class.parent = null or
    (not class."parent" = parentRef and
     check(model.(class."parent"), parentRef, model))
```

6.2.4 Examples

In this section we give a few examples to show how to write Beanbag programs in practice. First, let us implement the same synchronization behavior for relation C2 as IBM RSA: 1) users cannot set the `sender`/`receiver` feature of a message to `null`, and 2) when a class instance in the sequence diagram is deleted, delete the connected messages. These requirements also correspond to the `composite` property in the MOF model. The Beanbag program is as follows.

```
def C2onAll(model) :=
 model->forall(obj |
  isTypeOf(obj, "Message", metamodel)
   and not model.(obj."sender") = null
   and not model.(obj."receiver") = null
  or not isTypeOf(obj, "Message", metamodel)
  or obj = null)
```

The program uses `forall` to check all objects and within `forall` there are three constraints connected by `or`. The first constraint deals with `Message` objects and requires their `sender` and `receiver` features not to be `null`. The second constraint deals with non-`Message` objects and the third constraint deals with object deletion. The third constraint is actually included in the second, but it can take a synchronization action (setting `obj` to `null`) while the second cannot.

When users try to change, for example, the `sender` feature to `null`, none of the three constraint is able to reestablish the relation (the first two constraints have no synchronization action to take and the last one cannot change `obj` to `null` because of PRESERVATION) and the synchronization function will return $\perp$ to denote the update is not allowed. Now suppose users try to delete a class instance. When we visit to a message connected to the class

instance, the first constraint will fail because the referred object is `null` and no synchronization action can be taken. The second constraint will also fail because it has no associated synchronization action. Finally, the third constraint will set `obj` to `null` to delete the message. In this way we can ensure all connected messages are deleted when a class instance is deleted.

The second example shows how to customize synchronization behavior using `or`. Suppose we have a set of object that may be persistent. If an object is persistent, it must be assigned to a persistent container. As a result, when a persistent container is deleted, we may have multiple actions to take on the persistent object belonging it. 1) We may delete the persistent objects. 2) We may simply change the `persistent` attribute of these object to `false`. The following program implements the first option.

```
def persistentConsistent(objs, model) :=
 objs->forall(obj |
  obj."persistent" = true
   and not model.(obj."persistentContainer") = null
  or obj = null
  or obj."persistent" = false
   and obj."persistentContainer" = null)
```

This program has a similar structure to the first one. We use three constraints to deal with three different situations: the object is persistent, the object is deleted and the object is not persistent. When a persistent container is removed, the second constraint will delete the objects belonging to it. If we want to instead change the `persistent` attribute of these object, we can just swap the last two constraints. After swapping, the attribute-changing constraint will have higher priority to the object-deleting constraint and the synchronization function will change the attribute rather than delete an object.

Finally, let us construct the full program for relation C1. The program in Section 6.1 will always insert a new operation to resolve an inconsistency. However, if the inconsistency is caused by changing the receiver of a message, changing the base type of a class instance, or deleting an operation, we would prefer to set the name of the affected message to `null` to indicate that it does not related to an operation. If users rename an operation in the class diagram, we would prefer to rename the related messages accordingly. The following program implements this synchronization behavior.

```
def C1(msg, model) :=
 let rec = model.(msg."receiver") in
 let opRefs = model.(rec."base")."operations" in
  protect model in
```

```
  (opRefs->exists!(r | msg."name"=model.r."name")
   and not msg."name" = null)
or msg."name"=null
or (opRefs->exists(r | model.r."name"=msg."name")
    and not msg."name" = null)
```

This program connects three constraints using the `or` operator. The first constraint protects `model` so that updates are only propagated from operations to `msg`. The second constraints force the message name to `null`. The last one is similar to the first but it does not protect `model`. When we rename an operation in a class diagram, the first constraint will propagate the update to the related messages. If we change the receiver of a message, change the type of a class instance, or delete an operation, the first constraint will fail because we cannot insert a new operation, and the second constraint will set the message name to `null`. If we rename a message to a new name, the first two will both fail and the third constraint will insert a new operation. In addition, we can still customize the synchronization behavior of renaming a message by changing the last "`exist`" to "`exist!`".

6.3 Properties

One important question to ask is whether the semantics of Beanbag satisfies the three properties for operation-based synchronizers. If we consider the consistency relation is defined by the checking mode of Beanbag and a general update consists of a dictionary-based value and a dictionary-based update (cf. Section 3.4.1), the three properties can be redefined on Beanbag constraints.

Property 10 (Consistency for Beanbag constraints)**.**
$R[\![c]\!](\sigma, \tau) = \tau' \Longrightarrow E[\![c]\!](\tau'(\sigma))$

Property 11 (Preservation for Beanbag constraints)**.**
$R[\![c]\!](\sigma, \tau) = \tau' \Longrightarrow \forall var \in dom(\tau) : var^{\tau} \sqsubseteq var^{\tau'}$

Property 12 (Stability for Beanbag constraints)**.**
$E[\![c]\!](\sigma) \wedge \tau(\sigma) = \sigma \Longrightarrow R[\![c]\!](\sigma, \tau)(\sigma) = \sigma$

Based on these property definitions, we can prove the following theorem.

Theorem 6.1. *Any Beanbag constraint satisfies* CONSISTENCY, PRESERVATION *and* STABILITY.

Proof. As the full proof is very long, here we only outline the proof. The basic procedure is to use structural induction over the syntax rules for constraints. We first prove that the primitive constraints satisfy the three properties, and then prove other constructs will ensure the three properties when

their sub-constraints satisfy the properties. Most of the syntax rules can be straightforwardly proved by checking the semantics definition, but there are two issues needed to be addressed. First, the `let` statement contains an expression and we need to know the properties of expressions before we discuss `let`. By using structural induction on expressions, we can see that there is a expression counterpart for each property on constraint.

Property 13 (Consistency for expressions)**.**
$R[\![\texttt{v=}e]\!](\sigma,\tau) = \tau' \Longrightarrow E[\![e]\!](\tau'(\sigma)) = \texttt{v}^{\tau'(\sigma)}$

Property 14 (Preservation for expressions)**.**
$R[\![\texttt{v=}e]\!](\sigma,\tau) = \tau' \Longrightarrow \forall var \in dom(\tau) : var^{\tau} \sqsubseteq var^{\tau'}$

Property 15 (Stability for expressions)**.**
$E[\![e]\!](\tau(\sigma)) = v \wedge \texttt{v}^{\tau(\sigma)} = v \Longrightarrow (\texttt{v}^{R[\![\texttt{v=}e]\!](\sigma,\tau)})(\texttt{v}^{\sigma}) = v$

Then we can reason the `let` statement using the above properties.

Second, several constraints and expressions use recursive calls to reach a fixed point. To satisfy STABILITY, we must ensure such a fixed point always exist under the precondition of STABILITY. A fixed point exists if the function is increasing and has a upper bound. Because of PRESERVATION, the input updates must be included in the output. In this sense the fixing function is increasing. Because all inner constraints and expressions satisfy STABILITY, no variables will be changed when the precondition of STABILITY is satisfied. As a result, the updates on the variables cannot grow beyond the size of the bound values, and thus a fixed point always exists under the precondition of STABILITY. □

Although the three properties are satisfied, it is possible that the fixing function of a Beanbag constraint does not terminate for some input when the precondition of STABILITY is not satisfied. For example, the synchronization function $R[\![\texttt{a."x"=b and b."x"=a}]\!]$ does not terminate for a input (σ,τ) where $\texttt{a}^{\tau} = \mathsf{void}$ and $b = \{\}$. However, such a non-terminating Beanbag program often involves some counter-intuitive constraints (e.g., the example constraint is universally invalid) and is rarely encountered in practice. Based on our experience, *most Beanbag programs in practice always terminate.*

6.4 Evaluation

Since Beanbag satisfies the correctness properties, in the evaluation we focus on the expressiveness and usability. We collected 32 consistency relations from the MOF standard [OMG02] and 34 consistency relations on UML

models from Alexander Egyed who used the relations to evaluate their synchronization action generation work [Egy07, ELF08][1]. From the 66 relations we identify 18 relations that can be automatically established through synchronization actions. (Some consistency relations can be automatically established without synchronization actions, and these relations are not counted. For example, the name of a class should not be `null`. We can simply disallow users to change the name to `null`.) These consistency relations range from high level semantic relations like C1 to low-level syntactic relations like C2. For some relations, we also designed multiple synchronization behavior for each of them. As a result, we have the requirements for 24 Beanbag programs.

Then we proceed to implement these programs in Beanbag to see whether Beanbag is expressive enough for MOF and UML models. The result is positive. We have successfully implemented 17 programs, that is, about 71% of all programs. This result shows that *although Beanbag is not expressive enough for any synchronization behavior, it can support many scenarios and is useful in practice.*

Reviewing the 7 unimplemented programs, we noticed that one program can be implemented with a trivial extension to Beanbag: a function counting the number of entries in a dictionary with no synchronization action need. The other 6 programs need a non-trivial, yet small extension to Beanbag: the ability to access the key when iterating entries in `forall`. This observation shows that *the problems on expressiveness are not fundamental.* All the 7 programs can be implemented by extensions under the basic philosophy of Beanbag: attaching synchronization actions to primitive constraints and expressions, and composing them using high-level constructs.

On the whole, the development of Beanbag program is much easily than manually implementing the synchronizer. A Beanbag program is usually much shorter than a manually implemented synchronizer, and Beanbag ensures CONSISTENCY, PRESERVATION and STABILITY of a program, which already eliminates many bugs.

However, during our development we also identified several problems on usability. First, Beanbag only ensures the correctness of the output updates, but does not ensure the existence of an output. It is up to the programmers to ensure the primitive constraints and functions are composed correctly so that the synchronization function will not return $\perp$ for a proper input. As the interaction among constraints and expressions may be complex, it sometimes needs quite some efforts to achieve this. Second, the synchronization behavior

[1] In their publications they only mentioned 24 relations, but actually they have 34 relations in total.

involving inconsistent data is sometimes difficult to analyze. When we take inconsistent data into account, the domain of the synchronization function becomes much larger. It is sometime very difficult to consider all situations. One possible solution to the two problems is to find some design patterns of Beanbag. We leave this for future work.

One issue often discussed in synchronization is how to deal with the interaction of consistency relations [NEF03, ELF08]. In Beanbag this is handled by connecting all consistency relations by "`and`" in a proper order. If one relation propagates user updates to some other locations, other related relations will be invoked and further propagate the updates to more locations.

6.5 Summary

In this chapter we have presented a novel language, Beanbag, for developing on-site synchronizers. Beanbag attaches synchronization actions to primitive constraints and functions, and composing them through logic operators and other high-level constructs. As a result, one Beanbag program has two meanings: one for defining a relation over data, and one for defining a synchronizer that establish the relation over data by automatically propagating updates. Our study has shown that this approach greatly eases the development of on-site synchronizers and can support many, though not all, useful fixing scenarios in practice.

Chapter 7

Implementation and Application of Beanbag

In the previous chapter we have seen the syntax and the semantics of the Beanbag language, and how this language is effective for describing the synchronization behavior and implementing synchronizers. However, although the semantics are given formally in mathematical terms, it is not quite clear how to convert these mathematical terms into programs so that we can compile a Beanbag program into an executable piece of software. In this chapter we discuss the key issues in the implementation of Beanbag.

One of the key issues in implementing Beanbag is how to implement synchronization functions. In the definition of Beanbag semantics, synchronization functions are partial functions. The function returns a set of update bindings when the input updates are possible to be synchronized and is undefined when the input updates conflict. The domains of constraints constructed using logic operators and quantifiers often depend on the domains of their inner synchronizers. For example, $R[\![\texttt{a and b}]\!]$ is defined only when $R[\![\texttt{a}]\!]$ and $R[\![\texttt{b}]\!]$ are defined in the series of synchronization invocations.

There can be many ways to represent a synchronization function in a common programming language like C or Java. For example, we can represent a synchronization function as a procedure returning a new set of update bindings, where the procedure throws an exception when the input updates conflict. We can also let the procedure return `bool` and modify the input updates at the spot. Furthermore, we can use two exception-free procedures, one for testing whether the input will lead to an output and one for returning output updates. Nevertheless, whichever method we use, we must ensure the synchronization function is free of side-effect when the synchronization fails. In other word, when the synchronization fails, the input value bindings and update bindings should not be modified.

This property is not easy to achieve. In the semantics definitions, many language constructs have to apply a series of modifications to the input update bindings to reach the output. If any of these steps fail, the whole synchronization fails. To recover the original input update bindings, we need to either 1) backup the update bindings at the start of the procedure, or 2) undo all the applied modifications when we encounter a failure.

The first option is not preferred because of performance. In a large application, it is possible that many parts of a large model is updated and the input update bindings may be large. It may take too much time if we copy the whole update bindings every time we invoke a synchronization procedure. The second option solves the performance problem, as in most cases the size of the modifications on the update bindings is smaller than the that of the whole bindings. However, this option requires a lot more efforts on implementation. We have to develop an inverse operation for every modification operation, and be very careful to ensure each operation applied is correctly undone in every synchronization function.

In this chapter we solve this dilemma by implementing Beanbag in Haskell [BW88], a functional programming language. Different from traditional imperative languages, a modification operation in Haskell does not change the data in memory. Instead, it stores the update operation, and execute the operations only when the data is really needed. Intuitively, the new data is stored as a link to the old data and the operation that changes the old data to the new data. As a result, a copy operation in Haskell is very cheap because we only need to store an "identity" operation and a link to the old value. The execution time is constant regardless of the size of the copied value. Using Haskell does not only provide an elegant solution for representing synchronization functions. The closeness of Haskell to the mathematical terms also greatly facilitates the implementation, making the implemented code similar to the semantics definitions.

Although we can provide a clean solution in Haskell, people may prefer implementations in other languages so that the synchronizers can be better integrated into their system. In this chapter we also discuss how to ensure the synchronization function is free of side effect at failures when using an imperative language. The basic idea is to use aspect-oriented techniques to reduce the development cost in the second option. Aspect-oriented extensions widely exist in popular imperative languages [KHH+01, RS03, ALS08]. Using our reusable aspects, one can effective avoid the extra care taken in the implementation of synchronization functions and only have to implement an inverse operation for every operation used.

More concretely, the contributions of this chapter can be summarized as follows.

- We present an implementation of a Beanbag compiler in Haskell. This implementation features clean and correct code, and the full code is available at the Beanbag website [Xioa].
- We discuss the implementation issues in imperative languages. We use aspect-oriented techniques to reduce the development cost and show a reusable aspect written in AspectJ [KHH+01].
- We present an application where we use Beanbag to build a multi-view modeler. This application shows how to use beanbag in productive cases.

The rest of this chapter is organized as follows. We first introduce our Haskell implementation in Section 7.1. We classify the implementation into three parts and introduce them one by one. We start with the dictionary-based data and updates (Section 7.1.1), build synchronizers upon them (Section 7.1.2), and then compile Beanbag code into Haskell code that invokes these synchronizers (Section 7.1.3). We then discuss the implementation issues in Java in Section 7.2. Finally, we present the application in Section 7.3.

7.1 Implementing in Haskell

7.1.1 Dictionary-based Data and Updates

The key concept used in dictionary-based data and updates is dictionary. The structured data are described by dictionaries. The updates on dictionaries are described by dictionaries. Furthermore, the sets of value bindings and update bindings can be also considered as dictionaries mapping variables to values/updates. To implement dictionary-based data and updates, the first thing we should do is to implement a reusable data structure for dictionaries.

Haskell already provides a data structure, `Data.Map`, that maps keys to values. `Data.Map` is a suitable candidate for implementing dictionaries, but there is a gap between `Data.Map` and dictionaries. All keys that are not in the domain of a dictionary has a default value. For example, the dictionary value has a default value of `null`. The dictionary updates have a default value of void. The value bindings and update bindings also have corresponding default values. `Data.Map` does not support default values, and we have to implement this mechanism ourselves.

To support default values, we define a data type `DValMap`, as shown in Figure 7.1. The name `DValMap` is an abbreviation of the default value map. This map contains an ordinary map and a default value. Three basic operations are provided for `DValMap`: `get` for getting a value at specified key, `set`

```
data DValMap k a = DValMap (Map k a) a
    deriving (Eq)

get :: (Ord k) => (DValMap k v) -> k -> v
get (DValMap theMap value) k =
    Map.findWithDefault value k theMap

infixl 9 #
(#) :: (Ord k) => (DValMap k v) -> k -> v
(#) = get

set :: Ord k => Eq v =>
    (DValMap k v) -> k -> v -> (DValMap k v)
set (DValMap theMap value) k v
    | value == v && Map.notMember k theMap =
        DValMap theMap value
    | value == v && Map.member k theMap =
        DValMap (Map.delete k theMap) value
    | otherwise =
        DValMap (Map.insert k v theMap) value

infixl 8 \\
(\\) dict (k, v) = set dict k v

getDefaultValue (DValMap theMap value) = value

foldWithKey :: Ord k => (k->v->a->a)->a->DValMap k v->a
foldWithKey func initValue (DValMap theMap value) =
    (Map.foldWithKey func initValue theMap)
```

Figure 7.1: The data type `DValMap`

for changing a value at a specified key and `getDefaultValue` for returning the default value. When we get a value from the map, `get` returns the value at the input key when the key exists in the map and returns the default value when the key does not exist. When we set a value at a key, `set` will compare the value with the default value. If it is the default value and the key does not exist in the dictionary, `set` returns the original dictionary. If the key exist, `set` deletes the key to free up space. If the value is not the default value, `set` proceeds like normal `Map.insert`. We also define two operators, `#` and `\\`, for users to easily access the map.

Other functions in `Data.Map` can also be adapted to the default value version. In Figure 7.1 we show an example function `mapWithKey`. Compare to normal `mapWithKey` function, this function takes an extra parameter, the default value, and construct the target map with the default value. Note the target map is constructed with `\\` instead of `Data.Map` to avoid storing the default value into the target map.

Based on `DValMap`, we can build values, updates, value bindings and update bindings. Their definitions are shown in Figure 7.2. These definitions just follow the mathematical definitions in Chapter 3. A value is either `Prim` or `Dict` where `Prim` defines primitive values including `Null` and `Dict` is a `DValMap` mapping `Prim` to `Value` with a default value `Null`. An update is either a primitive update (`PUpdate`), a dictionary update (`DUpdate`) or `Void`, where `PUpdate` is simply `Prim` and `DUpdate` is a `DValMap` mapping `Prim` to `Value` with a default value `Void`. The set of bindings is called environment in the implementation. Value environment `ValEnv` is a `DValMap` mapping variables to values with a default value `Null` while update environment `UpdateEnv` is a `DValMap` mapping variables to updates with a default value `Void`.

Several other functions and operators are also defined for manipulate these data types, including function `apply` that applies an update to a value, operator `<*` that merges two updates, function `findUpdate` that returns the minimal updates between two values. Because the definitions of these functions simply follow the formal definition we presented in Chapter 3, we only list their types here.

7.1.2 Constraints and Expressions

Based on the definition of the data and updates, we turn to the implementation of constraints and expressions. One option is to use the same method in the implementation of data and updates, declaring a data type of synchronizers. The following code shows this option.

```
data Synchronizer = Equal Var Var
```

```
-- Values
data Prim = Int Int | String String | Null
    deriving (Eq, Ord)
type Dict = DValMap Prim Value
data Value = Prim Prim | Dict Dict
    deriving (Eq)

emptyDict = empty (Prim Null)

-- Updates
type PUpdate = Prim
type DUpdate = DValMap Prim Update
data Update = PUpdate PUpdate | DUpdate DUpdate | Void
    deriving (Eq)

emptyDUpdate = empty Void

apply :: Update -> Value -> Value
    -- apply an update to a value

(<*) :: Update -> Update -> Update
    -- merge an update with another update, where the
    -- first update is considered to be applied earlier

findUpdate :: Value -> Value -> Update
    -- find the minimal update that changes the first
    -- value into the second value

-- Variables
data Var

string2Var :: String -> Var
    -- create a variable from a string

-- Environments (sets of bindings)
type ValEnv = DValMap Var Value
type UpdateEnv = DValMap Var Update
```

Figure 7.2: The basic definitions of values and updates

```
    | And Synchronizer Synchronizer
    | Or Synchronizer Synchronizer
    | ...
```

However, to implement the checking semantics and the synchronization semantics, we have to create functions where each function deals with all of these constructors, as the following code shows.

```
check (Equal v1 v2) env = env#v1 == env#v2
check (And s1 s2) env = check s1 env && check s2 env
check (Or s1 s2) env = check s1 env || check s2 env
...
```

Because there are a number of constructs for constraints and expressions, this leads to large functions that are difficult and error-prone to implement. If we miss one synchronizer in the function definition, the system will only check and report the error during the runtime. Furthermore, this causes maintenance problems. Every time we want to add a new construct, we have to modify the functions. Omission of modifying a function is also only checked at runtime.

To avoid these problems, we capture constraints and expressions as classes. A class captures the common features of a set of types. In this way, when we add a new constraint or a new expression into the system, we only need to define a new data type and declare the new data type is an instance of the corresponding classes. The large functions are now implemented in small pieces under the instance declaration, and the original maintenance problem is avoided.

The definitions are shown in Figure 7.3. Each constraint has two functions. Function `check` takes the constraint and a set of value bindings, and returns a boolean value to indicate whether the constraint is satisfied or not. Function `sync` takes the constraint, a set of value bindings and a set of update bindings, and returns a new set of update bindings to make data consistent.

The definition of `Expression` is more interesting. Besides function `eval` that evaluates the expressions according to the input value bindings, every expression also has a function `esync`, which is similar to `sync`, but takes an extra variable as parameter. This design is originated from the semantics definitions in Chapter 6, where we define the synchronization semantics of expressions by deducing expressions into the $var = expr$ form. In this way an expression plus a variable form a synchronizer.

Based on class `Constraint`, we can define constraints as data types, and declare them as instances of the class. The instance declaration just directly follows the semantics definition in Chapter 6. Here we give two constraints as examples.

```
type Check = ValEnv -> Bool
type Eval = ValEnv -> Maybe Value
type Sync = ValEnv -> UpdateEnv -> Maybe UpdateEnv

class (Show a) => Constraint a where
    check :: a -> Check
    sync :: a -> Sync

class (Show a) => Expression a where
    eval :: a -> Eval
    esync :: a -> Var -> Sync
```

Figure 7.3: The definitions of `Constraint` and `Expression`

```
data EqualCstraint = EqualCstraint Var Var

instance Constraint EqualCstraint where
    check (EqualCstraint v1 v2) env =
        (env # v1) == (env # v2)
    sync (EqualCstraint v1 v2) vEnv uEnv = do
        u <- syncTwoValues (vEnv # v1)
                    (vEnv # v2) (uEnv # v1) (uEnv # v2)
        return (uEnv \\ (v1, u) \\ (v2, u))

syncTwoValues v1 v2 u1 u2
    | compatible u1 u2 =
        Just (u1 <* u2 <* findUpdate v1 new_v)
    | otherwise = Nothing
    where new_v = apply (u1 <* u2) v2
```

Figure 7.4: The implementation of `v1=v2`

```
data (Constraint c1, Constraint c2) =>
    AndCstraint c1 c2 = AndCstraint c1 c2

applyUntilEqual :: (Monad m, Eq a) => (a->m a)->a->m a
applyUntilEqual f v = do
    v' <- f v
    if v' == v then return v' else applyUntilEqual f v'

instance (Constraint c1, Constraint c2) =>
    Constraint (AndCstraint c1 c2) where
    check (AndCstraint c1 c2) env =
        (check c1 env) && (check c2 env)
    sync (AndCstraint c1 c2) vEnv uEnv =
        applyUntilEqual f uEnv
        where
        f uEnv' = (sync c1 vEnv uEnv') >>= (sync c2 vEnv)
```

Figure 7.5: The implementation of `c1 and c2`

Figure 7.4 shows the first example, the equal constraint. The constructor takes two variables and this constraint ensures the two variables are equal. Function `check` just checks if the two variables are equal according to `env`. Function `sync` uses function `syncTwoValues` to synchronize the two updates, and `syncTwoValues` is defined according to the definition of u in the semantics definition. In `sync`, we treat the returned `Maybe` values as monads. In this way we can keep the code in a clean flow while ensuring `Nothing` is returned whenever a step fails.

Figure 7.5 shows the second example, the `and` operator. The definition first requires `c1` and `c2` are constraints, and then declares `AndCstraint c1 c2` as a constraint. Functions `check` and `sync` just follow the semantics definitions. Function `check` checks if both `c1` and `c2` are satisfied while `sync` repeatedly applies `c1` and `c2` until we reach a fixed point.

Similarly, we can define expressions based on class `Expression`. Here we give one example, the constant expression, in Figure 7.6. The data constructor takes only a constant as parameter, function `eval` returns the constant, and the function `sync` changes `var` to be equal to the constant.

In Chapter 6, we assume all bound variables have different names from free variables when defining the semantics. This assumption is useful in simplifying the definition, but in actual implementation we must handle this issue. We introduce an extra variable replacement step to handle this issue. We explain this using the `Let` statement as an example. The other constructs involving bound variables, `forall`, `exists` and `exists!`, are implemented

```
data ConstExpr = ConstExpr Value

instance Expression ConstExpr where
    eval (ConstExpr c) _ = Just c
    esync (ConstExpr c) var vEnv uEnv
        | compatible u (uEnv # var) =
            return (uEnv \\ (var, u <* uEnv # var))
        | otherwise = fail ""
        where
            u = findUpdate (apply (uEnv # var) (vEnv # var)) c
```

Figure 7.6: The implementation of the constant expression

in a similar way. The implementation code of `let` is shown in Figure 7.7.

The synchronization semantics of `let` is first to find the value of the bound variable, and then proceed like `and` by repeatedly invoking the inner expression and the inner constraint. To invoke the inner expression, we introduce `ExprCstraint` to wrap an expression into a constraint. The constructor of `ExprCstraint` takes a variable and an expression to construct a constraint. Its `check` function checks the value of the variables is equal to the value returned by the expression. Its `sync` function invokes `esync` by passing the variable to the `esync`.

We implement `let` using `ExprCstraint`. The data type definition and function `check` are same as the semantics definitions. Function `sync` first evaluates the expression, then uses `applyUntilEqual` to repeated apply `invokeOnce` to reach the fixed point. Function `invokeOnce` first invokes the expression and then invokes the constraint, where each invocation is through the function `invoke`. Function `invoke` performs the replacement step. It takes a constraint to be invoked, a bound variable, the environment, and the value and the update to be set to the bound variable. It changes the value and the update on the variable according to the input before invocation, and restores the original update after the invocation. In this way we can ensure that the value and the update in the environment are correctly changed according to the scope of the variable even if there is a free variable having the same name as the bound variable,

Another issues we have omitted in the formal semantics definition is the named relation and recursive relation references. In imperative languages, named relation are often implemented by keeping a lookup table of relation definitions and each relation reference is a pointer to a definition in the lookup table. In Haskell, this implementation can be greatly simplified by the lazy evaluation mechanism of Haskell. Haskell evaluates a value only

```
data (Expression expr) =>
    ExprCstraint expr = ExprCstraint Var expr
    deriving (Show)

instance (Expression expr) =>
    Constraint (ExprCstraint expr) where
    check (ExprCstraint var e) env = case eval e env of
        Just c -> c == env # var
        Nothing -> False
    sync (ExprCstraint var e) vEnv uEnv =
        esync e var vEnv uEnv


data (Expression expr, Constraint cst) =>
    LetCstraint expr cst = LetCstraint Var expr cst

instance (Expression expr, Constraint cstraint) =>
    Constraint (LetCstraint expr cstraint) where
    check (LetCstraint var expr cst) env =
        let evalResult = eval expr env in check' evalResult
        where
            check' (Just v) = check cst (env \\ (var, v))
            check' Nothing = False
    sync (LetCstraint var expr cstraint) vEnv uEnv =
        let replaced =  case eval expr vEnv of
                Just v -> v
                Nothing -> Prim Null
        in do
        (_, _, result) <- applyUntilEqual invokeOnce
                          (replaced, Void, uEnv)
        return result
        where
        invokeOnce e = invokeExpr e >>= invokeCstraint
        invokeExpr = invoke (ExprCstraint LetVar expr) LetVar
        invokeCstraint = invoke cstraint var
        invoke c var (v, u, uEnv) = do
            resultEnv <-
                sync c (vEnv \\ (var, v)) (uEnv \\ (var, u))
            return (v, resultEnv # var,
                      resultEnv \\ (var, uEnv # var))
```

Figure 7.7: The implementation of the `let`statement

when needed, and thus can support recursive data definitions. Consequently we can simply define relations as Haskell values and define recursive relations as recursive values. For example, a Beanbag relation

```
def c1(a, b) := a = b
```

can be implemented as

```
c1 a b = EqualCstraint a b
```

When this relation is referenced in other places, the parameters, `a` and `b`, will be replaced by the concrete arguments, and the relation reference thus has the same semantics of defining the relation in place.

Now let us consider a recursive relation, `def c2(a, b) := a = b and c2(a, b)`. This is clearly a non-terminating relation, but it is enough for illustration purpose here. Following the above strategy, the implementation code should be:

```
c2 a b = AndCstraint (EqualCstraint a b) (c2 a b)
```

This code looks correct. According to the lazy evaluation mechanism of Haskell, the enclosed `c2` definition will be unfolded only when needed. However, when we compile this code, we will get the following message.

> Occurs check: cannot construct the infinite type:
> c2 = AndCstraint EqualCstraint c2

This is because although Haskell supports infinite values, it does not support infinite types. The type of `c2` is defined by the type of `AndCstraint`. However, as the type of `AndCstraint` is parameterized on its operands, the compiler needs the type of `c2` to calculate the type of `AndCstraint`.

To solve this problem, we have to break the dependence chain on type deduction, either to make `c2` not depend on the type of `AndCstraint` or to make `AndCstraint` not depend on the type of `c2`. Fortunately, there is one extension to Haskell, the existential type provided by the GHC compiler [Jon96], supporting removing type dependence. For example, we may define the following existential type, where the type parameter on the right does not appear on the left.

```
{-# OPTIONS_GHC -fglasgow-exts #-}
data DynCstraint = forall a. Constraint a => DynCstraint a
```

This code is similar to `data DynCstraint a = DynCstraint a` but the type is always `DynCstraint` regardless of the type of `a`. The trade-off is that we can only use `a` as an instance of `Constraint` and can never know the

concrete type of `a`. However, this is enough for our purpose. We can use `DynCstraint` to wrap a recursive relation and unwrap the inner relation during synchronization. To make the process easier, we can further define `DynCstraint` as an instance of `Constraint`.

```
instance Constraint DynCstraint
    where
        check (DynCstraint c) = check c
        sync (DynCstraint c) = sync c
```

Using `DynCstraint`, we can define the previous `c2` relation as follows.

```
c2 = AndCstraint EqualCstraint (DynCstraint c2)
```

Finally, to make the construction of Beanbag program easier, we provide a set of auxiliary functions as shown in Figure 7.8. Using these auxiliary functions, we can write Beanbag program directly in Haskell using a Beanbag-like syntax. For example, the constraint

```
(let c = "1" in a = c) or (a = b)
```

can be written in Haskell as

```
letc "c" (intConst 1) ("a" <=> "c") <|> "a" <=> "b"
```

7.1.3 Compiler

Although we provide functions for writing Beanbag program in Haskell, the ultimate goal of our implementation is to convert a Beanbag program into a Haskell program so that we can invoke the program to check data or synchronize updates. In our implementation of Beanbag we have implemented a compiler that converts a Beanbag source file into a Haskell source file that uses the above functions to define Beanbag relations in Haskell. This compiler is implemented using the lexical analyzer generator Alex [Mar] and the parser generator Happy [GM].

One interesting issue is after we compile the generated Haskell file, how we interact with the program to check or to synchronize models. In the implementation we use standard input and output to pass values. In this way if users integrate the synchronizer into an application not written in Haskell, they can redirect the standard input and output, avoiding more expensive operations, e.g., file operations.

Suppose the Beanbag program is

```
main(a, b) := a == b
```

and the compiled file is `equal.exe` (on Windows), we can invoke the checking mode using the following command.

```
-- v1 = v2
infixl 5 <=>
(<=>) :: String -> String -> EqualCstraint
-- c1 and c2
infixl 4 <&>
(<&>) :: (Constraint c1, Constraint c2) =>
    c1 -> c2 -> AndCstraint c1 c2
-- c1 or c2
infixl 3 <|>
(<|>) :: (Constraint c1, Constraint c2) =>
    c1 -> c2 -> OrCstraint c1 c2
-- d->forall(v|c)
forall :: (Constraint c) =>
    String -> String -> c -> ForAllCstraint c
-- d->exists(v|c)
exists :: (Constraint c) =>
    String -> String -> c -> ExistsCstraint c
-- d->exists!(v|c)
exists' :: (Constraint c) =>
    String -> String -> c -> Exists'Cstraint c
-- test c
test :: Constraint c => c -> TestCstraint c
-- protect v in c
protect :: (Constraint c) =>
    String -> c -> ProtectCstraint c
-- not c
notc :: Constraint c => c -> c
-- let v = e in c
letc :: (Expression e, Constraint c) =>
    String -> e -> c -> LetCstraint e c
-- constant
intConst :: Int -> ConstExpr
stringConst :: String -> ConstExpr
nullConst :: ConstExpr
-- d.k
infixl 6 <.>
(<.>) :: String -> String -> DictGetExpr
-- let v = e in e
lete :: (Expression expr1, Expression expr2) =>
    String -> expr1 -> expr2 -> LetExpr expr1 expr2
```

Figure 7.8: Auxiliary functions for building Beanbag constraints

```
>equal.exe --check
```

The program will then wait for you to input the data for the `main` relation. We can input the data using the syntax we presented in the previous chapter, and Beanbag will prompt whether the constraint is satisfied or not.

```
>{a=1, b=1}
The constraint is satisfied.
```

Alternatively, if we provide no argument for the executable file, the program will enter synchronization mode. The program will wait for us to input two lines of text, where the first line is interpreted as the value bindings and the second line is interpreted as the update bindings. After typing the two lines, the program output the updates to make the variables consistent.

```
>{a=1, b=1}
>{a->2}
output updates:   {$a->2,$b->2}
```

We can further ask the program to output the updated data (using parameter "–updatedValues") as well as the effective update (using parameter "–effectiveUpdates") that actually changes the data.

```
>equal.exe  --updatedValues --effectiveUpdates
>{a=1, b=2}
>{}
output updates:   {$a->2,$b->2}
effective updates:{$a->2,}
output values:    {$a->2,$b->2}
```

It is worth remarking the output of the compiled program is always through a standard format. If a user wants to integrate this program into an application written in another programming language, it should be easy for him to write code to interact the Beanbag program through standard IO, generating input and parsing output.

7.2 Imperative Implementation Issues

Although the Haskell implementation is easy and clean, people may still want to implement Beanbag in other languages so that the generated synchronizers can be better integrated into applications written in other languages. Another reason for implementing Beanbag in other languages is performance. Haskell language is designed to be a high-level language independent of hardware model of computation. Optimizing the performance of Haskell programs often requires the knowledge of particular implementations and is very difficult in general.

As we have discussed, one of the most difficult issue in implementing Beanbag is how to ensure the synchronization function is free of side-effect at failures while ensuring efficiency both in development and in execution. One method is to copy the update environment every time, which is efficient in development but is not efficient in execution. Another one is to directly change the input update environment to reach the output. Every time we change the update environment, we store an inverse operation to cancel the operation, and when the synchronization fails, by throwing an exception or returning a failure flag, we invoke all stored operations to roll back the updates.

This second method has better performance, but every time we implement a synchronization function, we have to store and inverse operations, which is a development-intensive task. Here we use aspect-oriented techniques to solve this problem. We capture the repeated code as a reusable aspect and let aspect-orient compiler to automatically weave the aspect into the implementation code. Figure 7.9 shows the pseudo code of the aspect. The syntax is borrowed from AspectJ [KHH+01], but the concepts are general and can be applied to other aspect-oriented languages.

We assume class `Operation` is the base class of all operations applied on an update environment, where its `apply` method changes the update environment, and its `getInverse` method returns an inverse operation to cancel the operation. Note here we are talking about the update operations on update environments, and should be distinguished from updates on values. Nevertheless, as the update environments share the similar dictionary structure as values, the update operations on the environments can be similar implemented.

The aspect contains two abstract pointcuts, where `callSynchronize` is supposed to capture the join points where the synchronization functions are called, and `modifyUpdateEnvironment` is supposed to capture the join points where the update environment is changed by users.

We use a stack of stack to store operations, where the external stack is to distinguish different invocations in nested invocations. A stack of operations is created before any invocation to the synchronization function, and is popped out after the invocation. Every time an operation is to be applied, we push the inverse of the operation into the stack at the top of the external stack. If the synchronization function fails, the operations in the top stack are applied in the inverse order to cancel the changes. If the synchronization function succeeds, the operation in the top of the stack is popped without applications. One special case is that the external stack is not empty when the synchronization successfully returns. In other words, the synchronization function is invoked by some external synchronization function. We need

```
public abstract aspect StateAspect {
  abstract pointcut callSynchronize();
  abstract pointcut modifyUpdateEnvironment(Operation op);

  Stack<Stack<Operation>> opeartionStack;

  before() : callSynchronize() {
      operationStack.push(new Stack<Operation>());
  }

  after() returning : callSynchronize() {
    Stack<Operation> ops = operationStack.pop();
    if (!operationStack.empty()) {
        operationStack.peek().addAll(ops);
    }
  }

  after() throwing : callSynchronize() {
    Stack<Operation> ops = operationStack.pop();
    while(!ops.emtpy())
      ops.pop().apply();
  }

  before(Operation op) : modifyUpdateEnvironment(op) {
    operationStack.peek().push(op.getInverse());
  }
}
```

Figure 7.9: The pseudo code for the state aspect

to append all operations to the stack of the external function so that the operations are successfully canceled when the external functions fails.

7.3 Application

In software engineering, there exist many applications that Beanbag can be applied to. Examples include synchronizing multi-views in visual language editors [GHZL06], integration of heterogeneous tools [Tra05], synchronizing software architecture and runtime system [HMY06], and etc. We have successfully applied Beanbag to several case studies. In this section we describe one application - a multi-view Enterprise JavaBean (EJB) modeling tool. This application shows a typical architecture of integrating a Beanbag synchronizer into an application. We implement this application using a Java implementation of a previous version of Beanbag [XZH+08] that is slightly different from the version described in the thesis. However, the techniques described here also apply to the new language and the Haskell implementation.

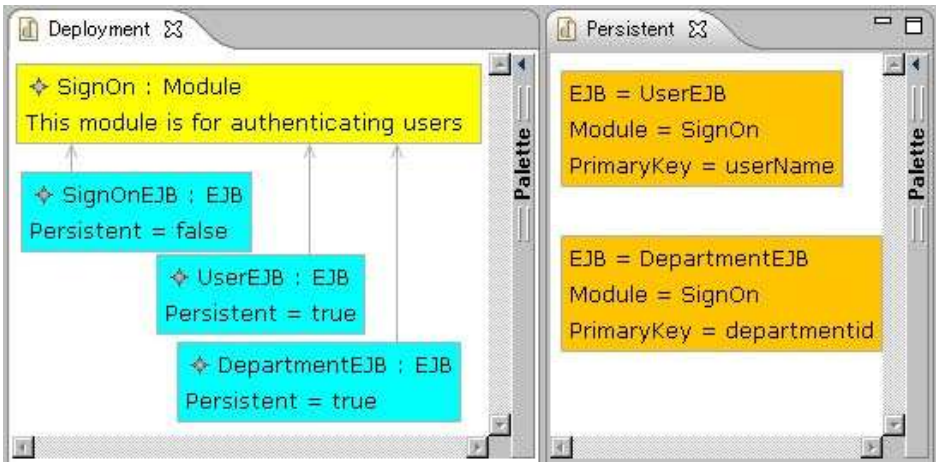

Figure 7.10: An EJB modeling tool

Figure 7.10 shows the interface of the EJB modeling tool. The tool provides two types of editable diagrams: the deployment diagram and the persistent diagram. The deployment view shows how EJBs are organized into modules, while the persistent diagram shows a list of persistent EJBs (entity beans). In the figure there are three EJBs: `SignOnEJB`, `UserEJB`, and `DepartmentEJB`, all of which belong to a module `SignOn`. The persistent attributes of `UserEJB` and `DepartmentEJB` are `true`, indicating they are entity

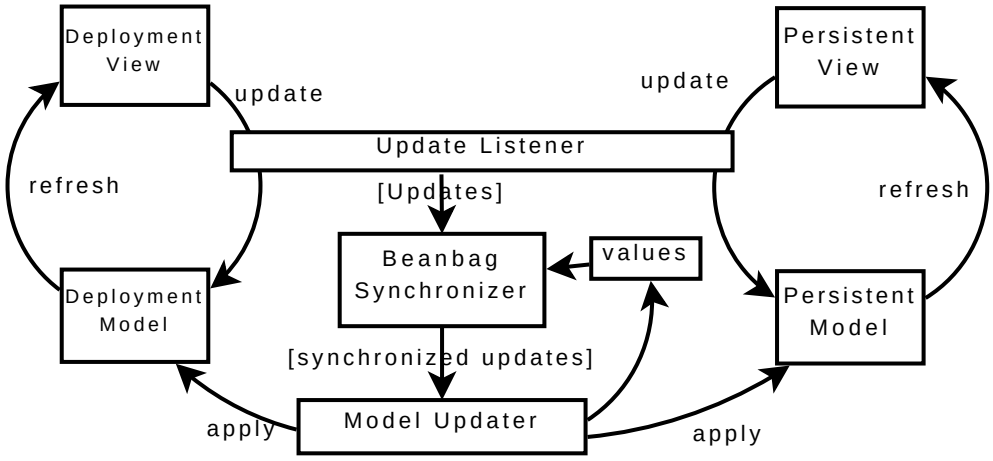

Figure 7.11: The architecture of the EJB tool

beans and are listed in the persistent diagram. For each entity bean, we list its EJB name, its module name and its primary key.

Several consistency relations exist over the two diagrams. For example, the EJB name and the module name of an entity bean should be equal the names in the deployment diagram. An EJB should only exist when there is a module. We capture the consistency relation between the two diagram by a Beanbag program.

The main components of the tool are editing components generated by Eclipse Graphical Modeling Framework (GMF) [Ecl08] and a synchronization component generated from the Beanbag program, and we only write a few hundred lines of Java code to glue them together.

GMF is a framework for generating graphical editors. Given a model definition, a view definition and their mappings, GMF generates a graphical view that reads from and writes to the model. GMF can generate multiple views for one model, but in a quite limited way: the views and the model cannot be structurally different, and multiple views cannot be edited at the same time. As the two views in the EJB modeling tool are structurally different (one hierarchical and one flat), the tool cannot be directly generated by GMF.

Therefore we discard the usual way of generating two views for one model. Instead, we generate two editors, each with an independent model. The two models can be structurally different and their consistency is maintained by a Beanbag synchronizer. On the interface side, the two editors are both integrated into Eclipse and act like one application.

The architecture of our implementation is shown in Figure 7.11. Besides

the two diagrams, we also keep a set of dictionary-based values representing the contents of the diagrams. Initially, the values are empty dictionaries, corresponding to empty diagrams. When users update a model, we capture the updates by an update listener. When the two views need to be synchronized (when users explicitly request synchronization or, more automatically, whenever users update a model), we pass the updates and the values to the synchronizer. After synchronization, a model updater updates the models as well as the values according to the output. By keeping the dictionary-based values, we avoid converting models into dictionaries, saving both the development cost and the execution time.

One issue of implementing the update listener is how to convert the GMF updates to the Beanbag format. The GMF updates refer to objects through the in-memory addresses, but in Beanbag we generate unique keys for each object. To convert the object addresses into Beanbag keys, we keep a bijective mapping between the keys and the in-memory addresses of objects. Because the generated keys are just integers, we can easily save the mapping with models using the serialization support of GMF, ensuring that the object addresses are always valid.

This small technique has great value in practice. To identify objects in state-based synchronization, users are often required to designate some key attributes [Obj08][BFP+08] whose values are unique among all instances. However, based on our experience, many application data do not have a suitable candidate to be a key attribute [YKW+08]. On the other hand, as operation-based synchronizers are tightly integrated into the system, we can directly use the in-memory address and get rid of the key attribute.

Chapter 8

Concluding Remarks

In this thesis we propose a language-based approach for model synchronization. We give synchronization semantics to high-level specification languages and derive synchronizers from high-level specifications in these languages. In particular, for off-site synchronization, we show that synchronizers can be derived from a unidirectional transformation program by recording an executable trace. We also show that a bidirectional model transformation can be wrapped into a synchronizer by a model difference operation. For on-site synchronization, we propose a first-order logic language to write synchronizer, showing that complex synchronization behavior can be specified by assigning synchronization semantics to (mainly) the original constructs in first-order logic. We also show that this language can be compiled into an efficient incremental synchronizer to ensure short synchronization time in practice. All these techniques are built upon our theoretical foundation framework for model synchronization, which includes three properties to ensure the correctness of synchronization and discusses the relation between operation-based synchronizer and state-based synchronizer. In addition, all the languages and algorithms have been implemented and have shown their usefulness in practical cases.

In the following, we highlight some interesting future work.

Conflict Management

Through this thesis, conflicts are treated in a simplified way. In the theoretical foundation, conflicts are defined by the union operation and the synchronization function. This definition does not really impose the requirement of conflict handling on these operations. One can, for example, define a function that is undefined at all input.

In the languages and algorithms for synchronization, we only report the

existence of conflict and provide no support for conflict-resolving. This makes it difficult to resolve conflicts in some situations. For example, it is possible that two users have edited the models a lot before synchronization in a distributed environment. If there is a conflict, users have to check through all edited parts to find the conflict updates.

To support real world synchronization work, the current approach needs to be augmented with conflict management. In the theoretical foundation, we need to clearly define what a conflict is and requires the union operation and synchronization function to capture conflicts. As a conflict in heterogeneous synchronization is often related to the consistency relations considered, we need to build a model to capture updates and consistency relations together. In addition, as the existence of conflicts differs when we consider different sets of updates, we probably need to neglect the current black-box way of treating updates as single units and defining consistency using a subset, and use a white-box means so that we can analyze the structure of updates and the consistency relations. One possible way to achieve this is to capture consistency relations as classic logic expressions and use paraconsistent logic to build a model that connects the classic logic expression and conflicts.

In the languages and algorithms, we need to add the capability of handling conflicts. Grundy et al. [GHM98] discuss a set of requirements for conflict management. Besides the detection of conflicts, we need at least to represent the conflicts and the reason of conflicts to users, interact with users to resolve the conflicts and support negotiation if there is more than one user involved in conflict resolution. Most of these requirements are related to the definition of conflicts and are highly promising to be satisfied if we build a well formal foundation of conflicts.

Handling Ordered Data

The approach in this paper depends on the dictionary-based representation of models and updates. We show how to represent most concepts of models in dictionaries but so far we have not developed a method to represent order attributes in dictionaries. This is a pragmatic simplification so that we can focus on other aspect of synchronization by considering only a small set of data types. However, as ordered data plays an important role in many systems, we need to handle ordered data to make our synchronization system practical.

One possible way of handling ordered data is to represent ordered data in dictionaries. Foster et al. [FGM+07] discuss a method to represent ordered data as a dictionary of two entries where the first entry is the first item in the list and the second entry is the rest of the list. For example, a se-

quence of two elements, `<"e1", "e2">`, can be represented as {`head->"e1", tail->{head->"e2", tail->null}`}. However, on this representation it is difficult to define some common list operations like removing an item at an index using the current dictionary updates.

Another possible way is to develop a new data representation which contains ordered data. However, how to represent updates on order data, how to calculate the union of updates so that the result preserves both updates, and how to detect conflicts between updates are all unsolved problem. Both semantic foundation and practical algorithm are needed for a ordered data structure.

Bibliography

[AAAN+06] Marwan Abi-Antoun, Jonathan Aldrich, Nagi Nahas, Bradley Schmerl, and David Garlan. Differencing and merging of architectural views. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, pages 47–58, Washington, DC, USA, 2006. IEEE Computer Society.

[AC06] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *Proc. 9th MoDELS*, pages 692–706, 2006.

[AC08] MichałAntkiewicz and Krzysztof Czarnecki. Design space of heterogeneous synchronization. *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 3–46, 2008.

[ALS08] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[AP03] Marcus Alanen and Ivan Porres. Difference and union of models. In *Proceedings of the 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML'03)*, volume 2863/2003 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin / Heidelberg, 2003.

[Arn96] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[Bal91] Robert Balzer. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[BBF06] N. Bencomo, G. Blair, and R. France. Summary of the workshop Models@run.time at MoDELS 2006. In *Lecture Notes in Computer Science, Satellite Events at the MoDELS 2006 Conference, LNCS,*, pages 226–230, 2006.

[BBM03] F. Budinsky, S.A. Brodsky, and E. Merks. *Eclipse modeling framework*. Pearson Education, 2003.

[BBS01] G.J. Badros, A. Borning, and P.J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[BC95] Robert H. Bourdeau and Betty H.C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, 1995.

[BCRP98] G.S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998.

[BDE+05] AW Brown, M. Delbaere, P. Eeles, S. Johnston, and R. Weaver. Realizing service-oriented solutions with the IBM rational software development platform. *IBM systems journal*, 44(4):727–752, 2005.

[BFBW92] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and symbolic computation*, 5(3):223–270, 1992.

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proc. 35th POPL*, 2008.

[BG98] M. Bickford and D. Guaspari. Lightweight analysis of UML. Technical report, TM-98-0036, Odyssey Research Associates, Ithaca, NY, 1998.

[BGF+08] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 811–814, 2008.

[BMS08] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4), June 2008. Earlier version in Proc. 10th International Workshop on Database Programming Languages, DBPL '05, Springer-Verlag LNCS vol. 3774.

[Bor81] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[BRST05] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model transformations in practice workshop. In *Satellite Events at MoDELS*, pages 120–127, 2005.

[BS81] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[BW88] R. Bird and P. Wadler. *An introduction to functional programming*. Prentice Hall International (UK) Ltd. Hertfordshire, UK, UK, 1988.

[CC03] Alvin T. S. Chan and Siu-Nam Chuang. MobiPADS: a reflective middleware for context-aware mobile computing. *IEEE Trans. Softw. Eng.*, 29(12):1072–1085, 2003.

[CD+99] J. Clark, S. DeRose, et al. XML path language (XPath) version 1.0. *W3C recommendation*, 16:1999, 1999.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. GRACE meeting notes, state of the art, and outlook. In *International Conference on Model Transformations (ICMT), Zurich, Switzerland*, June 2009. Invited paper. To appear.

[CK01] M.V. Cengarle and A. Knapp. A formal semantics for OCL 1.4. *Lecture notes in computer science*, pages 118–133, 2001.

[CRE06] Antonio Cicchetti, Davide Di Ruscio, and Romina Eramo. Towards propagation of changes by model approximations. In *International Workshop on Models for Enterprise Computing, In Proc. EDOC*, 2006.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[CRP08] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing model conflicts in distributed development. In *Proc. 11th MoDELS*, pages 311–325. Springer, 2008.

[DB82] U. Dayal and P.A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 8(3):381–416, 1982.

[DDH01] AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. *SIGMOD Rec.*, 30(2):509–520, 2001.

[Dis08] Zinovy Diskin. Algebraic models for bidirectional model synchronization. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 21–36, Berlin, Heidelberg, 2008. Springer-Verlag.

[Ecl08] Eclipse Consortium. The Eclipse Graphical Modeling Framework. `http://www.eclipse.org/modeling/gmf/`, 2008.

[Ede93] D. Vera Edelstein. Report on the ieee std 1219–1993—standard for software maintenance. *SIGSOFT Softw. Eng. Notes*, 18(4):94–95, 1993.

[EGSK07] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 64–73, 2007.

[Egy07] Alexander Egyed. Fixing inconsistencies in UML design models. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 292–301, Washington, DC, USA, 2007. IEEE Computer Society.

[ELF08] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and evaluating choices for fixing inconsistencies in

UML design models. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 99–108. IEEE, 2008.

[EMRP01] Todd Ekenstam, Charles Matheny, Peter Reiher, and Gerald J. Popek. The bengal database replication system. *Distrib. Parallel Databases*, 9(3):187–210, 2001.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[FGH+94] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng.*, 20(8):569–578, 1994.

[FGK+05] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Schema-directed data synchronization. Technical Report MS-CIS-05-02, University of Pennsylvania, March 2005. Supersedes MS-CIS-03-42.

[FGM+05] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: a linguistic approach to the view update problem. In *POPL '05 : ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 2005.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE) in ICSE '07*, pages 37–54, 2007.

[FW04] D.C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October, 2004. *World Wide Web Consortium*, pages 0–20041028, 2004.

[GCH+04] David Garlan, ShangWen Cheng, AnCheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[GHM98] John Grundy, John Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998.

[GHZL06] John C. Grundy, John G. Hosking, Nianping Zhu, and Na Liu. Generating domain-specific visual language editors from high-level tool specifications. In *Proc. 21st ASE*, pages 25–36, 2006.

[GM] Andy Gill and Simon Marlow. Happy: The parser generator for haskell. http://www.haskell.org/happy/.

[HMY06] Gang Huang, Hong Mei, and Fu-Qing Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Eng.*, 13(2):257–281, 2006.

[ISO06] ISO. International Standard - ISO/IEC 14764 IEEE Std 14764-2006. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998*, pages 1–46, 2006.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[JK06] Frdric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2006.

[JOn] JOnAS Project, `http://jonas.objectweb.org`. *Java Open Application Server*.

[Jon96] S.L.P. Jones. Compiling Haskell by program transformation: A report from the trenches. *Lecture Notes in Computer Science*, pages 18–44, 1996.

[Kel85] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS '85: Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 154–163, New York, NY, USA, 1985. ACM.

[KH06] Shinya Kawanaka and Haruo Hosoya. biXid: a bidirectional transformation language for XML. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 201–214, 2006.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, pages 327–353, 2001.

[KKP07] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of diff3. In Arvind and Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 485–496, December 2007.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[KM07] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE) in ICSE*, pages 259–268, 2007.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218, New York, NY, USA, 2001. ACM.

[LDGR04] Michael Lawley, Keith Duddy, Anna Gerber, and Kerry Raymond. Language features for re-use and maintainability of MDA transformations. In *Workshop on Best Practices for Model-Driven Software Development*, 2004.

[LHG07] Na Liu, John Hosking, and John Grundy. Maramatatau: Extending a domain specific visual language meta tool with a declarative constraint mechanism. In *Proceedings of 2007 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007.

[LHT07] Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of XQuery. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 21–30, New York, NY, USA, 2007. ACM.

[LNH+07] Dongxi Liu, Keisuke Nakano, Yasushi Hayashi, Zhenjiang Hu, Masato Takeichi, Akimasa Morihata, and Yingfei Xiong. Bi-X core: A general-purpose bidirectional transformation language.

In *Proceedings of the 24th Japan Society for Software Science and Technology Conference (JSSST'07)*. JSSST, 2007.

[Mar] Simon Marlow. Alex: A lexical analyser generator for haskell. http://www.haskell.org/alex/.

[Mas84] Yoshifumi Masunaga. A relational database view update translation mechanism. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 309–320, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[Mee98] L. Meertens. Designing constraint maintainers for user interaction. `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`, 1998.

[MGH05] Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2005. ACM Press.

[MHN+07] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 47–58, New York, NY, USA, 2007. ACM.

[MOSMI03] Pascal Molli, Gérald Oster, Hala Skaf-Molli, and Abdessamad Imine. Using the transformational approach to build a safe and generic data synchronizer. In *GROUP '03: Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pages 212–220, New York, NY, USA, 2003. ACM.

[MT86] C.B. Medeiros and F.W. Tompa. Understanding the implications of view update policies. *Algorithmica*, 1(1):337–360, 1986.

[NEF03] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 455–464, Washington, DC, USA, 2003. IEEE Computer Society.

[Obj06] Object Management Group. Object constraint language specification 2.0. `http://www.omg.org/spec/OCL/2.0`, 2006.

[Obj07] Object Management Group. XML metadata interchange specification. `http://www.omg.org/docs/formal/07-12-01.pdf`, 2007.

[Obj08] Object Management Group. MOF query / views / transformations specification 1.0. `http://www.omg.org/docs/formal/08-04-03.pdf`, 2008.

[OMG02] OMG. MetaObject Facility specification. `http://www.omg.org/docs/formal/02-04-03.pdf`, 2002.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 177–186, 1998.

[OMT08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Companion version*, pages 899–910, 2008.

[PLA] PLASTIC Team, `http://www.ist-plastic.org/`. *The PLASTIC Platform.*

[PSR+89] G. Priest, R. Sylvan, R. Routley, J. Norman, and A.I. Arruda. *Paraconsistent logic: essays on the inconsistent.* Philosophia Verlag Gmbh, 1989.

[RS03] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. *ACM SIGSOFT Software Engineering Notes*, 28(5):297–306, 2003.

[SAG+06] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Trans. Softw. Eng.*, 32(7):454–466, 2006.

[SBP08] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. Using components for architecture-based management: the self-repair case. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 101–110, 2008.

[SK03] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[SK08a] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *Proc. 4th ICGT*, pages 411–425, 2008.

[SK08b] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *Proc. of the 4th International Conference on Graph Transformation*, pages 411–425, 2008.

[SMFBB93] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software-Practice and Experience*, 1993.

[SMSJ03] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between UML models. In *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, Proceedings*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2003.

[SSZH07] Hui Song, Yanchun Sun, Li Zhou, and Gang Huang. Towards instant automatic model refinement based on OCL. In *APSEC07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 167–174, 2007.

[Ste07] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, pages 1–15, 2007.

[Sut95] Jeff Sutherland. Business objects in corporate information systems. *ACM Comput. Surv.*, 27(2):274–276, 1995.

[SXH+08] Hui Song, Yingfei Xiong, Zhenjiang Hu, Gang Huang, and Hong Mei. A model-driven framework for constructing runtime architecture infrastructures. Technical Report GRACE-TR-2008-05, Center for Global Research in Advanced Software Science and Engineering, National Institute of Informationtics, Japan, Dec 2008. http://grace-center.jp/downloads/GRACE-TR-2008-05.pdf.

[Tea] Atlas Team. The ATL web site. http://www.eclipse.org/m2m/atl/.

[Tra05] Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.

[Tra08] Laurence Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–126, March 2008.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[XHZ+09] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. Supporting automatic model inconsistency fixing. In *Proceedings of 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) (to appear)*, August 2009.

[Xioa] Yingfei Xiong. The Beanbag website. `http://www.ipl.t.u-tokyo.ac.jp/~xiong/beanbag.html`.

[Xiob] Yingfei Xiong. The SyncATL website. `http://www.ipl.t.u-tokyo.ac.jp/~xiong/modelSynchronization.html`.

[XLH+07] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 164–173, New York, NY, USA, 2007. ACM.

[XS05] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[XSHT09] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. Supporting parallel updates with bidirectional model transformations. In *Proceedings of the Second International Conference*

on Model Transformation (ICMT'09) (to appear). Springer, 2009.

[XZH+08] Yingfei Xiong, Haiyan Zhao, Zhenjiang Hu, Masato Takeichi, Hui Song, and Hong Mei. Beanbag: Operation-based synchronization with intra-relations. Technical Report GRACE-TR-2008-04, Center for Global Research in Advanced Software Science and Engineering, National Institute of Informationtics, Japan, Dec 2008. http://grace-center.jp/downloads/GRACE-TR-2008-04.pdf.

[YBP+04] F. Yergeau, T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0. *W3C Recommendation*, 2004.

[YKW+08] Yijun Yu, Haruhiko Kaiya, Hironori Washizaki, Yingfei Xiong, and Zhenjiang Hu. Enforcing a security pattern in stakeholder goal models. In *Proceedings of the 4th Workshop on Quality of Protection*, 2008.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 371–380, 2006.