SEQUENCE DIAGRAMS INTEGRATION VIA TYPED GRAPHS: THEORY AND IMPLEMENTATION

by

HONGZHI LIANG

A dissertation submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
August 2009

Copyright © Hongzhi Liang, 2009
Abstract

It is widely accepted within the software engineering community that the support for integration is necessary for requirement models. Several methodologies, such as the role-based software development, that have appeared in the literature are relying on some kind of integration. However, current integration techniques and their tools support are insufficient. In this research, we discuss our solution to the problem. More precisely, we present a general integration approach for scenario-based models, particularly for UML Sequence Diagrams, based on the colimit construction known from category theory.

In our approach, Sequence Diagrams are represented by SD-graphs, a special kind of typed graphs. The merge algorithm for SD-graphs is an extension of existing merge operations on sets and graphs. On the one hand, the merge algorithm ensures traceability and guarantees key theoretical properties (e.g., “everything is represented and nothing extra is acquired” during the merge). On the other hand, our formalization of Sequence Diagrams as SD-graphs retains the graphical nature of Sequence Diagrams, yet is amenable to algebraic manipulations. Another important property of our process is that our approach is applicable to other kinds of models as long as they can be represented by typed graphs.

A prototype Sequence Diagram integration tool following the approach has been
implemented. The tool is not only a fully functional integration tool, but also served as a test bed for our theory and provided feedback for our theoretical framework. To support the discovery and specification of model relationships, we also present a list of high-level merge patterns in this dissertation.

We believe our theory and tool are beneficial to both academia and industry, as the initial evaluation has shown that the ideas presented in this dissertation represent promising steps towards the more rigorous management of requirement models.

We also present an approach connecting model transformation with source transformation and allowing an existing source transformation language (TXL) to be used for model transformation. Our approach leverages grammar generators to ease the task of creating model transformations and inherits many of the strengths of the underlying transformation language (e.g., efficiency and maturity).
Co-Authorship

Some parts of Chapters 3, 4 and 5 are based on joint work with my supervisor Dr. Juergen Dingel, Dr. Zinovy Diskin and Dr. Ernesto Posse, and were previously published at the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008) [LDDP08] and the 1st International Conference on Software Language Engineering [LD08].
Acknowledgments

This research is supported by the Centre of Excellence for Research in Adaptive Systems (CERAS) and IBM, and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

I would like to thank my supervisor, Dr. Juergen Dingel, for his supportive advises over this research and guidance on the writing of this dissertation.

I would like to thank Dr. Zinvoy Diskin, for his assistance with all the details about Category Theory.

I would like to thank my other thesis committee members, Dr. Thomas Dean and Dr. Hossam Hassanein, for their time, knowledge and opinions.

To my father Kunwu, my brothers Hongyi and Hongxuan, and my entire family for your long time support.

To my daughter Emily and my son Justin for the happiness you give me.

To my wife Yuan for your love.

In memory of my mom, Guiying Wang.
Statement of Originality

Most of the material in this dissertation is based on existing papers. Chapter 3 is based on [DDL06], to which my personal contribution was approximately 10%, and on [LDDP08], to which my personal contribution was approximately 35%. Chapter 4 is based on [LD08], to which my personal contribution was approximately 95%. The work reported on in Chapters 5 and 6 is entirely my own. Any published (or unpublished) ideas and/or techniques of others are fully acknowledged in accordance with the standard referencing practices.

Hongzhi Liang
February 2009
# Table of Contents

Abstract ................................................. i

Co-Authorship ........................................ iii

Acknowledgments ........................................ iv

Statement of Originality ............................... v

Table of Contents ...................................... vi

List of Tables .......................................... ix

List of Figures .......................................... x

Chapter 1:

<table>
<thead>
<tr>
<th>Introduction ........................................... 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Problem ........................................... 3</td>
</tr>
<tr>
<td>1.2 Dissertation statement ................................ 4</td>
</tr>
<tr>
<td>1.3 Contributions ...................................... 5</td>
</tr>
<tr>
<td>1.4 Structure of the dissertation ........................ 6</td>
</tr>
</tbody>
</table>
### Chapter 2:

<table>
<thead>
<tr>
<th>Background</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Category theory</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Model-Driven Engineering</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Source transformation via TXL</td>
<td>22</td>
</tr>
</tbody>
</table>

### Chapter 3:

<table>
<thead>
<tr>
<th>Sequence Diagrams integration via SD-graphs</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Sequence Diagrams as SD-graphs</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Transformations between Sequence Diagrams and SD-graphs</td>
<td>31</td>
</tr>
<tr>
<td>3.3 Merging SD-graphs</td>
<td>37</td>
</tr>
<tr>
<td>3.4 Scenario integration: An example</td>
<td>38</td>
</tr>
<tr>
<td>3.5 Scenario integration: General process</td>
<td>48</td>
</tr>
<tr>
<td>3.6 Advantages and limitations</td>
<td>48</td>
</tr>
</tbody>
</table>

### Chapter 4:

<table>
<thead>
<tr>
<th>Model Transformation with TXL</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Model transformation via TXL</td>
<td>54</td>
</tr>
<tr>
<td>4.2 Model transformation example</td>
<td>59</td>
</tr>
<tr>
<td>4.3 Advantages and limitations</td>
<td>64</td>
</tr>
<tr>
<td>4.4 Related work</td>
<td>66</td>
</tr>
</tbody>
</table>

### Chapter 5:

<table>
<thead>
<tr>
<th>A tool for Sequence Diagram integration</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Architecture</td>
<td>70</td>
</tr>
<tr>
<td>5.2 Implementation</td>
<td>72</td>
</tr>
</tbody>
</table>
5.3 Testing and evaluation ........................................ 88

Chapter 6:

Merge in practice .............................................. 90
6.1 Merge patterns ................................................ 91
6.2 Pattern applications ......................................... 117

Chapter 7:

Related work ...................................................... 128

Chapter 8:

Future work and conclusions ................................. 132
8.1 Limitations and future work .............................. 132
8.2 Conclusion ..................................................... 138

Bibliography ....................................................... 140

Glossary .......................................................... 150
List of Tables

6.1 The mapping pairs required for the sequential case. . . . . . . . . . . . . . 97
6.2 The mapping pairs required for the multiple meet operators example. 107
6.3 The mapping pairs required for the insertion example. . . . . . . . . . . . 112
6.4 The mapping pairs required for the refinement example. . . . . . . . . . . 116
6.5 The mapping pairs required for the extended brokered sale example. . 121
6.6 The mapping pairs required for the HACS example. . . . . . . . . . . . . . 127
List of Figures

2.1 An example of Fin. ............................................. 8
2.2 Graphs (a) and graph homomorphisms (b). ...................... 9
2.3 An example of Gra. ............................................. 10
2.4 Integration of objects $S_1, S_2$ modulo the span. ............... 13
2.5 The universal property. ....................................... 13
2.6 A sample Class Diagram. ..................................... 16
2.7 An Object Diagram showing an instance of the Class Diagram in Fig. 2.6. 17
2.8 A sample Sequence Diagram. .................................. 19
2.9 An overview of TXL processor .................................. 23
2.10 The grammar of an example TXL program. ..................... 23
2.11 The rules of an example TXL program. ........................ 25
3.1 A sample sequence diagram. ................................... 27
3.2 An example of SD-graphs. ..................................... 28
3.3 Category with partial arrows in (a), and category with total arrows in (b). .................................................. 37
3.4 The span of the $WS_{CG}$ and $RS_{CG}$ class graphs and the merged graph on top. .................................................. 40
3.5 The span of the \( WS_{OG} \) and \( RS_{OG} \) object graphs and the merged graph on top. .......................................................... 43

3.6 Setting the “horizontal” arrow \( h_{SN} \). .............................................. 44

3.7 The span of the sequence graphs \( WS_{SG} \) and \( RS_{SG} \) to be merged. .... 46

3.8 Result of merge as a typed graph (a) and as Sequence Diagram (b) . 47

4.1 An overview of model transformation via TXL. ................................. 55

4.2 The FamilyMModel is shown as a class diagram and a serialized Ecore meta-model. ................................................................. 60

4.3 The PersonsMModel is shown as a class diagram and a serialized Ecore meta-model. ................................................................. 61

4.4 The generated persons.grammar. ..................................................... 62

4.5 The TXL program for family to persons model transformation. ......... 63

4.6 An example of input Family model. .................................................. 64

4.7 The output Persons model. ............................................................... 64

5.1 The architecture of the prototype SD Integration Tool. ...................... 71

5.2 The meta-model of SD-graphs. ........................................................ 73

5.3 Scoping and scheduling rules. ........................................................ 75

5.4 A transformation rule. ................................................................. 76

5.5 The mapping meta-model. ............................................................. 77

5.6 The mapping editor. ................................................................. 79

5.7 The graphical editor of SD-graphs. ................................................. 81

5.8 The Java interfaces for category theory. ......................................... 82

5.9 An arrow between two graphs contains conflict mappings. ............. 83

5.10 The screen shots of the Sequence Diagrams Integration Eclipse Wizard. 86
6.1 Example showing a Sequence Diagram divided by InteractionFragments. 92
6.2 Example showing a lifeline divided by LifelineFragments. ............ 93
6.3 Example showing a Sequence Diagram augmented with TimePoints. . 94
6.4 The pattern of the simple composition. ............................ 95
6.5 The sequential case of the simple composition. .................... 96
6.6 The parallel case of the simple composition. ...................... 97
6.7 The general case of the simple composition. ...................... 98
6.8 The adj and the met operators. ................................. 100
6.9 Example showing a merge of LifelineFragments connected by the adj
operator. .......................................................... 101
6.10 The head and the mapping models for the adjacent merge. ....... 102
6.11 Example showing a merge of LifelineFragments connected by a single
met operator. .................................................... 103
6.12 The head and the mapping models for the meet merge case. ...... 104
6.13 Example showing an invalid Sequence Diagram. ................ 105
6.14 Example showing a merge of LifelineFragments connected by the met
operators. ...................................................... 106
6.15 The head and the mapping models for the multiple met operators
example. ...................................................... 106
6.16 Example shows the inserting elements from one diagram to another. 110
6.17 The head and the mapping models for the insertion example. ....... 111
6.18 The head and the mapping models for the adding new elements example. 113
6.19 Example of interaction refinement pattern. ...................... 115
6.20 The head and the mapping models for the refinement example. .... 115
6.21 The extended sale Sequence Diagram. ........................................... 118
6.22 The first and second heads and the mapping models for the extended
brokered sale example. ................................................................. 119
6.23 The final head and the mapping models for the extended brokered sale
example. ..................................................................................... 120
6.24 The result of the extended brokered sale. ............................................. 121
6.25 A login scenario $SD_{LI}$ of HACS and its refinement. ......................... 122
6.26 A logout scenario $SD_{LO}$ of HACS. ................................................. 123
6.27 A check status scenario $SD_{CS}$ of HACS. ......................................... 123
6.28 The head Sequence Diagram $SD_h'$ and mappings for the sequential
composition part of the HACS example. ........................................... 124
6.29 The head Sequence Diagram $SD_h''$ and mappings for the refinement
part of the HACS example. ............................................................. 125
6.30 The combined head $SD_h$ for the HACS example. ................................. 125
6.31 A complete get status scenario of HACS. .......................................... 126
Chapter 1

Introduction

The need to integrate software artifacts seems inherent to modern software development. On the one hand, the development may be distributed over several teams to leverage different expertise, experience or capabilities. On the other hand, breaking a task into smaller, more manageable pieces often is an effective means to deal with the kind of complexity that comes from, e.g., large numbers of stakeholders, views, features, or platforms. In each case, the separately developed artifacts need to be assembled as efficiently as possible into a consistent whole in which the parts still function as described.

While support for integration is required for a large variety of artifacts to, e.g., support separation of development or concerns, it appears particularly necessary for models of requirements. This is because requirements models are especially prone to change and evolution. This phenomenon is widely accepted within the software engineering community and much work has been done to address it.

Many approaches aimed at mitigating the effect of changing requirements appear to rely on some kind of integration. For instance, in [JN04] use case slices need to
be composed, gradually refined and kept synchronized. Consequently, support for the integration of a different, possibly more detailed and separately developed, model into other models is required. In some situations this integration may be adequately realized through a simple kind of replacement operation; however, to achieve the traceability needed for large-scale, distributed development, a less destructive form of integration may be necessary. Moreover, the refined part of a model may have complex relationships with its context that need to be preserved by the refinement.

Another class of approaches is based on role modeling, that is, the identification of the parts of an object that address a particular concern such as performing a task or maintaining an invariant. Changing requirements can then often be dealt with either by adding a role to an object or by modifying a single role without affecting the others. Role-based software development methodologies include Reenskaug’s Object Oriented Role Analysis Method (OOram) [Ree95] and VanHilst’s Role-Oriented Programming (ROP) [Van97]. Both methodologies feature a “synthesis” step in which the implementation of an object is obtained from its roles. Again, possibly separately developed models of requirements describing the roles need to be integrated to produce a description of the overall behavior of the object. Typically, roles are not disjoint, that is, they can “overlap” in complex ways. For instance, two different collaborations may require the same interaction with an object. Indeed, a look at [Ree95, Van97] shows it is exactly these kinds of relationship between roles that complicate the synthesis step. The integration must properly deal with these relationships and, for instance, avoid the creation of duplicated parts in the integrated model in case of overlap among the integration participants.
1.1 Problem

Despite the apparent need for the integration of models of requirements, relatively little concrete support for this activity seems to exist. A lot of the existing work on the topic of model integration either assumes a large degree of disjointness between models (e.g., the work on composition operations for UML Sequence Diagrams and Interaction Overview Diagrams), or targets very specific notations with no clear potential for a more general application. For instance, all three methodologies mentioned above remain relatively silent on how exactly the integration of the requirements models is to be achieved. In [JN04], the composition of use case slices is performed on the code level using, e.g., AspectJ. UML’s package merge is mentioned as a mechanism to compose slices on the model level although package merge is currently not defined on interactions [DDZ08]. Moreover, no indication is given of how synchronization between separately evolving models can be accomplished. OOram cautions the developer to take care that the result of the synthesis model is “consistent with the meanings of all its base models” [Ree95, page 124]. However, no exact definition of the synthesis operation is provided. In [Van97], the transparent composition of roles on the code level is discussed in detail, while the model level is discussed much more informally. For another example, consider the “combine” operation offered in IBM Rational Software Architect (RSA) V7.0 which is intended to support the integration of different models, but appears limited to class and object diagrams [Let].
CHAPTER 1. INTRODUCTION

1.2 Dissertation statement

The dissertation studies the problem of integrating models of requirements expressed in the form of UML Sequence Diagrams. The statement is: **typed graphs and category theory offer a useful mathematical framework for the formalization of Sequence Diagrams and their integration.** The framework can be implemented efficiently in connection with modern software modeling tools, such as IBM RSA.

This dissertation presents the answers to the following questions: How should an integration operation look like for Sequence Diagrams? How could this operation properly deal with overlap between Sequence Diagrams? How could it be supported through tools? And finally, could this operation be generalized and be applicable to other kinds of models?

In particular, we present an approach to integrating a core subset of UML Sequence Diagrams which rests on a well-established theory, is potentially applicable to a large class of models, and supports many different kinds of scenario integration operations. Our approach uses category theory as a mathematical framework in which UML Sequence Diagrams are represented as a particular kind of typed graphs, called **SD-graphs**, and integration is achieved through the explicit representation of the relationships between models via structure-preserving maps and a colimit construction. This construction results in an intuitive and versatile operation which not only provides traceability “for free” (the original models can easily be identified in the integration result), but also serves as an effective mechanism to structure the implementation and implement consistency checking. Moreover, the approach is potentially applicable to other diagrams used to represent requirements (e.g., Message
Sequence Charts, Communication Diagrams). An Eclipse-based implementation of the approach is described.

1.3 Contributions

This dissertation makes a number of research contributions to the current state of the art in model integration. The contributions are:

1. the definition of a formal framework providing the necessary theoretical foundations for Sequence Diagrams integration.

2. the formalization of Sequence Diagrams as SD-graphs. Our formalization retains the graphical nature of Sequence Diagrams, yet is amenable to algebraic manipulations and supports consistency checking.

3. the design and implementation of a prototype Sequence Diagram integration tool that rests on the theoretical foundations of integration. The prototype on the one hand allows us to evaluate our approach. On the other hand, it is a fully functional Sequence Diagram integration tool, and is integrated into the Eclipse framework.

4. the description of a list of high-level merge patterns. The merge patterns allow us to improve the practicality of our current prototype implementation.

5. the practical evaluation of using TXL for model transformation and the description of a process for the application of TXL as a model transformation tool. The process benefits from a number of essential properties, e.g., explicit rule scoping,
flexible rule application strategy and user controllable rule scheduling, provided by TXL.

1.4 Structure of the dissertation

The remainder of this dissertation is structured as follows:

Chapter 2 reviews the necessary background material: category theory, Model-Driven Engineering and source transformation with TXL.

Chapter 3 describes the theory behind our approach to Sequence Diagrams integration. We show how UML Sequence Diagrams are represented as SD-graphs and describe our merge procedure. An example is given to illustrate the merge algorithm. Finally, the scenario integration process is summarized.

Chapter 4 gives a detailed description of our approach to implementing model transformation using TXL. The approach has been used to develop two main components of a Sequence Diagram integration tool.

Chapter 5 shows how a Sequence Diagram integration tool has been built based on the theory discussed in Chapter 3.

Chapter 6 shows how to improve the usability of our general integration process and the prototype implementation in particular through high-level merge patterns. Chapter 7 discusses the related work.

Chapter 8 presents the future work we would like to investigate, and concludes the dissertation.
Chapter 2

Background

2.1 Category theory

Category theory is branch of mathematics that studies mathematical structures and their relationships in an abstract way. Recent research has shown that category theory is also relevant and useful in software engineering. Several texts, e.g., [RB01, Pie91], focus on category theory within the context of computer science. To make the dissertation self-contained, a basic introduction to categories, some examples of categories and merging sets and graphs with the colimit operation of category theory will be given in this section.

2.1.1 Categories

A category $C$ is a structure consisting of two kinds of elements: objects $O$ and arrows (or morphisms) $A$ between the objects. Both objects and arrows can be of any kind. Every arrow $f \in A$ has a unique source object $a \in O$ and a unique target object $b \in O$, 
and can be written as \( f : a \to b \). A category also has to satisfy the following four rules:

**Identity arrow** For each object \( a \in O \), there is an identity arrow \( id_a : a \to a \).

**Composite arrow** For each pair of arrows \( f : a \to b \) and \( g : b \to c \), there is the composite arrow \( g \circ f : a \to c \) in \( A \).

**Identity composition** For each arrow \( f : a \to b \), the following equation must hold

\[
 f \circ id_a = f = id_b \circ f.
\]

**Associativity** For each set of arrows, \( f : a \to b \), \( g : b \to c \) and \( h : c \to d \), the following equation must hold

\[
 h \circ (g \circ f) = (h \circ g) \circ f.
\]

### 2.1.2 Examples

There are many different kinds of categories. Some of them are of importance to this research, i.e., the categories of sets, finite sets and graphs.

![Figure 2.1: An example of Fin.](image)
The category of sets $\textbf{Set}$ consists of sets as its objects and total functions between sets as its arrows. Identity arrows in $\textbf{Set}$ are simply identity functions. Composition of arrows is given by composition of functions. The category of finite sets $\textbf{Fin}$ is a subcategory of $\textbf{Set}$, whose objects are finite sets and arrows are total functions. Consider the example of $\textbf{Fin}$ shown in Fig. 2.1, it has the following finite sets and functions as its objects and arrows, $R = \{1, 2\}$, $S_1 = \{\text{one, two, three, four}\}$, $S_2 = \{\text{un, deux}\}$, $f : R \rightarrow S_1 = \{1 \mapsto \text{one}, 2 \mapsto \text{two}\}$, and $g : R \rightarrow S_2 = \{1 \mapsto \text{un}, 2 \mapsto \text{deux}\}$.

Another example of a category is the category of graphs $\textbf{Gra}$, whose objects are graphs and arrows are graph homomorphisms between the graphs. Basically, a graph consists of nodes and edges. An arrow, i.e., a graph homomorphism in $\textbf{Gra}$ preserves the structure of a graph. The formal definitions of graphs and graph homomorphisms are as follows.

**Definition 1** (Graphs and graph morphisms). A (directed multi)graph is a quadruple $G = (N, E, so, ta)$. The meta-model of graphs is shown in Fig. 2.2(a). A graph consists of a set of nodes $N$, a set of edges $E$ and two functions $so, ta : E \rightarrow N$ to reference the source and target of an edge.
We also write $e: x \to y$ when $so(e) = x$ and $ta(e) = y$. An edge $e$ is called a loop if $so(e) = ta(e)$. Saying “elm is an element of $G$” means $elm \in N \cup E$.

A graph homomorphism or graph mapping $h: G \to G'$ between two graphs $G = (N, E, so, ta)$ and $G' = (N', E', so', ta')$ shown in Fig. 2.2(b) is a pair of functions $h_N: N \to N'$, $h_E: E \to E'$ with the property that for all $e \in E$, $so'(h_E(e)) = h_N(so(e))$ and $ta'(h_E(e)) = h_E(ta(e))$.

An example of $\textbf{Gra}$ is given in Fig. 2.3. It contains three graphs, $G$, $G_1$ and $G_2$ as its objects and two homomorphisms $h_1$ between $G$ and $G_1$, and $h_2$ between $G$ and $G_2$ as its arrows.

![Figure 2.3: An example of $\textbf{Gra}$.

2.1.3 Merging sets and graphs

Category theory offers powerful structure manipulating mechanisms, such as colimit.

With colimit, new objects can be constructed from existing objects. Because our merge algorithm is based on graph merge, and graph merge in turn is based on set merge with colimit, we will give a description of set and graph merge in this section.
Specifying equivalence between sets

Let us consider the two sets $S_1 = \{\text{one, two, three, four}\}$ and $S_2 = \{\text{un, deux}\}$ in Fig. 2.1 on page 8. Assuming elements in sets are just labels for some entities, different labels may refer to the same entity. For example, suppose we know that the elements $\text{one} \in S_1$ and $\text{un} \in S_2$ are the labels of 1, and $\text{two} \in S_1$ and $\text{deux} \in S_2$ are the labels of 2. Formally, equivalence between two sets $S_1$ and $S_2$ is specified by a span $R$: a set $R$ with two projection functions (or set mappings) $f: R \rightarrow S_1$ and $g: R \rightarrow S_2$. In this example, $R = \{1, 2\}$ and the projections are $f = \{1 \mapsto \text{one}, 2 \mapsto \text{two}\}$, and $g = \{1 \mapsto \text{un}, 2 \mapsto \text{deux}\}$. We will refer to the sets $S_1, S_2$ as the hands, the set $R$ as the head, and the projections $f, g$ as the arms of the span.

Merging sets

Without their interconnecting span, the integration of sets $S_1$ and $S_2$ is just the union of the sets $S_1 \cup S_2 = \{\text{one, two, three, four, un, deux}\}$. To take the interconnecting span into consideration, the integration of sets $S_1$ and $S_2$ becomes $S = S_1 \oplus_R S_2$, that is, the integration modulo the span $R$. The integration is achieved through the following process:

- The disjoint union of all participating sets in the span $S' = S_1 \cup S_2 \cup R$ is created. In our case, $S' = \{1, 2, \text{one, two, three, four, un, deux}\}^1$. We have three inclusion functions $f_i: S_1 \rightarrow S'$, $g_i: S_2 \rightarrow S'$ and $h_i: R \rightarrow S'$ between the disjoint union $S'$ and the sets in the span.

- A binary relation $E$ over $S'$ is generated using the projection functions in $R$. In our case, $E \overset{\text{def}}{=} \{(1, \text{one}), (2, \text{two}), (1, \text{un}), (2, \text{deux})\}$.

\[^1\text{For brevity, the indexes for the elements are omitted.}\]
Then, we compute the least equivalence relation $E^*$ containing $E$ over $S'$. For example, $E^*$ in our case is
\[
\{(1, one), (2, two), (1, un), (2, deux), (1,1), (2,2), (one, one), (two, two), (un, un), (deux, deux), (one, 1), (two, 2), (un, 1), (deux, 2), (one, un), (un, one), (two, deux), (deux, two)\}.
\]

The quotient set $P_{E^*}$ (or the set of all equivalence classes) of $S'$ over $E^*$ is produced. The result of the operation is a partition of $S'$. In our case, the partition is a four-element set
\[
P_{E^*} = S'/E^* = \{(1, one, un), (2, two, deux), (three), (four)\}.
\]

We also have a function $q: S' \to P_{E^*}$ that maps each element in $S'$ to its equivalence class in $P_{E^*}$.

Next, $P_{E^*}$ together with inclusion functions $f': S_1 \to P_{E^*}$, $g': S_2 \to P_{E^*}$ and $h': R \to P_{E^*}$ (the composite functions $q \circ f_i$, $q \circ g_i$ and $q \circ h_i$ respectively), form the result of integration.

Finally, we need to uniquely name each set in the partition $P_{E^*}$. For a set with a single element, we can name the set by its element. For a set containing multiple elements, a reasonable agreement is to give priority to the element from the interconnecting set, i.e., to name a multi-element set by its element from $R$. The merged set is then the four-element set $P = \{1, 2, three, four\}$.

If we call a family of arrows with a common target a cospan, the integration procedure can be phrased as building a special cospan $S = (P, f', g', h')$ over the input span $R$. This construction is called the pushout of sets in category theory.
Formally, a pushout of two arrows \( f: R \to S_1 \) and \( g: R \to S_2 \) between objects \( R, S_1 \) and \( S_2 \) in a category is an object \( P \) with two arrows \( f': S_1 \to P \) and \( g': S_2 \to P \) such that the diagram in Fig. 2.4 commutes\(^2\).

Moreover, the following universal property is also satisfied: given any object \( Q \) and arrows \( f'': S_1 \to Q \) and \( g'': S_2 \to Q \), there is a unique \( u: P \to Q \) such that the diagram in Fig. 2.5 commutes.

We emphasize that the actual result of the integration is an arrow configuration shown in Fig. 2.4 rather than just the merged set \( P \). The inclusion functions (or inclusion arrows) \( f', g', h' \) are important. They allow us to trace how the component

\(^2\)A diagram with objects and arrows is commutative if selecting any two objects, any composition of arrows between the two objects leads to the same result as the direct arrows between the objects.
sets are represented in the merged set. Also, because these functions are totally defined, they guarantee that each element in the component sets is represented by an element in the merged set during the integration. Moreover, the universal property of the pushout ensures that the set $P$ is the “least upper bound” of the sets $S_1, S_2$ modulo span $R$, in other words, nothing is added during the integration. Thus, merging sets with a pushout means that “everything is represented and nothing is added”.

In category theory, pushout is an instance of a more general concept, called colimit. Using colimits, the procedure we have just described can be generalized to merge more than two sets. With the generalized integration procedure, any collection of sets $A_i$, $i = 1..m$, together with any collection of mappings between them $f_j$, $j = 1..k$, can be merged (integrated) into the least upper bound set $S$ with canonic embeddings of the component $\iota: A_i \rightarrow S$. We will sometimes call a general configuration $(A, f)$ with $A = (A_i, i = 1..m)$, $f = (f_j, j = 1..k)$ a (generalized) span in the category of sets and set mappings.

### 2.1.4 Merging graphs

The set merge process we just presented can be extended to merge graphs. Consider a generalized span $R = (G, f)$ over graphs and graph mappings, i.e., a configuration of graphs $G = (G_i, i = 1..m)$ and mappings between them $f = (f_j, j = 1..k)$. Since graphs are defined as tuples of sets and functions, the span $R$ can in fact be separated into two similar spans of sets $R_N = (G_{iN}, f_{jN})$, and $R_E = (G_{iE}, f_{jE})$, $i = 1..m$, $j = 1..k$ for nodes and edges respectively, connected by the family of “vertical” (with respect to Fig. 2.2(a)) functions $so_i$, $ta_i$, $i = 1..m$. The cospan can be built as follows:
First, we apply the same set merge process to the spans of nodes and edges. The results are two cospans, i.e., the merged sets $S_N$ and $S_E$ with their respective inclusion mappings $\iota'_N: G_{iN} \rightarrow S_N$ and $\iota'_E: G_{iE} \rightarrow S_E$, $i = 1..m$. Next, the sets $S_N$ and $S_E$ need to be related by their so and ta functions. To construct the so function, for example, we construct a set mapping (or a function composition) $\iota'_N \circ \text{so}_i: G_{iE} \rightarrow G_{iN} \rightarrow S_N$ from each of the input graphs $G_i$. The set mappings from all the input graphs together with $S_N$ form a cospan over the configuration of edge sets $G_{iE}$. Because of the universal property of $S_E$, there is a unique function $!_{\text{so}}: S_E \rightarrow S_N$. Similarly, the unique function $!_{\text{ta}}: S_E \rightarrow S_N$ can be constructed for the ta function. In this way we come to a graph $S = (S_N, S_E, !_{\text{so}}, !_{\text{ta}})$, and it can be checked that, 1) set mapping pairs $\iota'_i = (\iota'_{iN}, \iota'_{iE})$, $i = 1..m$ form graph mappings and hence the tuple $(S, \iota'_i, i = 1..m)$ is a cospan over $G_i$, and 2) this cospan possesses the universal property with respect to other graphs, i.e., it is the colimit (merge) of the initial span of graphs in the universe of graphs and graph mappings.

### 2.2 Model-Driven Engineering

The research presented in this dissertation focuses on some central parts, i.e., model integration and transformation, of a software development methodology called Model-Driven Engineering (MDE). In MDE, models are created for each of the different stages of software development processes. Models can help improve the communication between software developers, architects, managers, domain experts and customers, and therefore reduce misunderstanding among them. Moreover, by using models, the software development process can sometimes be significantly simplified through, for example, the automatic generation of design models from requirement
models using model transformation or the generation of code from state machine models.

### 2.2.1 Model-Driven Architecture

MDE has been standardized by the Object Management Group (OMG). The standard is called Model-Driven Architecture (MDA) [Grob]. MDA has a list of related standards. Some of the standards, e.g., the Unified Modeling Language (UML), are the foundation of this research. We will give a short description of each standard relevant for this research.

**UML**

The Unified Modeling Language (UML) [Obj07] is a general purpose modeling language provided by OMG. It contains a suite of 13 types of diagrams for describing different artifacts of a system under development. The following three diagram types are relevant for this research: Class Diagrams, Object Diagrams and Sequence Diagrams.

![Figure 2.6: A sample Class Diagram.](image-url)
Figure 2.7: An Object Diagram showing an instance of the Class Diagram in Fig. 2.6.

Class Diagrams

A Class Diagram typically shows a collection of classes and the relationships between them. Classes can contain, e.g., attributes and methods. Relationships can be shown using, e.g., associations or specialization between classes. A sample Class Diagram is illustrated in Fig. 2.6. In the Class Diagram, three classes, User, HACSCController and HACSDatabase, are connected by two associations between them.

Object Diagrams

An Object Diagram shows a specific instance of a system described by a Class Diagram. It contains a set of Objects (instances of Classes), and the Links (instances of Associations) between the Objects. A sample Object Diagram is illustrated in Fig. 2.7, which is an instance of the Class Diagram in Fig. 2.6.

Sequence Diagrams

Scenarios can be used to model system behaviours. Each scenario is a single possible behaviour of the system, and shows the interactions between objects of the system. They are well known to help requirements engineers elicit functional requirements, as well as comprehend and validate requirements. Sequence Diagrams are the most common notation provided in UML to describe scenarios. A Sequence Diagram specifies...
interactions, e.g., a sequence of exchanged messages, between a set of objects indicated by lifelines. Although UML Sequence Diagrams support many advanced features (e.g., combined fragments for modeling negative, alternative or parallel, rather than sequential flow of events), in this dissertation we only consider the core constructs of Sequence Diagrams, i.e., lifelines, asynchronized messages, executions, and occurrences.

The graphical concrete syntax for Sequence Diagrams is given in the UML standard. A sample Sequence Diagram is given in Fig. 2.8. The vertical dashed lines in the Sequence Diagram are the lifelines. The signatures, i.e., the names and the associated classes of the objects participating in the Sequence Diagram are explicitly specified in the boxes on the top of the lifelines. The objects and classes specified in a Sequence Diagram may be explicitly modeled in separate Class Diagrams and Object Diagrams. For example, the objects and classes in Fig. 2.8 are modeled in Fig. 2.7 and Fig. 2.6. Arrows in the diagram are the messages exchanged between the lifelines. Each arrow is labeled with a message name. Moreover, each message has a sender and a receiver. The source of an arrow, i.e., the sending message occurrence of a message, occurs on the sender’s lifeline. The target of an arrow, i.e., the receiving message occurrence of a message, occurs on the receiver’s lifeline. A lifeline may also contain local executions. Each execution must be started by an occurrence (either a start execution occurrence or a receiving message occurrence). Similarly, every execution must be stopped by an occurrence (either an end execution occurrence or a sending message occurrence).

Formally, a Sequence Diagram is a tuple $SD = (L, M, E, O, snd, rcv, str, end, <)$ where:
- $L$ is a finite set of lifelines. The lifelines represent the individual objects of some classes participating in the Sequence Diagram.

- $M$ is a finite set of messages between lifelines.

- $E$ is a finite set of local executions within lifelines.

- $O = O_M \cup O_E$ is a finite set of occurrences. Each occurrence appears on a particular lifeline $l \in L$. Occurrences in UML 2.0 are further classified as message occurrences $O_M$ or execution occurrences $O_E$ in order to model the sending and receiving of messages or the start and end of executions.

- $\text{snd}, \text{rcv}: M \rightarrow O$ are functions that match each message with its sending and receiving message occurrences respectively.

- $\text{str}, \text{end}: E \rightarrow O$ are functions that match each local execution with its start and end execution occurrences respectively.
CHAPTER 2. BACKGROUND

- $\leq O \times O$ is a relation between occurrences that satisfies the following two requirements: 1) For every lifeline $l \in L$, $o_1 < o_2$ if both $o_1$ and $o_2$ are occurrences on $l$ and $o_1$ is above $o_2$. 2) For each pair of occurrences $s$ and $r$ in $O$, $s < r$ if $s$ is the sending message occurrence of a message $m \in M$, i.e., $\text{snd}(m) = s$, and $r$ is the receiving message occurrence of the same message $m$, i.e., $\text{rcv}(m) = r$.

An informal semantics of Sequence Diagrams is provided in the UML 2.0 standard. Each Sequence Diagram is described by two sets of traces, a set of valid traces and a set of invalid traces. Because only Sequence Diagrams containing negative combined fragments (which are not dealt with in this research) can produce invalid traces, we describe valid traces only. The set of valid traces of a Sequence Diagram can be constructed by computing the transitive closure $\leq^+$ on $\leq$, which is a partial order on the occurrences $O$, i.e., message or execution occurrences. Then, a valid trace is a sequence of all the occurrences, i.e., a total order of $O$, that does not violate $\leq^+$. For example the semantics of the Sequence Diagram in Fig. 2.8 is a single valid trace, $(\text{send} \text{"login"} < \text{receive} \text{"login"} < \text{end} \text{"login execution"} < \text{send} \text{"validate"} < \text{receive} \text{"validate"} < \text{end} \text{"validate execution"} < \text{send} \text{"validUser"} < \text{receive} \text{"validUser"} < \text{end} \text{"validUser execution"} < \text{send} \text{"systemMenu"} < \text{receive} \text{"systemMenu"} < \text{end} \text{"systemMenu execution"})$.

Meta-modeling

The Meta-Object Facility (MOF) [Groc] is the OMG’s standard to support meta-modeling. Meta-modeling is similar to modeling in the sense that a model is an abstraction of some real world entities and a meta-model is an abstraction of models. MOF consists of a four-layered architecture. The top layer M3, or meta-meta model,
is used to describe meta-models in the top-middle M2 layer. An example of a meta-model in the M2 layer is the meta-model for UML. Models, e.g., UML models, are at the bottom-middle M1 layer. They are described by their associated meta-models. The bottom layer is the M0-layer and contains the real world entities. A similar yet simpler meta-modeling approach is the Eclipse Modeling Framework (EMF) [Fun09a].

**XMI**

The XML Metadata Interchange (XMI) [Gro07] is the OMG’s standard for exchanging metadata information via the Extensible Markup Language (XML). The primary usage of XMI is the serialization of UML models, and the interchange of UML models among different UML-based modeling tools. For example, a UML model containing the three diagrams in Fig. 2.6, Fig. 2.7 and Fig. 2.8 could be exported by one UML modeling tool as a serialized XMI file, and later the same XMI file can be deserialized and imported by another UML modeling tool.

**OCL**

The Object Constraint Language (OCL) [Gro06] is an OMG standard for describing declarative rules to provide additional constraints on any MOF based models and meta-models, e.g., UML models.

### 2.2.2 Tool support

A variety of tools and frameworks have been implemented to support MDE. The following three frameworks and tools have been used in this research.
CHAPTER 2. BACKGROUND

GMF

The Eclipse Graphical Modeling Framework (GMF) [Ecl07], which is an open framework based on EMF and Graphical Editing Framework for developing new graphical editors, has been used to create a new graphical editor for SD-graphs. More details regarding the usage of GMF will be given in Chapter 5.

UML2

The Eclipse UML2 framework [Fun09b] is a part of the ongoing Eclipse Model Development Tools project. Based on EMF, UML2 provides the implementation of the UML 2.x meta-model as a suite of APIs for the Eclipse platform. The Eclipse UML2 framework has been used in this research to access, to serialize, and to de-serialize UML models.

IBM RSA

The commercial UML modeling tool IBM Rational Software Architect (RSA) has been utilized in this research to create new UML models and to visualize merged UML Sequence Diagrams.

2.3 Source transformation via TXL

TXL [Cor06] is a hybrid functional and rule-based programming language particularly designed to support software analysis and source transformation tasks. The overall input-output behavior of TXL transformation is illustrated in Fig. 2.9. Each TXL program contains two components. First, the syntactic structure of the artifact to be
CHAPTER 2. BACKGROUND

Figure 2.9: An overview of TXL processor

Figure 2.10: The grammar of an example TXL program.

transformed is described by a context-free, possibly ambiguous grammar in Extended Backus-Naur Form (EBNF). Second, a rooted set of structural transformation rules has to be given in the form of pattern-replacement pairs. In TXL, rules not only specify rewriting, but also provide the strategy for applying them. The explicitly scoped application of parameterized subrules is scheduled by an implicit top-down search strategy automatically inferred from the rules. The formal semantics and implementation of TXL are based on tree rewriting in which matching transformation rules are applied to the input until a fixed point is reached.

Similar to definitions of nonterminal symbols in EBNF, define statements (e.g.,
lines 31 to 36 in the sample grammar in Fig. 2.10) in TXL grammars give definitions for TXL nonterminals (e.g., \texttt{FamilyMModel\_member\_Atts}) with nonterminals referenced in square brackets (e.g., \texttt{[attvalue]}) and terminal symbols. Terminal symbols are either prefixed with a single quote (e.g., '\texttt{firstName}' in line 32 in Fig. 2.10) or represented directly (e.g., the equal symbol immediately follows '\texttt{firstName}' in line 32). Previously defined nonterminals can also be overridden by using \texttt{redefine} statements. Moreover, modularity can be achieved by using \texttt{include} statements (e.g., the inclusion of the previously defined XML grammar in line 1 in Fig. 2.10).

Transformation rules are defined by \texttt{rule} (e.g., lines 10 to 22 in Fig. 2.11) and \texttt{function} (e.g., lines 2 to 9 in Fig. 2.11) statements. The only difference between rules and functions is that a rule searches its pattern on the entire tree it is applied to and replaces every match with its replacement, whereas a function only replaces the first match. Inside a rule, the pattern, e.g., lines 3 and 4 (or replacement, e.g., lines 7 and 8) is defined by a sequence of terminals and variables in the \texttt{replace} (or \texttt{by}) statement. A rule’s pattern and replacement can be further restricted by deconstructors (\texttt{deconstruct} statements, e.g., lines 13 and 14), constructors (\texttt{construct} statements, e.g., lines 18 and 19) and conditions (\texttt{where} statements, e.g., lines 16 and 17). Local variables (either explicitly introduced by \texttt{construct} statements, by patterns, or by formal parameters of rules), or global variables (introduced by \texttt{import} and \texttt{export} statements) (e.g., lines 5, 6 and 15) can also be used to construct rules.

In TXL, rules are explicitly scheduled by the rule calling mechanism. A calling rule directly invokes a sequence of (possibly parameterized) subrules with a given scope, i.e., the input tree of the called subrules. The scope of the applied subrules can be effectively limited by restricting the input tree to particular trees. For instance, it
is possible to identify a subtree that matches some predefined pattern by the calling rule, and then the called subrules are applied to the matched subtree rather than the entire input tree of the calling rule. To supplement the default rule application strategy (rules to be applied recursively in order to find every match, and functions to be applied exactly once for the first match), TXL also supports the so-called one-pass rules. For this kind of rule (specified by a $ followed by the keyword replace), it is applied to each matching item exactly once. The combination of all these mechanisms allows very precise control over the execution of the TXL program and thus presents the experienced TXL user with a very effective means to improve performance and achieve rapid translation even of very large inputs.

For example, a TXL program is shown in Fig. 2.10 and Fig. 2.11. The grammar part of the TXL program, which parses XMI files conforming to a specific schema, is given in Fig. 2.10. When the main function in Fig. 2.11 is invoked, it will call the changeLastName rule with the whole input program p as the scope. Then, the called changeLastName rule will replace a family’s last name to “Simpson” as long as the last name is not equal to “Simpson”.

```txl
10  rule changeLastName
11    replace $ [FamilyMMModel_Family_Atts]
12      att [FamilyMMModel_Family_Atts]
13    deconstruct att
14      lastName = oldLastName [stringlit]
15    import newLastName[stringlit]
16    where
17      oldLastName [~= newLastName]
18    construct newLastName [attvalue]
19      newLastName
20    by
21      lastName = newLastName
22  end rule
```
Chapter 3

Sequence Diagrams integration via SD-graphs

3.1 Sequence Diagrams as SD-graphs

As described in Section 2.2.1, a scenario, e.g., a Sequence Diagram, is a record of possible message exchanges between communicating objects. Figure 3.1 presents an example: a Sale scenario specified by a UML Sequence Diagram.

Sequence Diagrams have many advantages, but they are not directly amenable to formal manipulations. Since Sequence Diagrams (and an overwhelming majority of other modeling languages) are diagrammatic, a formalization based on graph-based structures seems to be advantageous. Indeed, graph-based formalisms possess the following desirable properties: they (i) provide a foundation for a large class of software specifications, e.g., Sequence Diagrams and Flow Charts, (ii) have solid theoretical foundations, e.g., category theory and graph grammars and graph rewriting, (iii) have
tool support. Thus, graph-based structures are amenable to effective algebraic manipulations. In this section, we will first present an informal description of a formalization for the core subset of Sequence Diagrams followed by a more formal description.

Any Sequence Diagram, for instance, the Sale Sequence Diagram in Fig. 3.1, explicitly specifies a particular behaviour of a system. Moreover, a Sequence Diagram has an (implicit) structural base: a set of interacting objects and the types of messages they can exchange. For example, it is reasonable to consider the two messages initialOffer and counterOffer as two different occurrences of the same message type offer between a Seller and a Buyer.

An formalization makes both the behaviour and the structural base explicit by representing Sequence Diagrams as SD-graphs, that is, a chain of two type mappings between three graphs

\[ G_0 \xrightarrow{\tau_1} G_1 \xrightarrow{\tau_2} G_2 \]

as shown in Fig. 3.2. Graph \( G_0 \), or class graph, is shown in the bottom cell of Fig. 3.2 and is very similar to a UML Class Diagram, except that the edges represent dynamic
rather than static associations [Ste02]. We therefore interpret them as message channels, e.g., the SB message channel between Seller and Buyer. A root node called Class is contained in every class graph. A special message channel Self is represented as a self-association on Class to allow messages being sent and received by the same object. For concurrent systems, an object of a class may or may not own a thread of control. In other words, an object can be specified as either an active object having its own thread of control or a passive object without its own thread of control [RJB04].

To distinguish these two different types of object, we introduce two sub-classes of the root Class, ActiveCls and PassiveCls. Inheritance is also used in our formalization to support sub-classing and association inheritance. For example, by introducing ActiveCls as a sub-class of Class, the message channel Self is inherited by ActiveCls.
Graph $G_1$ in the middle cell of Fig. 3.2 is called object graph. An object graph contains objects participating in interactions as its nodes, and dynamic links (instances of dynamic associations) or message types between objects as its edges\textsuperscript{1}. Note that nodes and edges of this graph are typed (or labeled) by classes and message channels of Graph $G_0$ by mapping $\tau_1$. We assume that an object of $ActiveCls$ can be in one of two states: “executing” or “blocked and waiting for a response”. To capture this, we introduce two special message types $exec$ and $wait$ as instances of self-association $Self$. Similarly, $exec$ and $rest$ (not used in Fig. 3.2) are introduced to capture the states of “executing” and “ready and really doing nothing” of an object of $PassiveCls$. A derived message type, $\backslash live$, is also shown in Fig. 3.2. A derived element is one that can be computed from other elements as a result of some algebraic operation. To be precise, we define $\backslash live = (exec + wait)^*$, i.e., any sequential composition of $exec$ and $wait$ arrows results in $\backslash live$. In our formalization, although derived elements contain no new semantic information, they either facilitate the understanding of scenarios, or are even required for proper scenario integration. We borrow from UML and prefix the names of derived elements by “$\backslash$”.

Graph $G_2$, called sequence graph, is a partial order of events and messages typed over Graph $G_1$ by mapping $\tau_2$, where nodes are event occurrences labeled by objects (to which these events happened) and arrows are message occurrences labeled by message types. The top cell of Fig. 3.2 shows the corresponding sequence graph of the scenario in Fig. 3.1. Labels of all event occurrences besides $S0$ and $B0$ are omitted. If an arrow in $G_2$ is labeled by $exec$, $wait$ or $rest$, then it will be attached to the lifeline of a single object (the one to which the self-association in $G_0$ is attached).

\textsuperscript{1}More accurately, $s$ and $b$ are roles (formal parameters in the interaction) that real objects could play. To simplify wording, we will call them objects when it will not lead to confusion.
For instance, all vertical arrows in our example sequence graph are labeled with either exec or wait. Intuitively, labeling an arrow by exec means in the time period between the source and target event occurrences the object is executing some procedure. In contrast, labeling an arrow with wait or rest means the object is blocked and waiting for a response or the object is ready respectively. Exec represents the execution of the procedure triggered by the message coming into the source event. To help us distinguish between arrows typed with wait (rest) or exec, we use the following concrete syntax for them: the former are shown with dotted arrows (e.g., between B0 and B1) and the latter with bold (e.g., between S0 and S1) arrows.

Given the definitions of graphs and graph morphisms in Definition 1 on page 9, the following definitions present our descriptions in a formal way.

**Definition 2** (Typed graphs). A graph $G$ is typed over $G_{\tau}$ if there is a graph homomorphism $\tau: G \rightarrow G_{\tau}$. Thus, the graph $G$, i.e., the source of the homomorphism, is typed over its type graph or base $G_{\tau}$, i.e., the target of the homomorphism. The homomorphism itself is called typing or labeling. Given two typed graphs over the same base, $\tau: G \rightarrow G_{\tau}$ and $\tau': G' \rightarrow G_{\tau}$, a morphism between them is a graph morphism $h: G \rightarrow G'$ that commutes with typing: $\tau'(h(elm)) = \tau(elm)$ for any element $elm$ in graph $G$.

**Definition 3** (SD-graphs - Class, Object and Sequence graphs). A SD-graph is a three-layer typed graph consisting:

1. The bottom layer called class graph $G_0$, whose nodes are classes and whose edges are dynamic associations or message channels.

2. The middle layer called object graph $G_1$. Graph $G_1$ is a typed graph with typing $\tau_1: G_1 \rightarrow G_0$, whose nodes are called objects or instances and whose edges are
dynamic links or message types. *The type base of an object graph is the class graph* \( G_0 \).

3. *The top layer called sequence graph* \( G_2 \), *which is a graph typed over the object graph* \( G_1 \) with \( \tau_2 : G_2 \rightarrow G_1 \). *Nodes and arrows of* \( G_2 \) *are called, respectively, event and message occurrences.*

Thus, a SD-graph is a three-element chain of graph mappings \( G_0 \xleftarrow{\tau_1} G_1 \xleftarrow{\tau_2} G_2 \).

To summarize, we formalize Sequence Diagrams as a special form of typed graphs called SD-graphs, i.e., three layers of directed, labeled graphs containing dynamic and static information, where lower layers serve as types for the higher layers. Such layered, typed structures are useful to support, e.g., consistency checking. Moreover, unlike formalizations of Sequence Diagrams based on, e.g., partially ordered multisets, our formalization retains the graphical nature of Sequence Diagrams. More precisely, there is an obvious similarity between the formalization and what is being formalized, which increases learnability.

### 3.2 Transformations between Sequence Diagrams and SD-graphs

Given the definition of Sequence Diagrams in Section 2.2.1 (i.e., a Sequence Diagram is a tuple \( SD = (L, M, E, O, snd, rcv, str, end, \prec) \)), and the definition of SD-graphs in the previous section, the SD-graph representation of a Sequence Diagram can be constructed by a transformation process.

Assume a Sequence Diagram \( SD \) is contained in a UML model. The model also
contains a list of classes and objects. Moreover, we also assume that each object participating in SD is uniquely represented by a single lifeline. Then the following process can be used to generate a SD-graph from a Sequence Diagram:

1. Six lists, \( l_{MO} \) for message occurrences, \( l_{EO} \) for event occurrences, \( l_{MT} \) for message types, \( l_{OB} \) for objects, \( l_{MC} \) for message channels, and \( l_{CL} \) for classes in the target SD-graph are first constructed to hold temporary results during the transformation.

2. For each lifeline \( l_{SD} \in L \) in SD, a new class \( cl_{SG} \) and a new object \( ob_{SG} \) are constructed.

   - Depending on whether the same class is already in the list \( l_{CL} \), the class \( cl_{SG} \) may be added to the list or discarded. If the class \( cl_{SG} \) has been added to the list,
     - a new self message channel \( self_{SG} \) will be constructed with the class \( cl_{SG} \) as the source and the target of the self channel,
     - the in and out edges of the class \( cl_{SG} \) are updated to include the newly constructed self message channel \( self_{SG} \),
     - the message channel \( self_{SG} \) is then added to the list \( l_{MC} \).

   - Depending on whether the same object is already in the list \( l_{OB} \), the object \( ob_{SG} \) may be added to the list or discarded. If the object \( ob_{SG} \) has been added to the list,
     - the special self exec message type \( exec_{SG} \) will be constructed with the object \( ob_{SG} \) as the source and the target,
the in and out edges of the object $ob_{SG}$ are updated to include the newly constructed self message type $exec_{SG}$,

- the message type $exec_{SG}$ is then added to the list $l_{MT}$,

- the type of the object $ob_{SG}$, i.e., its class label, is set to the class $cl_{SG}$ (or, the equivalent class in the list $l_{CL}$ in case $cl_{SG}$ has been discarded),

- the type of the message type $exec_{SG}$, i.e., its message channel label, is set to the self message channel $self_{SG}$ of the class $cl_{SG}$ (or, the equivalent class in the list $l_{CL}$ in case $cl_{SG}$ has been discarded).

3. For each occurrence $o_{SD} \in O$ in $SD$, a new event occurrence $e_{SG}$ is constructed. Because the occurrence $o_{SD}$ appears on a lifeline in $SD$, we can identify the object associated with the lifeline in $SD$ and the object in the SD-graph $ob_{SG}$ located in the object list $l_{OB}$. The type of the event occurrence $e_{SG}$ is then set to $ob_{SG}$. Finally, the event occurrence $e_{SG}$ is added to $l_{EO}$.

4. For each message $m_{SD} \in M$ in $SD$, a new message occurrence $mo_{SG}$ is constructed. The message occurrence $mo_{SG}$ is then added to $l_{MO}$.

5. For each local execution $e_{SD} \in E$ in $SD$, a new message occurrence $me_{SG}$ is constructed. The message occurrence $me_{SG}$ is then added to $l_{MO}$.

6. From the messages in $SD$, we update the lists of message occurrences, message types and message channels. More precisely, for each message $m_{SD} \in M$ in $SD$, its corresponding message occurrence $mo_{SG}$ can be located in the list of message occurrences $l_{MO}$. The sending and receiving message occurrences $s_{SD}$ and $r_{SD}$ of the message $m_{SD}$ can be returned by the $snd$ and $rcv$ functions. Their corresponding event occurrences $s_{SG}$ and $r_{SG}$ can be located in the list
of event occurrences \( l_{EO} \). The sending object \( sob_{SG} \) is the type of \( s_{SG} \). In turn, the sending class \( scl_{SG} \) is the type of \( sob_{SG} \). Similarly, we have the receiving object \( rob_{SG} \) and class \( rcl_{SG} \) from \( r_{SG} \).

- The source and the target of the message occurrence \( mo_{SG} \) are updated with the event occurrences \( s_{SG} \) and \( r_{SG} \). The in and out edges of \( s_{SG} \) and \( r_{SG} \) are updated to include \( mo_{SG} \).
- A new message channel \( mc_{SG} \) from \( scl_{SG} \) to \( rcl_{SG} \) is constructed. Depending on whether the same channel (i.e., a message channel from \( scl_{SG} \) to \( rcl_{SG} \)) is already in the channel list \( l_{MC} \), the new channel \( mc_{SG} \) can be added to the list or discarded. If the channel is added to the list,
  - the source and target of the message channel \( mc_{SG} \) are updated with the classes \( scl_{SG} \) and \( rcl_{SG} \),
  - the in and out edges of the classes \( scl_{SG} \) and \( rcl_{SG} \) are updated to include the channel \( mc_{SG} \).
- A new message type \( mt_{SG} \) from \( sob_{SG} \) to \( rob_{SG} \) is constructed. Depending on whether the same type (i.e., a message type from \( sob_{SG} \) to \( rob_{SG} \) with the name of the message) is already in the message type list \( l_{MT} \), the new type \( mt_{SG} \) is added to the list or discarded. If the type is added to the list,
  - the source and target of the message type \( mt_{SG} \) are updated with the objects \( sob_{SG} \) and \( rob_{SG} \),
  - the in and out edges of the objects \( sob_{SG} \) and \( rob_{SG} \) are updated to include the type \( mt_{SG} \).
CHAPTER 3. SEQUENCE DIAGRAMS INTEGRATION VIA SD-GRAPHS

- the type of the message type \( mt_{SG} \), i.e., the message channel label of \( mt_{SG} \), is set to the channel \( mc_{SG} \) (or, the equivalent channel in the list \( l_{MC} \) in case \( mc_{SG} \) has been discarded).

- The type of the message occurrence \( mo_{SG} \) is then set to \( mt_{SG} \) (or, the equivalent message type in the list \( l_{MT} \) in case \( mt_{SG} \) has been discarded).

7. From the local executions in \( SD \), we update the lists of message occurrences and event occurrences. For each local execution \( e_{SD} \in E \) in \( SD \), we have its corresponding message occurrence \( me_{SG} \) in the list \( l_{MO} \). The start and end execution occurrences \( s_{SD} \) and \( e_{SD} \) of the execution \( e_{SD} \) can be returned by the \( str \) and \( end \) functions. Their corresponding event occurrences \( s_{SG} \) and \( e_{SG} \) can be located in the list of event occurrences \( l_{EO} \).

- The source and the target of the message occurrence \( me_{SG} \) are updated with the event occurrences \( s_{SG} \) and \( e_{SG} \). The in and out edges of \( s_{SG} \) and \( e_{SG} \) are updated to include \( me_{SG} \).

- Because the type of both the event occurrences \( s_{SG} \) and \( e_{SG} \) is the same object \( ob_{SG} \) in the list \( l_{OB} \), the type of the message occurrence \( me_{SG} \), i.e., the message type of \( me_{SG} \), is set to the special exec message type \( exec_{SG} \) of \( ob_{SG} \).

8. We categorize classes into active classes and passive classes. For each class \( cl_{SG} \) in the list \( l_{CL} \), if there is an object \( ob_{SG} \) in \( l_{OB} \) that has \( cl_{SG} \) as its type and also contains at least one outgoing edge, the class \( cl_{SG} \) is set as an active class. Otherwise, the class \( cl_{SG} \) is set as a passive class. For each object \( ob_{SG} \) with \( cl_{SG} \) as its class label, the special self message type wait/rest \( wr_{SD} \) is constructed.
Depending on whether $cl_{SG}$ is an active or passive class, the $wr_{SD}$ could be the wait message type or the rest message type for the object respectively. As usual, the source and the target of $wr_{SD}$ is updated with the object. The in and out edges of the object are updated to include $wr_{SD}$. Before the message type $wr_{SD}$ is added to the list $l_{MT}$, its type (i.e., its message channel) is set to the class $cl_{SG}$'s self message channel.

9. We make sure the sequence graph layer of a SD-graph is connected properly. More precisely, for each pair of occurrences $o1_{SD}$ and $o2_{SD}$ on the same lifeline in $SD$, if $(o1_{SD}, o2_{SD}) \prec \prec$ and there is no message $m_{SD} \epsilon M$ such that $snd(m_{SD}) = o1_{SD} \land rcv(m_{SD}) = o2_{SD}$ and there is no execution $e_{SD} \epsilon E$ such that $str(e_{SD}) = o1_{SD} \land end(e_{SD}) = o2_{SD}$, then we construct a new message occurrence $mo_{SG}$. The source and the target of $mo_{SG}$ are set to $o1_{SD}$ and $o2_{SD}$'s corresponding event occurrences $eo1_{SG}$ and $eo2_{SG}$ in the list $l_{EO}$. The in and out edges of $eo1_{SG}$ and $eo2_{SG}$ are updated to include $mo_{SG}$. Before $mo_{SG}$ is added to the message occurrence list $l_{MO}$, the type of $mo_{SG}$ is set to $eo1_{SG}$ (or, $eo2_{SG}$)'s special wait or rest message type $wr_{SG}$.

10. Finally, according to the concrete syntax (e.g., the meta-model of SD-graphs shown in Fig. 5.2), the SD-graph can be populated from the six lists.

The process of transforming SD-graphs back to Sequence Diagrams is defined similarly, except that elements in Sequence Diagrams are generated from elements in SD-graphs.
3.3 Merging SD-graphs

The merge of sets and graphs based on the category theory colimit has been explained in Section 2.1.3. To merge typed graphs or SD-graphs, we also need to consider the “horizontal” labeling functions in Fig. 2.2(b). Thus, given a generalized span of typed graphs, we can merge them by first following the graph merge procedure. Next, similar to the computation of the “vertical” functions so and ta described in Section 2.1.3, we use the same universal property of the cospan to compute $!_{h_N}: S_N \rightarrow S_N$ and $!_{h_E}: S_E \rightarrow S_E$.

![Diagrams](image)

Figure 3.3: Category with partial arrows in (a), and category with total arrows in (b).

The SD-graphs merge procedure described above is rooted in the algorithm for finding the colimit of sets with total functions. However, arrows of a category, e.g., the category of SD-graphs, can be partial, i.e., an arrow may not connect every element of the source object to an element in the target object. For example, the arrow from the Retailer to WholeSale or to RetailSale in Fig 3.7 is a partial arrow. When arrows of a
category are partial, we consider the category as an abbreviation of another category with total arrows. For example, in Fig. 3.3 (a) we have a category with partial arrows. For each partial arrow $f_i : rel \rightarrow in_i$ where $i = 1..n$, $f_i$ is replaced by a new head $rel_i$ and two total arrows, $f'_i : rel_i \rightarrow rel$ and $f''_i : rel_i \rightarrow in_i$, where $rel_i$ is the subset of $rel$ where $f_i(rel)$ is defined. The replacement of partial arrows in Fig. 3.3 (a) with new heads and total arrows is illustrated in Fig. 3.3 (b). The colimit is then calculated by the same merge procedure on the new category with total arrows.

### 3.4 Scenario integration: An example

Scenario integration is required for several software development methodologies. In this section we consider an example reminiscent of the role composition process required by OOram [Ree95] or ROP [Van97], and show how the merge machinery developed above can work. Suppose we want to build a model (scenario) which integrates two copies of the Sale scenario shown in Fig. 3.1 on page 27 into a brokered sale model: a BrokeredSale ($BS$) is a composition of two Sales, called the WholeSale ($WS$) and the RetailSale ($RS$). Furthermore, we want this integrated scenario to satisfy the following requirements:

(i) The Retailer is the Buyer in WholeSale, and it is the Seller in RetailSale.

(ii) The Retailer’s role requires two activities in addition to those of the sale transactions: a Retailer must do some thinking and some banking.

In order to integrate the two given scenarios we need to merge the corresponding SD-graphs, i.e., the class, object and sequence graphs.
Given two SD-graphs to be merged, we need to specify the overlapping elements (i.e., nodes and edges). That is, we need to establish a correspondence between these graphs of those elements which are supposed to be identified. To do this, a first approach, found in tools such as RSA [Let], is to use heuristics like identifying elements by name. Such approach however, is not general enough. In particular it does not deal with requirements such as (i). A more general approach is to create a third graph representing those common elements, and possibly containing new elements as we explained in the introduction chapter. We then specify how this third graph establishes the correspondence by defining a pair of maps which map elements of the head to elements of each of the original graphs. According to the terminology on page 11, the third graph is the head and the maps are the arms. The head graph and the associated arms are the span of the two original graphs. From the span, the merged graph will be generated.

Going back to the problem, given the two copies of the SD-graphs of the sale scenario shown in Fig. 3.2\(^2\), we first look at how class and object graphs are merged, and then we consider sequence graphs. Please note that, although the merge of the sale Sequence Diagrams is explained as three separated merge operations, the actual merge is still performed as a single operation as explained in Section 3.3. The explanation proceeds in three separate steps to simplify the presentation, because it is impossible to lay out the generalized span of this example in a readable way.

**Integration of class graphs**

Fig. 3.4 shows the span, i.e., the class graphs to be merged (\(WS_{CG}\) for the WholeSale and \(RS_{CG}\) for the RetailSale), the head Retailer graph \(R_{CG}\) at the bottom, and

\(^2\)To distinguish the names between the two copies, we primed the names of one of the copies.
CHAPHER 3. SEQUENCE DIAGRAMS INTEGRATION VIA SD-GRAPHS

![Diagram of sequence diagrams integration via SD-graphs]

Figure 3.4: The span of the \( WSCG \) and \( RSCG \) class graphs and the merged graph on top.

two arms \( rw \) and \( rr \) between the class graphs. The calculated cospan, i.e., the merged class graph \( BS_{CG} \) satisfying the requirements (i) and the inclusion arrows \( rtb, rb \) and \( wb \), is also shown in Fig. 3.4.

To partially satisfy requirement (i), i.e., the retailer plays both the role of the buyer in WholeSale and the role of the seller in RetailSale, we know that the Buyer class in \( WSCG \) and the Seller' class in \( RSCG \) should be combined into a new class Retailer. Thus, we create the following span. We first create a new class graph \( RCG \) containing a new class Retailer with its self message channel \( Self'' \) as shown in Fig. 3.4. To specify the equivalences between the classes Buyer and Seller', and their associated Self message channels, the two arms are defined as:

\[
\begin{align*}
\text{rw} &: RCG \to WSCG = \{ \text{Retailer} \leftrightarrow \text{Buyer}, \text{Retailer}.Self'' \leftrightarrow \text{Buyer}.Self \} \\
\text{rr} &: RCG \to RSCG = \{ \text{Retailer} \leftrightarrow \text{Seller'}, \text{Retailer}.Self'' \leftrightarrow \text{Seller'}.Self' \}
\end{align*}
\]
Given the span, the cospan can be calculated by following the graph merge procedure described in Section 2.1.3. More precisely, the cospans of the nodes and edges are first calculated. The cospan of the nodes consists of the merged nodes $S_N = \{Retailer, Buyer', Seller\}$, and the three inclusion arrows:

- $wb_N: WSCG_N \rightarrow S_N = \{Seller \mapsto Seller, Buyer \mapsto Retailer\}$
- $rb_N: RSCG_N \rightarrow S_N = \{Seller' \mapsto Retailer, Buyer' \mapsto Buyer'\}$
- $rtb_N: RCG_N \rightarrow S_N = \{Retailer \mapsto Retailer\}$

The cospan of the edges consists of the merged edges $S_E = \{Self, Self', Self'', SB, SB'\}$, and the three inclusion arrows:

- $wb_E: WSCG_E \rightarrow S_E = \{Seller.Self \mapsto Self, Buyer.Self \mapsto Self'', SB \mapsto SB\}$
- $rb_E: RSCG_E \rightarrow S_E = \{Seller'.Self' \mapsto Self'', Buyer'.Self' \mapsto Self', SB' \mapsto SB'\}$
- $rtb_E: RCG_E \rightarrow S_E = \{Retailer.Self'' \mapsto Self''\}$

Next, we relate $S_N$ and $S_E$ by the $so$ and $ta$ functions. For example, the three $so$ functions of $WSCG$, $RSCG$ and $RCG$ are:

- $so_{WSCG}: WSCG_E \rightarrow WSCG_N = \{Seller.Self \mapsto Seller, Buyer.Self \mapsto Buyer, SB \mapsto Seller\}$
- $so_{RSCG}: RSCG_E \rightarrow RSCG_N = \{Seller'.Self' \mapsto Seller', Buyer'.Self' \mapsto Buyer', SB' \mapsto Seller'\}$
- $so_{RCG}: RCG_E \rightarrow RCG_N = \{Retailer.Self'' \mapsto Retailer\}$

Then, we compose the $so$ functions with the inclusion arrows from the cospan of the nodes. Thus, we have the following three functions,

- $wb_N \circ so_{WSCG}: WSCG_E \rightarrow S_N = \{Seller.Self \mapsto Seller, Buyer.Self \mapsto Retailer, SB \mapsto Seller\}$
- $rb_N \circ so_{RSCG}: RSCG_E \rightarrow S_N =$
\{\text{Seller}', \text{Self}' \mapsto \text{Retailer}, \text{Buyer}', \text{Self}' \mapsto \text{Buyer}', \text{SB}' \mapsto \text{Retailer}\}

\(rtb_N \circ so_{Rcg} : R_{CG_E} \to S_N = \)
\{\text{Retailer}', \text{Self}'' \mapsto \text{Retailer}\}

The three composite functions and \(S_N\) form another cospan over the edge span. As we explained in Section 2.1.3, because of the universal property of \(S_E\), we have the following unique function:
\(!_{so} : S_E \to S_N = \{\text{Self} \mapsto \text{Seller},
\text{Self}' \mapsto \text{Buyer}', \text{Self}'' \mapsto \text{Retailer}, \text{SB} \mapsto \text{Seller}, \text{SB}' \mapsto \text{Retailer}\}\)
to make \(wb_N \circ so_{WS_{CG}}\) and \(!_{so} \circ wb_E\), \(rb_N \circ so_{RS_{CG}}\) and \(!_{so} \circ rb_E\), and \(rtb_N \circ so_{Rcg}\) and \(!_{so} \circ rtb_E\) commute.

Similarly, we have another unique function:
\(!_{ta} : S_E \to S_N = \{\text{Self} \mapsto \text{Seller},
\text{Self}' \mapsto \text{Buyer}', \text{Self}'' \mapsto \text{Retailer}, \text{SB} \mapsto \text{Retailer}, \text{SB}' \mapsto \text{Buyer}'\}\).

Up to this point, we finished the merge of class graphs, i.e., we have computed the merged class graph \(BS_{CG} = (S_N, S_E, !_{so}, !_{ta})\), and the inclusion arrows \(wb = \{wb_N, wb_E\}\), \(rb = \{rb_N, rb_E\}\) and \(rtb = \{rtb_N, rtb_E\}\).

**Integration of object graphs** Fig. 3.5 shows the object graphs to be merged (\(WS_{OG}\) for the WholeSale and \(RS_{OG}\) for the RetailSale), the head being the Retailer graph \(R_{OG}\) at the bottom together with the arms.

The head object graph contains a new object \(r:\text{Retailer}\) with two new self links or message types *thinking* and *banking*. In the arm between \(R_{OG}\) and \(WS_{OG}\), there is an equivalence between \(r:\text{Retailer}\) in \(R_{OG}\) and \(b:\text{Buyer}\) in \(WS_{OG}\), representing the requirement that the retailer plays the role of buyer in WholeSale. Similarly, there is an equivalence between \(r:\text{Retailer}\) in \(R_{OG}\) and \(s:\text{Seller}\) in \(RS_{OG}\) in the arm between
Figure 3.5: The span of the $WS_{OG}$ and $RS_{OG}$ object graphs and the merged graph on top.

$R_{OG}$ and $RS_{OG}$, representing the requirement that the retailer plays the role of seller in RetailSale. Hence, requirements (i) and (ii) are captured.

The merged object graph is $BS_{OG}$. Fig. 3.5 also shows how the elements of $WS_{OG}$ and $RS_{OG}$ are mapped to the elements of the resulting graph $BS_{OG}$ by the inclusion arrows. The cospan, i.e., the object graph $BS_{OG}$ and the inclusion arrows, is obtained by the same graph merge algorithm we have just demonstrated in the merge of the class graphs.

In addition, because object graphs are typed over their associated class graphs, in this example, $WS_{OG}$ is typed over $WS_{CG}$, $RS_{OG}$ is typed over $RS_{CG}$, $R_{OG}$ is typed over $R_{CG}$, we also need to compute the typing function between $BS_{OG}$ and $BS_{CG}$, or in other words we need to set the two “horizontal” functions in Fig. 2.2 on page 9. Take the homomorphism between nodes as an example, all the merged nodes from the merged class graph $BS_{CG}$ and the merged object graph $BS_{OG}$ are
in a set $BS_N$. Moreover, there are the inclusion arrows that have been calculated during the merge, i.e., $wb_N: WSN \rightarrow BS_N$ between the nodes of the wholesale graphs and the nodes of the merged graphs, and similar $rb_N: RSN \rightarrow BS_N$ and $rtb_N: RN \rightarrow BS_N$. Given the graphs, we also have three homomorphism functions on nodes, $h_{WSN}: WSN \rightarrow WSN$, $h_{RSN}: RSN \rightarrow RSN$, and $h_{RN}: RN \rightarrow RN$. Using arrow composition, we obtain $wb_N \circ h_{WSN}: WSN \rightarrow BS_N$, $rb_N \circ h_{RSN}: RSN \rightarrow BS_N$, and $rtb_N \circ h_{RN}: RN \rightarrow BS_N$. The composite arrows along with $BS_N$ form another cospan over the span. Because of the universal property, we know that there is a unique arrow $h_{SN}: BS_N \rightarrow BS_N$ making $wb_N \circ h_{WSN}$ and $h_{SN} \circ wb_N$, $rb_N \circ h_{RSN}$ and $h_{SN} \circ rb_N$, and $rtb_N \circ h_{RN}$ and $h_{SN} \circ rtb_N$ commute. The configuration of nodes and arrows is shown in Fig. 3.6. By the same approach, we can set the homomorphism function on edges $h_{SE}: BS_E \rightarrow BS_E$. Together, $h_{SN}$ and $h_{SE}$ are the typing for $BS_{OG}$ over $BS_{CG}$.
**Integration of sequence graphs** Now suppose that we have some additional requirements for our BrokeredSale scenario:

(iii) The RetailSale follows the WholeSale.

(iv) After buying from the whole-seller, the Retailer does some thinking. After this, he/she begins the process of retail selling, which is followed by some banking.

(v) The Retailer pays the whole-seller after he/she receives payment from the retail Buyer.

These are behavioral requirements and so they are to be captured by the result of merging the sequence graphs for the WholeSale and RetailSale.

As with class and object graphs, to merge sequence graphs we need to specify the points of overlap, and we do this by providing a span, i.e., a head graph and the associated arms defining the correspondence.

Fig. 3.7 shows the sequence graphs to be merged with their span. The graph $WS_{SG}$ is the sequence graph for the WholeSale, $RS_{SG}$ is the sequence graph for the RetailSale, and $R_{SG}$ is the sequence graph showing the lifeline of the Retailer, and thus it is the head of the span. Each graph is typed over its corresponding object graph.

By explicitly defining the correspondence between the two sequence graphs as a span in terms of the Retailer’s lifeline, we capture requirements (iii) - (v). In this graph, we are able to define new elements which where not present in either of the sale scenarios, such as the thinking and banking arrows. We also introduce arrows buying and selling which, while not present in the sale scenarios as individual arrows, correspond to a composition of arrows in their respective graphs. For example, the
Figure 3.7: The span of the sequence graphs $WS_{SG}$ and $RS_{SG}$ to be merged.

Arrow $buying$ in $R_{SG}$ from R0 to R1 is mapped to the new derived arrow $/buying$ in $WS_{SG}$, which is the composition of the arrows from B0 to B3. Hence, the arrow $buying$ can be seen, from the point of view of the retailer, as an abstraction of a sequence of actions that occur in the whole sale. Furthermore, we map a composition of arrows ($/retail$) in $R_{SG}$ to a single arrow ($retail$) in $WS_{SG}$. Note that this composite arrow $/retail$ includes the $selling$ arrow (which itself is associated to the composite $/selling$ in $RS_{SG}$.) This allows us to ensure that the RetailSale occurs within the retail process and before the $Retailer$ pays the whole-seller, thus satisfying requirements (iii) - (v).

As with the class and object graphs, we build the merged sequence graph by computing the cospan over the span. This yields the sequence graph shown in Fig. 3.8(a). Note the derived (dashed-bold) arrows among the elements of this graph. When two arrows, where one is basic (in one graph) and the other is derived (in another graph), are glued together in the merge, the result is a derived arrow because it can be derived
Figure 3.8: Result of merge as a typed graph (a) and as Sequence Diagram (b) exactly in the same way as it is derived in its component graph. For example, the arrow retail was basic in WS$_{SG}$ but becomes derived in the merge after gluing it with the derived arrow /retail, because all the operands for its derivation are present in the merged graph. Derived arrows in the merged graph are useful for traceability, but apart from that they can be safely removed. We call this last step of the integration normalization.

Given the merged SD-graph, i.e., the merged class graph in Fig. 3.4, object graph in Fig. 3.5, sequence graph in Fig. 3.8(a) and the typing between them, the sequence diagram equivalent of the merged SD-graph is shown in Fig. 3.8(b).
3.5 Scenario integration: General process

The example we have just considered along with the machinery of merge suggest the following general process for scenario integration:

1. **Formalization**: We define a universe of typed graphs and specify the scenarios to be integrated (the *views*) as typed graphs in this universe.

2. **Specification of view correspondences**: We define the correspondence between views by providing *spans*, which consist of 1) *head graphs*, i.e., typed graphs which contain elements (nodes and edges) that represent the overlap of the views, and 2) *arms*, i.e., mappings specifying the roles that elements of a head graph play in each view. The head graphs can contain new information which was not present in the original views.

3. **Merge**: From the span of typed graphs, the merged typed graph is obtained by computing the colimit of the span.

4. **Normalization**: In the merged graph, we eliminate redundant arrows, i.e. arrows which can be derived from basic (non-composite) edges.

3.6 Advantages and limitations

After the description of our approach for scenario integration and the example of scenario integration in the previous sections, a list of advantages or properties is given now. Some of them are directly coming from the properties of the underlying theory foundation of our approach.

**Precise merge** As described in Section 2.1.3, a fundamental property of our approach is that the resulting graph contains exactly all of the information from the input graphs and the head graph: everything is represented because views
are mapped into the merge, and nothing extra is acquired owing to the universal property of the colimit. Nonetheless, the semantics of a resulting graph could be well restricted compared to the semantics of the set of the input graphs. For example, the set of valid traces of a merged Sequence Diagram could be a subset of all the valid traces of the original input Sequence Diagrams. Thus, we have to stress that the resulting graph is the precise merge only according to the given head and arms because different merge results could be computed from the same input graphs by giving different heads and arms.

**Traceability**  Given the inclusion arrows from the computed cospans and the arms from spans, we can easily trace an element in the merged graph back to its origin in the original graphs by following the arrows.

**Generalizability**  It is also worth noting that while the example described the integration of just two scenarios, our merge procedure is based on the colimit and therefore our integration process is applicable to any number of scenarios to be merged. Moreover, any diagrams, for example, Message Sequence Charts, Class Diagrams or Object Diagrams, that can be represented by typed graphs are amenable to our integration process.

Some of the advantages are provided by our SD-graph representation of Sequence Diagrams.

**Graphical nature of formalization**  SD-graphs retains the graphical nature of Sequence Diagrams. Thus, there is an obvious similarity between the formalization and what is being formalized, which increases learnability.
Consistency checking The explicit typing information maintained by the layered SD-graphs can greatly facilitate consistency checking. For example, a checking tool can automatically validate whether the resulting event occurrence merged from two event occurrences has a single object as its type by checking the typing arrow between the merged sequence graph and the merged object graph.

Also, as described in Chapter 5, our integration process helped structure the design and implementation a prototype Sequence Diagram integration tool.

Our integration approach also has some limitations. The most noticeable one is that a user has to manually discover mappings between input graphs and then explicitly specify the mappings in some internal representation such that an integration tool can process the mappings. To address this limitation, in Chapter 6 we will introduce high-level merge patterns to alleviate this burden, and will be explained in detail in Chapter 6. Other limitations, such as the limited Sequence Diagrams constructs supported by our approach, suggest interesting future work and will be discussed in Chapter 8.
Chapter 4

Model Transformation with TXL

In Chapter 3, a general technique to merge Sequence Diagrams is presented. To implement this technique, Sequence Diagrams contained in serialized UML models must be transformed to SD-graphs, then the SD-graphs are merged and transformed back to Sequence Diagrams. Thus, we need a form of model transformation to realize and implement the manipulation of requirements in the form of UML Sequence Diagrams.

One of the OMG’s Model-Driven Architecture (MDA) [Grob] main principles is that system development can be viewed as a process creating a sequence of models. A model can highlight a particular perspective, or view, of the system under development. Model transformation then converts one view to a different view. The target view may be on an equivalent level of abstraction (e.g., from one structural view to another), or it may be on a different level (e.g., from a platform-independent model (PIM) to a platform-specific model (PSM)). To support the principle, the OMG has issued a Request For Proposals (RFP) on a specification of Query/Views/Transformations (QVT). Either as explicit answers to the request or
motivated by practical needs, numerous approaches have been proposed in the literature recently. The final adapted version of QVT is released by the OMG as a standard [Gro03]. In [CH03], Czarnecki and Helsen have presented a set of model transformation comparison criteria, and have provided a survey and categorization of several approaches. In [MCG05], Mens et al. have provided a taxonomy of model transformation.

A large body of work and tools for model transformation already exists [Groa, Tra06, LS05, KPP06a, GRe, Wil03, ATo, VIA]. However, we decided to employ the source transformation system TXL [Cor06] for this purpose. Given that TXL was not originally designed for model transformation, this may appear as a surprising choice. Our decision was motivated by the following considerations: On the one hand, we felt that our considerable experience with TXL would outweigh the advantages that the use of other approaches might bring. On the other hand, we wanted to study the differences between source and model transformation more closely and see to how source transformation can be employed for model transformation. The central purpose of this chapter thus is not to suggest that TXL should be used for model transformation in general; rather, we want to study the exact circumstances and conditions under which TXL could be used for this task and present an approach to model transformation, which, under certain circumstances, would allow for existing expertise on TXL in particular and source code transformation in general to be leveraged. In short, a TXL-based model transformation tool would have the following characteristics:

- **Versatility:** In [PR03], an example of a transformation from a profile of UML models to Java code has already shown that TXL is capable of model-to-code (M2C) transformation. In this chapter, we will provide evidence that TXL is
capable of model-to-model (M2M) transformation as well. Thus, both M2M and M2C transformations required in the entire MDA “life cycle” could be supported by TXL.

- **Maturity and scalability:** In the last twenty years, TXL has been widely and successfully used in industry and academia for various software engineering, programming language processing and analysis tasks. Moreover, TXL has been proven to be able to handle very large input files with up to 100,000 source lines per input file. Considering the complexity and scale of modern systems, such maturity and scalability are definitely desired.

- **Efficiency:** As pointed out in [CH03], the performance of a transformation approach is influenced by various design and implementation strategies, such as *rule application scoping*, *rule application strategy*, and *rule scheduling*. To achieve efficient transformation, TXL allows very precise and flexible control over scoping, rule application, and scheduling strategies.

- **Formal semantics:** In [Mal93], a denotational semantics was given for the TXL programming language. By having a formal semantics, formal reasoning and verification of the correctness of TXL transformation becomes possible, e.g., by automated theorem proving.

- **Gradual learning curve:** In our experience, people with some knowledge in Extended Backus-Naur Form (EBNF), and functional and logic programming are able to start programming in TXL fairly easily. Thus, no extensive background in MDA, or Object Constraint Language (OCL) [Gro06] is required to develop TXL rules and grammars for model transformation.
In this chapter, we will first describe an approach suitable for using TXL as a model transformation tool. Then, we will highlight the advantages and disadvantages of the approach.

The rest of the chapter is organized as follows. In section 4.1, we first identify the connections and gaps between model transformation and source transformation, and then presents an approach suitable for turning TXL into a model transformation tool. One example of using TXL as a model transformation tool is given in Section 4.2. Section 4.3 discusses the advantages and limitations of the approach. We discuss the related work in Section 4.4.

### 4.1 Model transformation via TXL

A three-layer meta-modeling architecture of our transformation approach is shown in Fig. 4.1. It is influenced by the architectures used in MOF [Groc] and EMF [Fun09a]. At the top M3 layer is the meta-meta model, e.g., Ecore as shown in the diagram. It is used to build meta-models at the middle M2 layer, e.g., MM_s and MM_t. The models, e.g., M_s and M_t conforming to MM_s and MM_t respectively, are at the bottom M1 layer. A TXL model transformation, e.g., the (green)\(^1\) transformation link \(MT_{st}\) between \(M_s\) and \(M_t\) using the TXL program \(P_{st}\) inside the (green) dashed box, would take a source model and produce a target model.

However, there is a gap between source transformation and model transformation. The input and output of both transformations seem quite different. On the one hand, source transformation takes a piece of text conforming to a particular grammar as input, e.g., Java source code, and produces another piece of text conforming to another

\(^1\)Shown as green on a color display
grammar, e.g., C++ source code. On the other hand, the input and output of model transformation are models conforming to meta-models, i.e., presumably diagrams generated by modeling tools. Fortunately, this difference can be overcome because MOF or EMF models, and their conforming meta-models, can be easily serialized as XMI [Gro07] files. As a consequence, a (serialized) model transformation is nothing but a special case of source transformation that takes a source model in one XMI file and produces a target model in another XMI file.

Another factor complicating model transformation via TXL slightly is that the
grammars for both input and output artifacts of a TXL program must be combined into a single grammar, which appears to conflict with the fact that the source and target models may conform to different meta-models, and thus may require different TXL grammars. In [DCMS02], union and consume/emit grammars are introduced to address tasks that involve different grammars. When input and output artifacts’ grammars share a lot of similar concepts, the grammars can be described by a union grammar such that whenever the concepts are matched they are combined. In contrast, when grammars are very different, it is better to combine them by means of a consume/emit grammar where the combination of grammars is only happening at the top level or not at all.

4.1.1 Automatic grammar generation

As a special case of source transformation, grammars for input and output models are needed for each TXL-based (serialized) model transformation. Meta-models (e.g., represented as serialized Ecore meta-models or XMI schemas) are promising starting points, because both meta-models and grammars share a “conforms” relationship with the input/output models. Unfortunately, meta-models cannot simply be used as grammars directly. Developers of TXL model transformation will have to manually construct a union or a consume/emit grammar by consulting the meta-models for each model transformation. Since (serialized) models are XMI files, one minimal requirement for the resulting grammars is that the grammars must be able to parse XMI files. To achieve this, one can extend the XML grammar [ZC] with additional XMI elements. In addition to adding XMI elements, when constructing a TXL grammar from a serialized Ecore meta-model, the following Ecore components
will be transformed to TXL grammar nonterminals:

- **EPackage**: as the root element of the serialized Ecore meta-model, the `EPackage` will be mapped to a top-level nonterminal that contains a list of all the root elements of the models.

- **EClass**: as an inner element of the meta-model’s root element, i.e., `EPackage`, each `EClass` will be mapped to a nonterminal that specifies a root element of the models. Such nonterminal consists of terminals and nonterminals of the root element’s inner elements and attributes.

- **EReference** and **EAttribute**: as an inner element of an `EClass`, depending on the lower bound and upper bound values, an `EReference` or `EAttribute` will be mapped to nonterminal(s) that specify inner elements and/or attributes of a root element of the models.

Experienced source code transformation programmers know that the design of a grammar suitable for the transformation task is an important step towards an efficient transformation. Although the manual grammar construction process is relatively simple, it can quickly become tedious and error-prone when creating TXL grammars from complex meta-models, e.g., the meta-model of UML. In contrast, a carefully designed grammar generator can mimic the process of a manual grammar creation process and can generate the same efficient grammars from meta-models. To relieve programmers from this unnecessary burden, we have implemented two automatic TXL grammar generators. One generates grammars from serialized Ecore meta-models (Ecore2Grammar). The Ecore2Grammar generator, shown as $P_{gen}$ in Fig. 4.1, was implemented as a TXL transformation. To build the Ecore2Grammar
generator, first we created a consume/emit grammar, which combines the grammar of Ecore and the grammar of the TXL language itself at the top level. Then, we created the grammar generation rules, which basically map those Ecore components to TXL nonterminals as described above. Fig. 2.10 shows an example of such automatically generated grammar from the serialized Ecore meta-model shown in Fig. 4.2. TXL grammars generated by Ecore2Grammar ensure model instances are correctly typed according to their meta-models. Constrains, e.g., OCL constraints contained in Ecore meta-models, provide further restrictions on instance models. Currently, the Ecore2Grammar generator does not support OCL constrains. The other generator, XMI2Grammar, is very similar to Ecore2Grammar, except it generates grammars from XMI schemas.

4.1.2 Transformation process

We now have all the ingredients we need to let TXL work as a model transformation tool. To develop a model transformation with TXL, we can then follow the general process illustrated in Fig. 4.1:

1. Gathering the meta-models: In this step, the meta-models $MM_s$ and $MM_t$ of both input and expected output models are gathered.

2. Generating grammars: In this step, the grammars of the input and output models, $G_s$ and $G_t$ are generated by taking the (red) transformations $MMG_s$ and $MMG_t$ with $MM_s$ and $MM_t$ as inputs, respectively. Both $MMG_s$ and $MMG_t$
transformations could use one of our implemented automatic grammar generators, e.g., the TXL program $P_{gen}$ in the (red) dashed-box.

3. Creating a transformation program: In this step, a TXL program, for instance $P_{st}$, is created by model transformation developers. The program contains a union or consume/emit grammar which is constructed by combining the grammars generated from the previous step. Transformation rules relating the source and target grammars are also created for the program.

4. Executing the transformation: In this final step, the TXL program $P_{st}$ created from the last step is executed as the transformation $MT_{st}$ with $M_s$ as the input model and will produce $M_t$ as the output target model.

### 4.2 Model transformation example

In the previous section, we have identified some of the differences between source transformation and model transformation, and then we have described an approach suitable for using TXL as a model transformation tool. In this section, we illustrate the approach on one example of model transformation via TXL by following the general process we described in the last section.

#### 4.2.1 Family2Persons

This example demonstrates the transformation between a Family model to a Persons model. A slightly different example has also been used by ATL [Groa].

1. **Gathering the meta-models:** The meta-model FamilyMModel is shown in Fig. 4.2. It consists of a Family class which has a lastName attribute, and a Member
class which has attributes firstName and relation. A Family contains at least one Member. A Member must belong to a single Family, and its relation to the Family could be, for example, “Father”, “Mother”, “Son”, “Daughter” and so on. The meta-model PersonsMModel is shown in Fig. 4.3. It consists of a Persons class, a Male class and a Female class. A group of Persons may contain Males and Females. Each Male or Female has a fullName attribute.

2. Generating grammars: From the serialized Ecore meta-models, i.e., FamilyMModel and PersonsMModel gathered from the last step, two TXL grammars family.grammar and persons.grammar are generated by using the Ecore2Grammar generator. In Sec. 2.3, family.grammar has already been shown in Fig. 2.10\(^3\). Fig. 4.4 shows the generated persons.grammar.

3. Creating a transformation program: Since family.grammar and persons.grammar

\(^3\)Actually, the generated grammar doesn’t contain the first three include statements and the first define statement. They are included to make the grammar self-contained for the TXL example shown in Sec. 2.3.

Figure 4.2: The FamilyMModel is shown as a class diagram and a serialized Ecore meta-model.
Figure 4.3: The PersonsMModel is shown as a class diagram and a serialized Ecore meta-model.

are quite different, a consume/emit grammar is more suitable than a union grammar. The creation of the consume/emit grammar includes importing the predefined grammars and the generated grammars, and creating nonterminals that combine family.grammar and persons.grammar at the top level. The consume/emit grammar is shown in Fig. 4.5.

To transform a Family model to a Persons model, the following rules are created and are shown in Fig. 4.5:

1. function main is the required TXL main rule which initiates the transformation by explicitly invoking other rules and functions.

2. function family2Persons replaces the input model containing a Family by an output model containing a newly constructed Persons object with a set of Males and Females.
3. function getLastName extracts the lastName from the Family and exports it as a global variable.

4. rule member2MaleOrFemale constructs either a Male or a Female for each Member of the Family. This rule further delegates its constructions to:

   (a) function getFirstName extracts the firstName from a Member and exports it as a global variable.

   (b) function isMale first checks whether the relation attribute of a Member is equal to “Father” or “Son”. If the relation is indeed a male relationship, the function will construct a Male with a fullName constructed by function concatLastName. The newly constructed Male is then added to the set of Males and Females.

   (c) function isFemale is similar to function isMale except that it creates a Female from a Member with either a “Mother” or “Daughter” relation.
**Figure 4.5:** The TXL program for family to persons model transformation.

(d) function `concatLastName` constructs a Male or Female’s `fullName` by concatenating a Member’s `firstName` with the Family’s `lastName`.

4. Executing transformation: When executing the model transformation specified by the TXL program shown in the previous step, a serialized Family model, e.g., the one shown in Fig. 4.6, is transformed to a serialized Persons model shown in Fig. 4.7.
4.3 Advantages and limitations

As a model transformation tool, TXL has some strong advantages. Some advantages, such as scalability and efficiency, are directly coming from the characteristics that TXL has. There are also other advantages we discovered during the use of TXL for the implementation of the transformations between Sequence Diagrams and SD-graphs.

- TXL allows fine-grained control over the target model construction. For instance, the details of the placement of a model element inside a model can easily be controlled via the scheduling of TXL rules, and the manipulation of global variables. In many situations the additional effort for writing these rules is outweighed by the benefits of such fine-grained control. This fine-grained control is especially helpful in our case as we were dealing with ordered sets of model elements. For instance, when manipulating Sequence Diagrams, it is of vital importance to maintain the order of event and message occurrences across transformations.
• In addition to the nonterminal types in the grammars that are generated from meta-models, TXL allows the rapid creation of new nonterminals. More precisely, new nonterminal types representing intermediate transformation results can be added to the grammars without the need to modify the original meta-models. Paired with global variables, intermediate results are processed in an efficient way yet still benefit from the strong typing offered by TXL.

TXL also has some limitations. Some of them are related to the design of TXL.

• TXL lacks a flexible rule/function return mechanism. As a functional language, only a single return, i.e., the replacement, is allowed. Moreover, the type of the return value is bound to the type of the pattern of the same rule/function, which could be fairly restrictive as a desired return value may have a different type than that of the pattern. To relax this restriction, a large number of TXL global variables creations and accesses were used in our transformations between Sequence Diagrams and SD-graphs. Although this approach fulfilled our needs for return values with different types, it did introduce more complexity into our implementations.

• TXL does not have “if clauses”. Consequently, whenever branching is needed in a rule, the rule has to be duplicated up to the branching point. The rule duplications not only increased the size, but also negatively affected testability and maintainability of our implementation.

Both limitations were also identified in [TC06], and suitable solutions were given to address them as part of new version of TXL - ETXL. The following limitation applies to serialized model transformation specifically.
As explained in Sec. 4.1, TXL currently is only applicable as a model transformation tool when both models and meta-models can be serialized. This could be inconvenient for users as explicit export operations may be needed to obtain serialized input models from modeling tools, and import operations may also be required before further manipulations on output models can be carried out.

The above limitation suggests some interesting future work. It might be resolved by properly integrating the upcoming Eclipse-based TXL IDE with Eclipse-based modeling tools such that explicit model import/export operations become unnecessary.

### 4.4 Related work

Similar to our TXL grammar generators, generation of grammars from meta-models is one of the main features of TCS (Textual Concrete Syntax) [JBK06]. Through transformations, associated meta-model elements, which are specified in a user defined TCS model, are translated to grammar elements. TCS supports a broader range of capabilities, e.g., text-to-model and model-to-text transformations. In contrast, our grammar generators are tailor-made for the automatic generation of TXL grammars from Ecore meta-models or XMI schemas without user intervention.

An earlier demonstration of the potential of using TXL for model transformation was discussed in [CS92]. Paige and Radjenovic have used TXL as model transformation tool in [PR03]. Clearly, our work is closely related to theirs. In [PR03], connecting MOF and TXL to allow TXL specifications to be automatically generated, has been identified as fruitful future work. Our work established exactly this link through the implementation of automatic grammar generators.
The process and techniques used in this chapter could also be applied to other source transformation tools. XSLT [W3C], for example, is a source transformation tool that specifically targets the transformation between XML documents. Since XMI documents are also XML documents, XSLT is also capable of model transformation. That capability has already been demonstrated by various XSLT-based model transformation approaches, e.g., UMT [GO05] and MTRANS [PBG01]. However, TXL-based approaches, such as ours, are applicable to a broader class of model transformations, because XSLT is not suitable for M2C transformation.

Since we applied TXL as a model transformation tool, this work is inevitably related to other model transformation tools. Roughly speaking, the model transformation tools can be grouped into two categories, relation-based and graph-transformation-based.

Our approach shares many similarities and is much closer to relation-based approaches. The relationship between source model and target model is expressed textually following the same “rule”, “pattern”, “replacement” and “condition” paradigm as in TXL. Some examples include ATL [Groa], MT [Tra06], Tefkat [LS05] and the Epsilon Transformation Language (ETL) [KPP06a]. Similar to our approach, ATL supports the transformation of serialized models. Rule scheduling in ATL can be achieved explicitly by invoking called rules inside the imperative code section of another rule. MT is another relation-based transformation language. It comes with predefined top-down rule scheduling. This simple scheduling makes understanding of transformation programs easier. Yet, we feel that it might impose an unnecessary restriction on programmers to author new transformations, and also could be an unnecessary burden for programmers to maintain existing transformation programs.
In contrast to our TXL-based approach, both ATL and MT approaches do not support flexible application scoping on declarative rules. When dealing with complex source models, the lack of a flexible rule application scoping could lead to less efficient model transformations. Moreover, both transformation execution algorithms of ATL and MT do not support ordering on target elements creation, or more precisely, no user control over the placement of target model elements. On the other hand, both approaches provide traceability information in a more mature way.

Since models, especially UML models, are visualized as graphs and diagrams, graph transformation, as used in, e.g., Progres [SWZ97] and AGG [AGG], is a natural fit in the context of model transformation. Several graph-transformation-based approaches have already been proposed to address the program of model transformation, for instance, GReAT [GRe], UMLX [Wil03], AToM³ [ATo], and VIATRA2 [VIA]. Unlike our TXL-based approach, the “rule”, “pattern”, “replacement” and “condition” paradigm can be expressed graphically or textually to manipulate graphs.

Most of the graph-transformation-based approaches provide a sophisticated rule scheduling algorithm. For instance, VIATRA2 uses Abstract State Machines to manipulate the rule scheduling. In GReAT, rule scheduling is supported by a separate control flow language. While the current language implementation of TXL only supports sequential execution, the rules in GReAT can be scheduled to run both sequentially and in parallel depending on how they are connected by the control flow language. Rule application scoping is supported by both GReAT and VIATRA2 through limiting the search space of a transformation rule to a subgraph of the source graph. Compared to our approach, rule application for both GReAT and VIATRA2 is limited to either all the valid matches or only the first valid match, but not recursively.
as our TXL-based approach. One distinct feature provided by GReAT is the ability to produce C++ code from transformation rules, which potentially could speed up the transformations by executing the C++ code directly. Also, transformation rules and control flows in GReAT could be specified graphically.
Chapter 5

A tool for Sequence Diagram integration

In this chapter, we present a prototype Sequence Diagram integration tool. The tool allows us to evaluate our general integration process further. To take full advantage of other existing Eclipse-based modeling tools (e.g., IBM RSA) which implement editing, visualization, import and export of Sequence Diagrams, the tool is implemented as an Eclipse extension. The tool also relies on the EMF, UML2 and GMF frameworks.

In this chapter we will first give an overview of the architecture of the prototype in Section 5.1. Then, some highlights of the implementation will be provided in Section 5.2. Finally, testing and evaluation of the prototype are discussed in Section 5.3.

5.1 Architecture

The mathematical theory underlying our approach helped us design the structure of our prototype. The prototype is clearly divided into several cohesive components,
each realizing a separate portion of the theory. The main components of the prototype and the data flow among them are depicted in Fig. 5.1. Our prototype assumes that the Sequence Diagrams and the head of the merge are created by the user with the help of existing modeling tools, such as RSA. After importing the Sequence Diagrams, the SD2SG Transformer will automatically convert them into SD-graphs. The user interactively creates mappings or correspondences between Sequence Diagrams with the help of the Mapping Editor. The correspondences between Sequence Diagrams are then translated to correspondences between SD-graphs. Strictly speaking, a mapping editor operating on SD-graphs directly rather than Sequence Diagrams would be more efficient, thus obviating the need for transforming mappings. Instead, the current implementation hides the SD-graphs representation from the users. As a consequence, it is likely more user-friendly as the users can work with familiar concepts, i.e., Sequence Diagrams. Next, the SG Merger takes SD-graphs and mappings between them, and produces a merged SD-graph. Finally, before the merged result can be visualized by a UML modeling tool, the SG2SD Transformer transforms merged SD-graphs back to Sequence Diagrams.

An interactive editor, SG Editor, is also provided. It allows for the creation,
CHAPTER 5. A TOOL FOR SEQUENCE DIAGRAM INTEGRATION

Modification, and display of SD-graphs. Moreover, it allows for the merge results to be validated through the examination of the SD-graphs. Finally, the Manager facilitates the overall Sequence Diagram integration process and controls the data flow between components.

5.2 Implementation

In this section we will briefly go over the implementation of the components introduced in the previous section.

The meta-model of SD-graphs is given in Fig. 5.2. Several components of our Sequence Diagram integration tool are primarily implemented based on this meta-model and will be discussed in detail in the rest of the section.

Because of the extensive model transformations and model manipulations involved in the integration process, tracing an element along the integration process becomes an important and useful function. In the last subsection, the traceability support by the prototype will be discussed.

5.2.1 Pre-processing Sequence Diagrams

In our current implementation, Sequence Diagrams, i.e. view diagrams and head diagrams, are created by existing modeling tools. However, some particular SD-graphs concepts, e.g., wait, rest and derived, are important for the proper Sequence Diagram integration, but are not part of the UML standard and therefore cannot be specified with existing modeling tools. For example, it is a common task to add execution occurrences on a lifeline of a head Sequence Diagram, such that mappings
Figure 5.2: The meta-model of SD-graphs.
from the execution occurrences to elements in view diagrams can be specified. Yet, because the UML standard requires that an execution occurrence must belong to an execution specification, extra execution specifications have to be provided in the head diagram as well. Rather than manually removing extra execution specifications and thus increasing the risk of creating invalid diagrams, a UML profile for SD-graphs has been implemented as part of the prototype. It can be optionally applied to Sequence Diagrams before the Sequence Diagrams are passed to our integration tool. The profile allows extra execution specifications be stereotyped as either a \texttt{<< wait >>} or a \texttt{<< rest >>}. In addition, a message, a message occurrence, an execution specification, or an execution occurrence can be stereotyped as \texttt{<< derived >>}.

5.2.2 SD2SG and SG2SD Transformers

A variety of existing transformation tools could have been used to develop these two components. We have chosen TXL [Cor06], and both SD2SG and SG2SD have been implemented in TXL by following the model transformation process described in Section 4.1. In this subsection we will only highlight the most interesting aspects of the transformations.

Some relatively large and complex meta-models are involved in the transformation process. Instead of creating a new Ecore meta-model for Sequence Diagrams, we used the existing serialized Ecore meta-model of UML provided in [Fun09b], which contains more than ten thousand lines of text. Then, the meta-model was transformed into a TXL grammar of UML by using the Ecore2Grammar generator discussed in Section 4.1. Similarly, the meta-model of SD-graphs shown in Fig. 5.2 has been transformed into a TXL grammar.
CHAPTER 5. A TOOL FOR SEQUENCE DIAGRAM INTEGRATION

Figure 5.3: Scoping and scheduling rules.

To transform between Sequence Diagrams and SD-graphs, over a hundred TXL rules and functions have been developed for both the SD2SG and SG2SD Transformers with combined more than four thousand lines of code. Particularly, the following three types of rules have been implemented.

1. Application scoping: Some of the rules were purely implemented for the purpose of limiting the application scope of other rules. For instance, when applied to an input UML model, the rules uml2SGs (line 1 to line 8) and collaboration2SG (line 13 to line 21) in Fig. 5.3 will limit the application scope of subsequently called rules from UML Models to UML Collaborations and then to UML Interactions.

2. Control and scheduling: Some of the rules were created for the purposes of control and scheduling of other rules. For example, the rule interaction2SG (line 25 to line 37) in Fig. 5.3 first declares and exports three global variables (line 29 to line 34) that are used to store intermediate transformation results, and
then explicitly invokes other defined rules in sequence (lines 35 and 36).

3. Transformation: A majority of the rules were implemented to either generate target model elements or produce intermediate results. The rules and functions are implemented according to the transformation process described in Section 3.2. For example, the rule `message2MsgTypeTmp` in Fig. 5.4 will produce one `MsgTypeTmp` (a temporary representation of `MsgType`) from each `Message` in a UML `Interaction`. Its tasks consist of 1) collecting the source and target classes associated with the two `MessageEnds`, i.e., the receive and send events of the Message (line 4 to line 11), 2) gathering the name of the `MsgType` (line 12 to line 16), and 3) constructing the `MsgTypeTmp` and storing it in the predefined global variable `MsgTypeTmps` (line 17 to line 30). The tasks are then further delegated to other supporting rules.

The first two kinds of rules do not produce any target model elements. However, they are vitally important for more efficient model transformations and to ensure the
5.2.3 Mapping Editor

In the SD-graphs integration process, the overlap between views is given as correspondences or mappings from the head to the views. In particular, to integrate Sequence Diagrams with our prototype, mappings from the head to the view Sequence Diagrams need to be specified.

We consider a mapping model $M_m$ as a container of a set of mapping pairs that connects a set of elements in a source Sequence Diagram $SD_s$ to a set of elements in a target Sequence Diagram $SD_t$. The meta-model for mapping models is illustrated in Fig. 5.5.

- **MappingContainer** is the root class of the mapping model that contains mapping elements, i.e., pairs. Because a mapping model stores the mappings between two specific Sequence Diagrams contained in two UML models, two string attributes of the MappingContainer class, `source` and `target`, are used to refer to the two UML models via the full qualified file names of the models.

- **Pair** is the class for recording a mapping between the `source` and the `target` Sequence Diagrams. Because a mapping (or a directed edge) can be identified by its source and target nodes, two string attributes, `so` and `ta`, identify the
elements in the source, and the element in the target that the mapping is connected to. To avoid ambiguity, the so or the ta attribute of a Pair must point to a single unique element in the source model or the target model. Fortunately, each element in an XMI document, e.g., serialized UML model, has a unique string identifier “xmi:id” associated with it. Thus, a so or ta should be set to the xmi:id of the element it references.

Currently, our prototype expects the user to provide the correspondences or mappings between Sequence Diagrams. In other words, mapping models are created manually by the users rather than automatically. To ease the job, an editor MappingEditor based on the Eclipse and the EMF platforms is provided as a necessary component of the prototype. It provides a simple tree-based user interface to create a new mapping model or to modify an existing one.

A screen-shot of the editor is shown in Fig. 5.6. It contains a list of standard views, such as the Selection view, generated by the EMF. A new view, Mapping, with three viewers is implemented to provide all the editing functions. The top left viewer provides the functions to access, and to select so elements for mappings from the source, i.e., a head Sequence Diagram. The top right viewer provides the similar functions for the target, i.e., one of the view diagrams to be merged. In contrast, the bottom viewer provides the functions to display and manage the mapping model. Common editing tasks, such as copy and paste of mapping model elements, are supported as well.
5.2.4 SG Editor

As the internal representation of Sequence Diagrams, SD-graphs need to be efficiently processed, manipulated, and generated by our SD integration tool. The Eclipse Modeling Framework (EMF) has been utilized to create APIs for manipulating, e.g., serializing, deserializing, and editing, SD-graphs. More precisely, the meta-model of SD-graphs shown in Fig. 5.2 has been first converted to an Ecore meta-model. Then, from the Ecore meta-model, Java classes for manipulating SD-graphs have been automatically generated by EMF.

Although a command-based editor has been generated by EMF, the graphical nature of SD-graphs suggests that visualization of SD-graphs, i.e., through graphical
editor, is also essential. To this end, the Graphical Modeling Framework (GMF) has been used to generate a graphical editor for SD-graphs. Besides the Ecore metamodel, three definition models have been created in order to allow GMF to generate the graphical editor. The **graphical** definition model defines the visual elements used in the editor, e.g., square and arrow with solid arrowhead. The **tooling** definition model defines elements related to the palettes, menus and so on of the editor, e.g., menu items “Event” and “Msg” under the “Palette”. The **mapping** definition model links the elements in the Ecore metamodel with elements in the graphical and tooling models using mappings, e.g., the “Event” element in the Ecore model for SD-graph (representing event occurrence in sequence graphs) is mapped to a square in the graphical definition model and to “Event” in the tooling definition model, and similarly “Msg” in the Ecore model is mapped to an arrow with solid arrowhead in the graphical model and to “Msg” in the tooling model. Finally, the graphical editor TG Editor is generated from the three definition models by GMF. A screen shot of the graphical editor is shown in Fig. 5.7.

Basic validation of SD-graphs is also supported by the graphical editor. Several OCL constraints were created, for instance, an **exec** edge must be a loop on the same class node. Then, EMF validators based on the constraints were automatically generated by GMF. Depending on the situations, the validators can be either automatically invoked to ensure that only valid SD-graphs’ elements can be added to a model, or manually executed to check the correctness of an existing model.
5.2.5 SG Merger

As we discussed in Chapter 3, the merge operation in our integration process is based on the pushout, or more precisely the colimit, of category theory. SG Merger is a component that consists of Java interfaces and classes, which implement the category of SD-graphs, and the operation to calculate the colimit.

To support the general concepts of category theory, the following Java interfaces, *Category*, *Object*, and *Arrow*, shown in the class diagram Fig. 5.8, have been created. To recap, a *Category* is a collection of *Objects* and *Arrows* such that the four properties, *identity arrow*, *composite arrow*, *associativity*, and *identity composition* are satisfied. The category for SD-graphs is a specific category with SD-graphs as its
Objects and mappings between SD-graphs as its Arrows. The interfaces are implemented by the three Java classes, **SGCategory**, **SGObject** and **SGArrow**. In addition to implementing the public methods defined in the interface Category, **SGCategory** also provides several private methods to check the four properties of the category.

The colimit of SD-graphs discussed in sections Chapter 3 has been implemented by a Java class, **SGColimit**. **SGColimit** has the following attributes: a set of SGObejcts **inObjs** as the input SD-graphs, one SGObejct **relObj** as the head SD-graph, a set of SGAArrows **inArrs** as the arrows between the input and the head SD-graphs, and finally a category **cat** containing the objects and arrows where the colimit can be constructed.
Before the colimit can be constructed on the span, the following preconditions need to be verified by the \textit{canPerformCO} method,

1. The category \textit{cat} must be a valid one, i.e., \textit{cat} satisfies the four properties of a category.

2. The input SD-graphs \textit{inObjs}, the head \textit{relObj} and the arrows \textit{inArrs} must be contained in the category \textit{cat}, e.g., \textit{inArrs} is a subset of the arrows of \textit{cat}.

3. For each arrow in \textit{inArrs}, its source must be the head SD-graph \textit{relObj}, and its target must be one of the SD-graphs in \textit{inObjs}.

4. For each arrow in \textit{inArrs}, the mappings contained in that arrow must be structure preserving, that is, they must not cause conflicts in the calculated colimit. For instance, consider the example shown in Fig. 5.9, which has a category \textit{c} consisting of two SD-graphs \textit{g} and \textit{g}' and an arrow \textit{arr} between \textit{g} and \textit{g}'. Assuming that \( n_1 \neq n_2 \) and \( n'_1 \neq n'_2 \), for each pair of edges \( e : n_1 \rightarrow n_2 \) in \textit{g} and \( e' : n'_1 \rightarrow n'_2 \) in \textit{g}', the following set of mappings in \textit{arr} cause conflict, \( e \mapsto e' \), \( n_1 \mapsto n'_2 \) and \( n_2 \mapsto n'_1 \). The reason of the conflict is that for the merged edge

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig5.9.png}
\caption{An arrow between two graphs contains conflict mappings.}
\end{figure}
from $e$ and $e'$, it is unclear which of the merged nodes, i.e., the nodes merged from $n_1$ and $n'_2$ and from $n_2$ and $n'_1$, will become the source and which will become the target.

If all the preconditions are met, the colimit will be computed by the performCO method. The method implements the merge algorithm detailed in Section 3.3 by the following steps, 1) it first puts $inObjs$ and $relObj$ into a disjoint union, 2) then the mappings from the head $relObj$ to $inObjs$ are used to identify equivalent elements in the disjoint union, 3) for each set of equivalent elements, only the element from the head $relObj$ is kept in the union and the remaining elements in the set are removed from the union, 4) any reference (e.g., the source and target of an edge, the incoming and outgoing edges of a node) which has become invalid because the element the reference pointed to has been removed during the previous step, is updated to ensure that it points to the removed element’s equivalent element in the head, 5) the cospan is generated, i.e., the merged SD-graph is generated from the union and then the merged SD-graph is wrapped in a new SGOBJECT $outObj$, new SGArs are generated from the inclusion arrows from $inObjs$ to $outObj$, and from $relObj$ to $outObj$, 6) finally, the category $cat$ is updated with the cospan.

5.2.6 Manager

The Manager facilitates the overall Sequence Diagrams integration process by routing data among the internal components. In addition to passing data around, certain data transformations and manipulations are provided by the Manager. Several inputs are required by the SG Merge, i.e., the attributes $inObjs$, $relObj$, $inArrs$ and $cat$ of the class SGPushOut. Some of the inputs are directly available, i.e., both $inObjs$
and $relObj$ are constructed by simply wrapping the generated SD-graphs from the SD2SG Transformer into SGObjects. But others, i.e., $inArrs$ and $cat$, have to be constructed by the Manager. The $inArrs$ are generated by transforming the mapping models between Sequence Diagrams produced by the Mapping Editor into SGArrows between SGObjects. Basically, from each mapping model between two Sequence Diagrams, a new SGArrow is created between the two Sequence Diagrams’ corresponding SGObjects. Then, from each mapping in the mapping model between two elements of the Sequence Diagrams, a new mapping contained in the SGArrow between two corresponding elements in the SGObjects is created. Finally, the category $cat$ is constructed from the objects, i.e., $inObjs$ and $relObj$, and the produced $inArrs$.

The Manager also acts as the interface between the user and the internal components. An Eclipse wizard, $MergeWizard$, shown in Fig. 5.10 has been implemented as a key part of the Manager. It guides the user through several steps which allow the user to provide the required inputs for the merge operation, and then invokes the internal components to produce the merged result. The bottom $Layer 1$ in Fig. 5.10 illustrates the first step of the wizard. By highlighting the Sequence Diagrams to be merged, i.e., the view diagrams, and the head Sequence Diagram, the user can access the $MergeWizard$ by right clicking the highlighted diagrams and then clicking the menu item “Merge Sequence Diagrams”. The second step of the wizard, i.e., $Layer 2$ in Fig. 5.10, lets the user to select the diagrams where the xmi:ids of the selected diagrams’ elements will be replaced by Universally Unique Identifiers (UUIDs). The second step is an optional one and has been implemented in order to support traceability. The detailed explanation regarding this step will be given later in this section. The third step, i.e., $Layer 3$ in Fig. 5.10, lets the user select the name and the location
Figure 5.10: The screen shots of the Sequence Diagrams Integration Eclipse Wizard.

of the model file for the merged Sequence Diagrams. The fourth step, i.e., Layer 4 allows the user select one Sequence Diagram from the highlighted Sequence Diagrams as the head diagram. In the last step, i.e., Layer 5, the user can load pre-specified mapping models between view and head Sequence Diagrams. Finally, the user clicks the finish button in Layer 5, and exit from the wizard. The merged result will be produced and stored in the selected model file.
5.2.7 Traceability

Traceability refers to the ability to track particular information through each step of a process chain. In our case, it should be possible to track a model element in a merged Sequence Diagram back to another model element in either one of the view Sequence Diagrams or to the head Sequence Diagram.

Tracing an element in the merged Sd-graph back to an element in the original SD-graph is straightforward due to the calculated “inclusion arrows” as part of the cospan and the arms in the span. However, the ability to trace an element in a merged Sequence Diagram back to an element in the original Sequence Diagram must be implemented separately as part of our Sequence Diagrams and SD-graphs transformations.

In our prototype, traceability across the Sequence Diagrams and SD-graphs transformations is supported and implemented through the collaboration of several internal components, i.e., through the preservation of xmi:ids by different components across the integration process. In an XMI document, e.g., a serialized UML model, each element has a string ID attribute, xmi:id, which is unique to the document. Although the chance is slim, it is possible that elements from different XMI documents share the same xmi:ids. To address this issue, the Layer 2 of the MergeWizard in Fig. 5.10 can replace the xmi:ids of a XMI document with UUIDs, which are unique across all XMI documents.

Xmi:ids are consistently preserved in our implementation. When transforming Sequence Diagrams to SD-graphs, rules are created in the SD2SG Transformer such that the xmi:id id of an element $sd_{in}$ in a Sequence Diagram is first recorded and then the same xmi:id id is given to the corresponding element $tg_{in}$ in the generated
SD-graph. During the merge phase of the integration process, the element $tg_{in}$ will have a corresponding element $tg_{out}$ in the merged SD-graph where both elements are connected by the span and the computed colimit. In case $tg_{in}$ is an element in the head or an element in one of the view diagrams but not merged with another element in the head, $tg_{out}$ will be assigned the same $\text{xmi:id}$ $id$ of $tg_{in}$. Otherwise, i.e., $tg_{in}$ is merged with an element $tg_{h}$ in the head diagram, $tg_{out}$ will be assigned the $\text{xmi:id}$ $id'$ of $tg_{h}$. Finally, the merged SD-graph is translated back to a Sequence Diagram by the SG2SD Transformer. Similar rules are defined in the SG2SD Transformer to preserve $\text{xmi:ids}$ such that an element in the final merged Sequence Diagram will have the same $\text{xmi:id}$ as its corresponding element in the merged SD-graph. For example, to track an element in an merged Sequence Diagram, we can simply follow the chain of preservation of $\text{xmi:ids}$ back to its original element(s) in the input and head Sequence Diagrams.

Nonetheless, because the span and the computed colimit of an integration operation are not stored after the integration, traceability is currently only available across a single integration operation.

### 5.3 Testing and evaluation

To evaluate the correctness of the implementation, we have run the prototype on a set of 19 test cases. The tests include, for instance, error test cases, variations of the brokered sale and all the examples given in Chapter 6. The merged results produced by the prototype were then visualized using IBM RSA. Finally, the visualized results were manually inspected and compared with the expected results. Most of the actual results are exactly the same as the expected results. Yet, some of the results visualized
in RSA are slightly different from the expected results, e.g., the placements of lifelines are different. We believe this is due to the layout algorithm used in RSA. Nonetheless, both the expected and actual results still have the same trace semantics. These evaluation results give us sufficient confidence that the prototype correctly implements our merge process. More comprehensive tests involving public evaluation of real case studies, and applications to some of the methodologies mentioned in the introduction are left for future work.
Chapter 6

Merge in practice

In this chapter, we investigate practical aspects of our general integration process and the prototype implementation in particular through high-level merge patterns. An important step during integration is the discovery and specification of the correspondences between views. We agree with [BCE+06] that an explicit representation of these correspondences is unavoidable for a general and manageable notion of model integration. However, our merge process requires the user to discover and specify these relationships. In this chapter, we present high-level merge patterns to alleviate this burden. For instance, the brokered sale example discussed in Chapter 3 suggests that combinations and variations of, e.g., lifeline composition and refinement constitute one such pattern.

Analogous to the use of patterns in software development, merge through patterns provides some significant benefits. First, patterns can present merge operations at a more abstract level, and hide some of the detail of the underlying merge operation by the automatic generation of head diagrams and mapping models. Second, patterns provide a list of basic operators to help construct more complex merge operations.
Finally, the reuse of parts of a merge becomes possible. For example, a previously constructed merge through patterns is no longer bound to a specific merge tool. Rather, the merge may be reused and carried out by another merge approach if the merge through patterns is supported by that approach.

To facilitate the application of our merge, in this chapter we will present a collection of high-level merge patterns. An overview of the discovered merge patterns and the associated terminology is given in Section 6.1. Two merge examples will be used to illustrate the introduced merge patterns in Section 6.2.

### 6.1 Merge patterns

A high-level merge pattern is a general reusable solution to a commonly occurring merge problem. A merge pattern by itself is not a complete merge. Rather, it provides the following:

1. an operator or a user-fillable template, such that relationships between diagrams could be described in more high-level terms by the users,

2. a mechanism allowing the details, such as head diagrams and mapping models, to be automatically generated.

One of the discovered patterns, i.e., simple composition, is inspired by existing UML constructs. The remaining patterns, i.e., adjacent and meet operators, insertion, addition and interaction refinement, have been identified through experimentation with our prototype. The list of patterns described here is the result of preliminary research, and by no means comprehensive. Moreover, the patterns will be presented in a somewhat informal way because more formal definitions of the patterns and any
associated mathematical proofs required are beyond the scope of this dissertation. Before we present the patterns, some required terminology will be defined first.

![Sequence Diagrams](image)

**Figure 6.1:** Example showing a Sequence Diagram divided by InteractionFragments.

- **InteractionFragment**: in the UML standard, an InteractionFragment is “a piece of an interaction. Each interaction fragment is conceptually like an interaction by itself”. For example, the Sequence Diagram (interaction) \(SD\) in Fig. 6.1 (a) can be divided into different InteractionFragments, the combination of two InteractionFragments \(IF_1\) and \(IF_2\) in Fig. 6.1 (b), or the combination of two InteractionFragments \(IF_3\) and \(IF_4\) in Fig. 6.1 (c). Although an InteractionFragment of a Sequence Diagram can be as complex as the whole Sequence Diagram, or as simple as a single occurrence on a lifeline, we do limit InteractionFragments to be complete ones, i.e., a sending message occurrence and its corresponding receiving message occurrence must be in the same InteractionFragment.

- **LifelineFragment**: a LifelineFragment is a piece of a lifeline. In contrast to
InteractionFragments which focus on interactions between lifelines, LifelineFragments focus on the behaviour of single lifelines. For example, the lifeline \( l_1 \) in Fig. 6.2 (a) can be represented by two LifelineFragments \( LF_1 \) and \( LF_2 \) in Fig. 6.2 (b), or \( LF_3 \) and \( LF_4 \) in Fig. 6.2 (c).

- **InitEvent**: an InitEvent references the first event on a lifeline or a LifelineFragment. For instance, the receiving of message \( m_1 \) in Fig. 6.2 (b) is the InitEvent of the LifelineFragment \( LF_1 \), and the sending of message \( m_2 \) is the InitEvent of the LifelineFragment \( LF_2 \).

- **EndEvent**: an EndEvent references the last event on a lifeline or a LifelineFragment. For instance, the sending of message \( m_2 \) in Fig. 6.2 (c) is the EndEvent of the LifelineFragment \( LF_3 \), and the end of execution is the EndEvent of the LifelineFragment \( LF_4 \).

- **TimePoint**: a TimePoint references a time instance on a lifeline. A TimePoint could coincide with an occurrence on a lifeline, e.g., \( TP_1 \) on lifeline \( l_x \) in Fig. 6.3. A TimePoint could also lie before, after, or in between occurrences, e.g., \( TP_2 \).
Matched Lifelines: lifelines in different diagrams are said to match if they are to be merged into a single lifeline in the resulting diagram. Throughout this chapter, unless explicitly stated otherwise, we assume lifelines are matched by names. More precisely, a lifeline $l_i$ in some diagram $SD_i$ matches another lifeline $l_j$ in another diagram $SD_j$ ($i \neq j$) if the name and the type of $l_i$ are the same as $l_j$. This is done for notational convenience and not a restriction, as matched lifelines could also be indicated by mappings.

6.1.1 Merge pattern 1: simple composition

Sequence Diagrams can be composed sequentially or in parallel. The sequential composition refers to placing one Sequence Diagram “on top” of another Sequence Diagram where each lifeline in the diagram matches exactly one lifeline in the other diagram. The matched lifelines will be linked together and a new Sequence Diagram is produced. In a parallel composition, on the other hand, no lifelines are matched.
and the two Sequence Diagrams are placed “side by side”. Although the same terms “sequential” and “parallel” compositions are used in the UML standard, they are different from our definitions of sequential and parallel compositions. For example, the “parallel” composition in the UML standard would put each pair of matched lifelines into a coregion, where our parallel composition only considers Sequence Diagrams with non-matching lifelines. Both our sequential and parallel compositions are defined as special cases of a general operator called simple composition.

![Diagram](comp)

**Figure 6.4: The pattern of the simple composition.**

**Definition 6.1.** The simple composition operator \( \text{comp} \) is defined as: \( SD_1 \times SD_2 \rightarrow SD \), where the two arguments \( SD_1 \) and \( SD_2 \) are Sequence Diagrams with an arbitrary number of matched lifelines, and the result of the simple composition operation is another Sequence Diagram \( SD \) such that for each pair of matched lifelines \( l_1 \in SD_1 \) and \( l_2 \in SD_2 \), the content of a new lifeline \( l \in SD \) will be the result of placing the content of \( l_1 \) “on top of” the content of \( l_2 \). A graphical representation of the operator is given in Fig. 6.4.

Before we discuss the general construction of the heads and mapping models for the simple composition operator, let us consider the two special cases first.

**Special case 1: sequential composition** In this case, all lifelines between \( SD_1 \) and \( SD_2 \) are matched. Consider Fig. 6.5 (a) and (b). Each Sequence Diagram
contains two lifelines and one message between the lifelines. To merge the input Sequence Diagrams using the simple composition operator \textit{comp}, a head Sequence Diagram $SD_h$ in Fig. 6.5 (c), and two mapping models between the head and the input Sequence Diagrams are provided. The result is shown in Fig. 6.5 (d). The head $SD_h$ has two lifelines, $l_1$ and $l_x$. Lifeline $l_1$ ($l_x$) contains an execution specification $R_1$ ($R_2$) with a start execution occurrence $SR_1$ ($SR_2$) and an end execution occurrence $ER_1$ ($ER_2$) respectively. Actually, only the four execution occurrences are needed for the merge operation. However, as explained earlier, in UML there is no way to specify or select execution occurrences without execution specifications. Therefore, both $R_1$ and $R_2$ have been provided and stereotyped as \texttt{<< rest >>}. For the mapping models between $SD_h$ and $SD_1$, and between $SD_h$ and $SD_2$, the mapping pairs are shown as the dashed-dotted arrows in Fig. 6.5. The mapping pairs are also listed in Table 6.1. In
Table 6.1: The mapping pairs required for the sequential case.

<table>
<thead>
<tr>
<th>Source: SD_h</th>
<th>Target: SD_1</th>
<th>Target: SD_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class L_1</td>
<td>Class L_1</td>
<td>Class L_1</td>
</tr>
<tr>
<td>Class L_x</td>
<td>Class L_x</td>
<td>Class L_x</td>
</tr>
<tr>
<td>Lifeline l_1</td>
<td>Lifeline l_1</td>
<td>Lifeline l_1</td>
</tr>
<tr>
<td>Lifeline l_x</td>
<td>Lifeline l_x</td>
<td>Lifeline l_x</td>
</tr>
<tr>
<td>Execution Occurrence SR_1</td>
<td>Message Occurrence EndM_1</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence ER_1</td>
<td>Message Occurrence RcvM_2</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence SR_2</td>
<td>Message Occurrence SndM_1</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence ER_2</td>
<td>Message Occurrence SndM_2</td>
<td></td>
</tr>
</tbody>
</table>

The table, a pair of cells will represent a mapping pair. For example, the cells Class L_1 in the Source : SD_h column and the Class L_1 in the Target : SD_2 column indicate there is a mapping from class L_1 in Sequence Diagram SD_h to class L_1 in Sequence Diagram SD_2. The merge result SD_r is shown in Fig. 6.5 (d).

Figure 6.6: The parallel case of the simple composition.

The effect of using simple composition when all lifelines in SD_1 and SD_2 are matched is that SD_1 and SD_2 are put together sequentially.
Special case 2: parallel composition In this case, no lifelines are matched between $SD_1$ and $SD_2$. Consider, e.g., Fig. 6.6 (a) and (b). An empty sequence diagram, i.e., no lifelines and no messages, is used as the head. The two mapping models between the head and the input Sequence Diagrams are also empty. The merge result is shown in Fig. 6.6 (c).

The effect of using simple composition when no lifeline in $SD_1$ and $SD_2$ is matched is that $SD_1$ and $SD_2$ are put together side-by-side.

![Figure 6.7: The general case of the simple composition.](image)

Both of the above special cases are subsumed by the following general case of simple composition.

Note that we can permute the lifelines in a Sequence Diagram. For instance, all matched lifelines may be placed on the left and enclosed in an InteractionFragment, whereas all non-matched lifelines are placed on the right and enclosed in another InteractionFragment\(^1\). As shown in Fig. 6.7 (a) and (b), an arbitrary number of

---

\(^1\)The placement of the lifelines in a Sequence Diagram can potentially help users comprehend
matched lifelines from $l_1$ to $l_k$ in both $SD_1$ and $SD_2$ are enclosed in $F_{11}$ and $F_{21}$ respectively. The non-matched lifelines from $l_{k+1}$ to $l_n$, and from $l'_{k+1}$ to $l'_n$ are enclosed in $F_{12}$ and $F_{22}$ respectively. Generalizing the above two special cases, we conclude that for a general definition of the simple composition operator $comp$, we need to add a lifeline to the head for each pair of matched lifelines. Moreover, an execution specification should be added to that lifeline in order be able to link the EndEvent of the matched lifeline in $SD_1$ to the InitEvent of the lifeline in $SD_2$. The head $SD_h$ in Fig. 6.7 (c) can be considered a combination of the heads of the “sequential” and “parallel” cases. Therefore, the result $SD_r$ in Fig. 6.7 (d) is also the combination of the two cases. On the one hand, it has the result from the “sequential” case, i.e., the merged lifelines $l_1$ to $l_k$ with $F_{11}$ placed on top of $F_{21}$. On the other hand, it also has the result from the “parallel” case, i.e., the lifelines in the InteractionFragments $F_{12}$ and $F_{22}$. In other words, simple composition can be described solely in terms of two successive applications of the two special cases mentioned above.

Finally, the binary simple composition operator can be extended to take any number of sequence diagrams as arguments, i.e., $comp_n: SD_1 \times \ldots \times SD_n \rightarrow SD$ can be defined through repeated applications of the binary simple composition operator $comp$,

$$comp_n(SD_1, \ldots, SD_n) = comp(\ldots comp(comp(SD_1, SD_2), SD_3), \ldots, SD_n).$$

Alternatively, the same merge result can be achieved through the building of a single head first and then creating mapping models such that the mappings will maintain the desired composition orders. E.g., the only valid compositions of $comp_3(SD_1, SD_2, SD_3)$ would be $SD_1$ “on top of” $SD_2$, and in turn “on top of” $SD_3$. From the single head the meaning of the Sequence Diagram. Yet, it does not affect the trace semantics of the Sequence Diagram.
and the mapping models, the result $SD$ can be generated by a single merge operation.

### 6.1.2 Merge pattern 2: adjacent and meet

The simple composition pattern can be used to merge Sequence Diagrams. However, when dealing with more complex situations, e.g., fragments of one lifeline needing to be interleaved with fragments from another lifeline, the pattern is insufficient. Inspired by use case interleaving introduced by Jacobson [Jac92], we consider interleaving of Sequence Diagrams as the result of merging LifelineFragments.

![Diagram of merge patterns](image)

**Figure 6.8:** The $adj$ and the $met$ operators.

In particular, we introduce the following two operators,

**Definition 6.2** (Adjacent and meet operators). *Both the adjacent operator $adj$ and the meet operator $met$ have the signature of: $SD_1 \times SD_2 \times LLF_1 \times LLF_2 \rightarrow SD$, where the arguments $SD_1$ and $SD_2$ refer to two Sequence Diagram to be merged which contain LifelineFragments $LLF_1$ and $LLF_2$ respectively. The graphical representations for the $adj$ and $met$ operators are given in Fig. 6.8 (a) and Fig. 6.8 (b).*

The goals of these two operators are similar. Both operators first explicitly identify a matching relationship between the two lifelines which the two LifelineFragments
belong to. Then, for the adjacent case, the two LifelineFragments \( \text{LLF}_1 \) and \( \text{LLF}_2 \) will be linked together, such that the InitEvent of \( \text{LLF}_2 \) will follow the EndEvent of \( \text{LLF}_1 \) immediately. The meet case is similar except that InitEvent of \( \text{LLF}_2 \) will be merged with the EndEvent of \( \text{LLF}_1 \) to a single event.

We will use the following examples to illustrate how details, e.g., heads and mapping models should be constructed for the two operators.

![Diagram](image)

Figure 6.9: Example showing a merge of LifelineFragments connected by the \textit{adj} operator.

**Example for adjacent operator \textit{adj}** In this case, we consider how the head and mapping models should be constructed when a pair of LifelineFragments are
connected by a single adj operator. The first two augments, SD₁ and SD₂, to the adj operator are the two input sequence diagrams shown in Fig. 6.9 (a) and (b). They contain two LifelineFragments f₁ of lifeline l₁ and f₂ of lifeline l₂, which are the last two arguments for the adj operator as shown in Fig. 6.9 (c). SD₁ and SD₂ may contain an arbitrary number of lifelines and occurrences, e.g., lifelines lₓ, lₘ and any additional lifelines between them in Fig. 6.9 (a).

Please note that, the lifelines l₁ and l₂ in the adjacent pattern are matched through explicitly equivalence mappings rather than by name. To link f₁ and f₂, the lifelines l₁ and l₂ will be merged together to lifeline l₁₂, and then the elements from f₁ will be placed “on top of” those in f₂ in the merged lifeline l₁₂ as depicted in the resulting Sequence Diagram SDᵣ in Fig. 6.9 (d).

To achieve the result, a head Sequence Diagram SDₕ is provided in Fig. 6.10 (c). Two mapping models between the head SDₕ, SD₁ and SD₂ are illustrated in Fig. 6.10 as mapping pairs. Essentially, by having the << rest >> stereotyped execution specification Exec₁₂ and its start and finish execution occurrences Ini₁₂ and End₁₂, we can specify that the InitEvent RcvM₂ happens after the EndEvent SendMₓ on the combined lifeline l₁₂ by the mapping arrows.
CHAPTER 6. MERGE IN PRACTICE

$\text{Ini}_{12} \leftrightarrow \text{Snd}M_x$ and $\text{End}_{12} \leftrightarrow \text{Rcv}M_2$. The remaining lifelines, the message and execution occurrences (e.g., the lifelines $l_x$, $l_m$ and additional lifelines between them) that do not appear in the mapping pairs are simply copied to the resulting Sequence Diagram $SD_r$.

![Lifeline Fragments Connected by a Single met Operator](image)

Figure 6.11: Example showing a merge of Lifeline Fragments connected by a single $\text{met}$ operator.

**Example for meet operator $\text{met}$** In this case, the EndEvent of LifelineFragment $f_1$ in Fig. 6.11 (a) needs to be merged with the InitEvent of LifelineFragment $f_2$ in Fig. 6.11 (b). The result of the merge is given in Fig. 6.11 (d).

In order to produce the result, a head Sequence Diagram $SD_h$ is provided in Fig. 6.12 (c). Two mapping models between the head $SD_h$, $SD_1$ and $SD_2$ are given in Fig. 6.12. Compared to the adjacent case, both the EndEvent $\text{End}M_x$ of LifelineFragment $f_1$ and the InitEvent $\text{Snd}M_y$ of LifelineFragment
Figure 6.12: The head and the mapping models for the meet merge case.

\( f_2 \) are mapped to the same execution occurrence \( Ini_x \). Thus, during the merge operation, they will be merged together into a single execution occurrence. As mentioned above, due to the way interactions are defined in UML, in order to be able to specify the execution occurrence \( Ini_x \), we have to provide the execution specification \( Exe_x \) and the execution occurrence \( End_x \) as well. Both \( Exe_x \) and \( End_x \) have been stereotyped as \( << \text{derived} >> \), and will be removed from the result.

In general, we believe that the \( adj \) and the \( met \) operators constitute some of the very basis of describing interleavings between Sequence Diagrams. The previous examples of using the \( adj \) and \( met \) operators demonstrate how head diagrams and mapping models should be constructed under the two patterns.

However, because consistency and correctness issues can easily arise, precautions are required when using the \( adj \) and \( met \) operators. In particular, assuming the input Sequence Diagrams are valid, the following post-conditions should be checked on the merge results to ensure that these two merge operations are meaningful and valid,

1. All traces obtained from the result of the \( adj \) and \( met \) operations should be acyclic. Consider, for example, the two Sequence Diagrams \( SD_1 \) and \( SD_2 \) in
Fig. 6.13 (a) and (b). Assuming that two \textit{adj} operators are used, i.e., one to merge $f_1$ and $f_2$, and the other to merge $f_y$ and $f_x$, the merge result according to the pattern would be $SD_r$ in Fig. 6.13 (c). But, the result clearly violates the partial order semantics of Sequence Diagrams by forming a cycle, i.e., the sending of the message $m_1$ happened after the receiving of $m_1$.

2. Despite the similarities between the \textit{adjacent} and the \textit{meet} patterns, the later requires even more careful considerations than the former. More precisely, no more than one message occurrence should occur at the same time for the result of the \textit{met} operator. For example, if we change the operators from \textit{adj} to \textit{met} in Fig. 6.9 (c), the merged result would never be a valid Sequence Diagram, because in UML a sending and a receiving message occurrence, e.g., the sending of message $m_x$ in Fig. 6.9 (a) and the receiving of message $m_2$ in Fig. 6.9 (b), should never occur at the same time.

\textbf{General description of the adjacent and meet operators:}

Both the \textit{adj} and the \textit{met} operators can be extended to take multiple sets of Lifeline-Fragments as arguments, e.g.: $adj_n(SD_1, SD_2, (LLF_1, LLF_2), \ldots, (LLF_{1n}, LLF_{2n}))$. 

Figure 6.13: Example showing an invalid Sequence Diagram.
CHAPTER 6. MERGE IN PRACTICE

Figure 6.14: Example showing a merge of LifelineFragments connected by the met operators.

Figure 6.15: The head and the mapping models for the multiple met operators example.

Again, the result can be obtained either through stepwise merge or through a single merge. The following example demonstrates how the head and mapping models should be constructed under the single merge approach. In this example, the EndEvents of LifelineFragments \( f_1 \) and \( f_x \) in Fig. 6.14 (a) need to be merged with the InitEvents of LifelineFragments \( f_2 \) and \( f_y \) in Fig. 6.14 (b). The result of the merge is shown in Fig. 6.14 (c). Please note that, to simplify the presentation, inside LifelineFragments \( f_x \) and \( f_y \), we considered the combination of \( RcvM_x, ExeM_x \) and \( EndM_x \) instead of only \( EndM_x \) as the EndEvent, and the combination of \( RcvM_y, ExeM_y \) instead of only \( EndM_y \) as the EndEvent.
Table 6.2: The mapping pairs required for the multiple meet operators example.

and $\text{EndM}_y$ instead of only $\text{EndM}_y$ as the InitEvent. Otherwise, according to the definitions of EndEvent and InitEvent, the merge of this example has to be carried out by three separated merge operations, i.e., between the lifelines $l_x$ and $l_y$, $\text{RcvM}_x$ and $\text{RcvM}_y$ will be merged together first, followed by $\text{ExeM}_x$ and $\text{ExeM}_y$ merged together and finally $\text{EndM}_x$ and $\text{EndM}_y$. Fig. 6.15 (c) shows the head Sequence Diagram $SD_h$ between two input Sequence Diagrams $SD_1$ and $SD_2$ in Fig. 6.15 (a) and (b). The mapping models between the head $SD_h$, $SD_1$ and $SD_2$ are given in Fig. 6.15 as mapping pairs. The mapping pairs are also listed in Table 6.2. Although the head and the mapping models look totally different from the head and the mapping models of the $\text{meet}$ example, they are actually just multiple applications of the single $\text{met}$ operator. As explained in the $\text{meet}$ example, what we really need is a
single occurrence, e.g., message or execution occurrence, on a lifeline in the head, in order to merge the EndEvent and InitEvent of two LifelineFragments connected by a \textit{met} operator. For example, since the sending message occurrence $SndM_{xy}$ on lifeline $l_{12}$ in the head $SD_h$ can be used to specify the mappings between $SndM_x$ on lifeline $l_1$ in $SD_1$ and $SndM_y$ on lifeline $l_2$ in $SD_2$, an execution specification and its start and finish occurrences as suggested in the \textit{meet} example are no longer needed. For the same reason, the execution specification $ExeM_{xy}$ and the finish execution occurrence $EndM_{xy}$ are not stereotyped, because they are needed for the specification of the mappings.

The construction of the heads and mapping models for the \textit{adj} operator with multiple sets of LifelineFragments is similar. Moreover, from the simple composition examples, it is clear that simple compositions are just special cases of using adjacent operators. One \textit{adj} operator would be placed in between each pair of matched lifelines.

6.1.3 Merge pattern 3: insertion and addition

The brokered sale example in Chapter 3 demonstrated how our integration process can be used to add new information, e.g., banking performed by the broker, to the merged result. More precisely, on top of merging elements between input and head diagrams, our approach is also capable of inserting elements from one diagram to another, and adding new elements to the merged result that are only present in the head and not in the input diagrams. Both inserting and adding elements can be achieved through the use of \textit{adj} operators with multiple LifelineFragments. Thus, we consider them as sub-patterns of the \textit{adjacent} pattern.

\textbf{Definition 6.3} (Insertion). \textit{The signature of the insertion operator \textit{ins} is defined as:}
Definition 6.4 (Addition). The signature of the addition operator \texttt{add} is defined as: $SD_1 \times LL_1 \times LLF_2 \times TP \rightarrow SD$, where \emph{LLF}_2 references a LifelineFragment or a lifeline in \emph{SD}_2 to be inserted into another lifeline \emph{LL}_1 at a particular TimePoint \emph{TP} in \emph{SD}_1.

Because a TimePoint can be a location anywhere on a lifeline, we limit the TimePoint \emph{TP} in the insertion pattern to be the following three locations, 1) before the InitEvent, 2) after the EndEvent, or 3) between two occurrences on the lifeline \emph{LL}_1. We consider cases where the \emph{TP} coincides with an occurrence as another pattern, i.e., the interaction refinement pattern, and will explain them later. The first two locations, i.e., before the InitEvent or after the EndEvent, can be handled by using the adjacent pattern. We will thus focus on the third location, i.e., between two occurrences. In general, the following three-step process can be used to insert elements from one lifeline or one LifelineFragment \emph{LLF}_2 into another lifeline \emph{LL}_1 at \emph{TP},

1. The lifeline \emph{LL}_1 can be first split into two LifelineFragments \emph{f}_1 and \emph{f}_2 at the insertion point \emph{TP}.

2. Following the adjacent pattern, one execution specification will be added to the combined lifeline in the head, such that the execution specification and its start and finish execution occurrences can be used to link \emph{f}_1 and \emph{LLF}_2.

3. Another execution specification will be added to the head on the same combined lifeline, such that the execution specification and its start and finish execution
occurrences can be used to link $LLF_2$ and $f_2$.

Adding new elements is a variation of the insertion pattern. The two Lifeline-Fragments $f_1$ and $f_2$ split from the first step in the process can be reconnected with a single $adj$ operator. The purposes of using the single $adj$ operator are not only to reconnect the two LifelineFragments, but also to use the execution specification in the head required by the $adj$ pattern as placeholder for specifying new elements from $LLF_2$.

Similar to the $adj$ and $met$ operators, both the $ins$ and $add$ operators can be extended to take multiple sets of LifelineFragments and TimePoints as arguments.

Figure 6.16: Example shows the inserting elements from one diagram to another.

The following example will be used to explain how the head and mapping models should be constructed when multiple sets of LifelineFragments and TimePoints are given for the $ins$ operator. The input Sequence Diagrams for the example are shown in Fig. 6.16 (a) and (b). The TimePoints $ins_1$ and $ins_2$ highlight the locations where the LifelineFragments $f_{ins_1}$ and $f_{ins_2}$ will be inserted into the lifelines $l_1$ and $l_x$ respectively in Fig. 6.16 (a). By following the process, the lifelines $l_1$ and $l_x$ each can be split at the insertion points $ins_1$ and $ins_2$ to the LifelineFragments $f_{11}$ and $f_{12}$, and $f_{x1}$
CHAPTER 6. MERGE IN PRACTICE

and $f_{x2}$ respectively as shown in Fig. 6.16 (a). Next, we apply the second and the third steps to merge the LifelineFragments, i.e., we create a head diagram $SD_h'$ containing the combined lifeline $l_{12}$ and the two execution specifications to link the LifelineFragments $f_{11}$, $f_{ins1}$, and then $f_{12}$. Giving the result Sequence Diagram $SD'$ produced from the first iteration, the second and the third steps can be taken again. Thus, another head diagram $SD_h''$ containing the combined lifeline $l_{xy}$ and the two execution specifications is created, and then the LifelineFragments $f_{x1}$, $f_{ins2}$, and $f_{x2}$ are linked together by the mappings. Alternatively, we can create a single head Sequence Diagram $SD_h$ in Fig. 6.17 (c), and the two mapping models between the head $SD_h$, $SD_1$ and $SD_2$ shown as mapping pairs in Fig. 6.17 and in Table 6.3. The merge result $SD_r$ is shown in Fig. 6.16 (c).

To highlight the differences between the insertion and the addition patterns, we will reuse the example in Fig. 6.16 to explain how the heads and the mapping models should be constructed for the addition pattern. The Sequence Diagram $SD_1$, where the new elements will be added to, is shown in Fig. 6.16 (a). Fig. 6.16 (b) shows

Figure 6.17: The head and the mapping models for the insertion example.
the new elements to be specified somewhere in the head diagram. Given the same TimePoints \( \text{ins}_1 \) and \( \text{ins}_2 \) in Fig. 6.16 (a), a new message \( m_2 \) should be added to the result in between the two existing messages \( m_1 \) and \( m_x \). Similar to the insertion example, the lifelines \( l_1 \) and \( l_x \) can be split at \( \text{ins}_1 \) and \( \text{ins}_2 \) respectively. Next, rather than creating two execution specifications per lifeline, only one is needed as shown in the head Sequence Diagram in Fig. 6.18 (b) for both \( l_1 \) and \( l_x \). The mapping model between \( SD_h \) and \( SD_1 \) as mapping pairs is shown in Fig. 6.18. If we only

Table 6.3: The mapping pairs required for the insertion example.
Figure 6.18: The head and the mapping models for the adding new elements example.

have the $\ll\text{wait}\gg$\(^2\) and $\ll\text{rest}\gg$ stereotyped execution specifications, and the mappings, this example would be just another application of using the $\text{adj}$ operators. However, new elements, e.g., the sending and receiving of message $m_2$, can be added to the merge result by specifying them on or between the stereotyped execution specifications in the head. The merge result of this example is again the one in Fig. 6.16 (c).

6.1.4 Merge pattern 4: refinement

Model refinement can be understood in terms of integration: given a model $A$, a refinement $A'$ of $A$ can be seen as the result of merging $A$ with a model $B$ elaborating parts of $A$. For example, the BrokeredSale scenario can be seen as a refinement of the WholeSale scenario, where the retail activity has been detailed by merging the RetailSale scenario via $R_{SG}$.

\(^2\)Because of the sending of the message $m_2$, it is necessary to stereotype the execution specification on lifeline $l_{12}$ as $\ll\text{wait}\gg$ instead of $\ll\text{rest}\gg$.
With such a view of refinement as integration, the relationships between the refined part of a model and its context are preserved by the operation of refinement. Nevertheless, this notion of refinement contrasts with the notion of refinement as proposed in, e.g., the STAIRS framework [HHRS05]. In STAIRS, the meaning of a Sequence Diagram is defined by the set of traces that any implementation must satisfy, and refinement is defined in terms of containment of sets of traces. We present an orthogonal view of refinement, where an action or sequence of actions (and therefore the corresponding trace) is refined by a sequence of actions (resp. a trace) by expansion, i.e., an action (an arrow in the sequence graph) can be refined by a composition of actions (arrows).

**Definition 6.5.** The signature of the refinement operator $\text{ref}$ is defined as $SD_1 \times IF_1 \times SD_2 \rightarrow SD$, where $SD_2$ is the refining Sequence Diagram or InteractionFragment, and $IF_1$ is the part to be refined and is an InteractionFragment enclosed in its context SequenceDiagram $SD_1$.

There is a close tie between the insertion and the refinement patterns. If we can turn $IF_1$, i.e., the part to be refined, into insertion points, the result of the refinement can be achieved through first inserting the refinement part $SD_2$ into $SD_1$ at the insertion points followed by the removing of $IF_1$ from its new context, i.e. the result after the insertion. More precisely, the following process can be used to construct the heads and the mapping models for the refinement operator $\text{ref}$,

1. Each lifeline and its contained occurrences in $IF_1$ are first represented by a TimePoint.

2. Next, depending on the types of the TimePoints, heads and mapping models can be created by following the adjacent pattern, i.e., for the TimePoints before
InitEvent or after EndEvent, and/or the insertion pattern, i.e., for between occurrences.

3. Finally, \( IF_2 \) should be explicitly removed from the result, e.g., set every element contained in \( IF_2 \) as “derived” for the current prototype.

![Figure 6.19](image1)

**Figure 6.19:** Example of interaction refinement pattern.

![Figure 6.20](image2)

**Figure 6.20:** The head and the mapping models for the refinement example.

We will use the following example to explain how the process can be applied. Two input sequence diagrams, \( SD_1 \) and \( SD_2 \), are given in Fig. 6.19 (a) and (b). \( SD_1 \) gives a simplified combined login and logout interactions used in ATM systems. It also acts as the context for the part to be refined, i.e., the enclosed InteractionFragment
$f_1$. $SD_2$ expands the login message by modeling the interactions between a user and an ATM.

<table>
<thead>
<tr>
<th>Source: $SD_h$</th>
<th>Target: $SD_1$</th>
<th>Target: $SD_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class User</td>
<td>Class User</td>
<td>Class User</td>
</tr>
<tr>
<td>Class ATM</td>
<td>Class ATM</td>
<td>Class ATM</td>
</tr>
<tr>
<td>Lifeline u</td>
<td>Lifeline u</td>
<td>Lifeline u</td>
</tr>
<tr>
<td>Lifeline a</td>
<td>Lifeline a</td>
<td>Lifeline a</td>
</tr>
<tr>
<td>Message Occurrence &lt;&lt;derived&gt;&gt;SndL</td>
<td>Message Occurrence SndLI</td>
<td></td>
</tr>
<tr>
<td>Message &lt;&lt;derived&gt;&gt; l</td>
<td>Message login</td>
<td></td>
</tr>
<tr>
<td>Message Occurrence &lt;&lt;derived&gt;&gt;RcvL</td>
<td>Message Occurrence RcvLI</td>
<td></td>
</tr>
<tr>
<td>Behavior Execution &lt;&lt;derived&gt;&gt; e</td>
<td>Behavior Execution exec</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence &lt;&lt;derived&gt;&gt;EndE</td>
<td>Execution Occurrence EndE</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence SU</td>
<td>Message Occurrence SndP</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence EU</td>
<td>Message Occurrence SndLO</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence SA</td>
<td>Execution Occurrence EndP</td>
<td></td>
</tr>
<tr>
<td>Execution Occurrence EA</td>
<td>Message Occurrence RcvLO</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: The mapping pairs required for the refinement example.

According to the first step of the process, two TimePoints are created to replace the part to be refined, i.e., $f_1$. During the second step, since both TimePoints are before the InitEvents of the lifelines they belonged to, the adjacent pattern can be followed to insert the refinement into the context. A head Sequence Diagram $SD_h$ in Fig. 6.20 (c) is created with one << wait >> and one << rest >> stereotyped
execution specification. Mapping pairs are specified in order to link the context and the refinement. Finally in the third step, some derived elements, i.e., the message $l$, the sending and receiving message occurrences of $SndL$ and $RcvL$, the execution $e$ and the execution occurrence $EndE$ are provided in the head as well. By stereotyping them as derived, and then setting the mappings between them to the elements in the InteractionFragment $f_1$, our integration tool will correctly recognize the elements in the InteractionFragment as the derived view of the refinement, and will remove them during the final step in our general integration process. The final mapping models are given in Fig. 6.20 and in Table 6.4. The result of this example is shown in Fig. 6.19 (c).

### 6.2 Pattern applications

In this section, we will show how these identified merge patterns can be applied to more realistic, and more complex examples.

#### 6.2.1 Brokered sale examples

The brokered sale example has been explained in Chapter 3. Here, let us consider an extended version of the brokered sale example.

In Fig. 6.21, we have an extended version of the sale Sequence Diagram. Compared to the original sale Sequence Diagram in Fig. 3.1, the extended sale scenario contains a new lifeline $p$ of class $Product$. Moreover, the seller $s$ can set and the buyer $b$ can inquire the price of the same product. Again, the merge goal of this example is to

---

3 For conciseness, some of the mapping pairs are not shown, e.g., between the $\ll derived \gg$ $SndL$ to $Snd_{L1}$. 


build a brokered sale scenario from two identical copies of the extended sale scenario. In addition to the five requirements identified for the original brokered sale example, we will have one additional requirement for this test case: (0) Both sales deal with the same \textit{product}.

By analyzing the requirements, we can discover that the requirements 1, 2, 3, and 5 can be achieved by taking two identical copies of the Sequence Diagram in Fig. 6.21 and using the insertion pattern explained earlier. To follow the pattern, the insertion point has to be identified first, i.e., the identification of the TimePoint between message occurrences \textit{Snd}_D and \textit{Snd}_P on the lifeline \textit{b} in the whole sale Sequence Diagram \textit{SD}_{ws}. Then, the lifeline \textit{b} can be split into two LifelineFragments \textit{f}_1 and \textit{f}_2. The insertion is identified as a LifelineFragment \textit{f}_{ins} on the lifeline \textit{s} in the retail sale Sequence Diagram \textit{SD}_{rs}. The remaining steps in the pattern consist of creating the required execution specifications and specifying mappings among the diagrams. Thus,
a head $SD_h'$ in Fig. 6.22 (c), and mappings between $SD_{ws}$ and $SD_{rs}$ are specified as in Fig. 6.22.

To satisfy requirement 4, we can follow the addition pattern. Thus, we modify the previous head slightly by adding thinking and banking to the execution specifications. The updated head $SD_h''$ is given in Fig. 6.22 (d).

Finally, to fulfill the new requirement for the extended example, we can follow the adjacent pattern on the lifelines $p$ in $SD_{ws}$ and $p$ in $SD_{rs}$ by adding one more execution specification and creating the required mappings.

The patterns, i.e., insertion, addition and single adjacent, can be applied stepwise. Though, through visual inspection, the result produced from the stepwise merge seems
correspond to the result generated from a single final head $SD_h$ given in Fig. 6.23 (c). Thus, all three merge operations needed for the patterns seems can be replaced by a single merge\textsuperscript{4}. The mapping pairs for the single merge are given in Fig. 6.23 and in Table 6.5. The generated brokered sale Sequence Diagram is given in Fig. 6.24.

\footnote{\textsuperscript{4}Formal proof of the equivalence between the effects of the single merge and the stepwise merge is beyond the scope of this dissertation.}
Table 6.5: The mapping pairs required for the extended brokered sale example.

<table>
<thead>
<tr>
<th>Source: SD</th>
<th>Target: SDprev</th>
<th>Target: SDnext</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Retailer</td>
<td>Class Buyer</td>
<td>Class Seller</td>
</tr>
<tr>
<td>Class Product</td>
<td>Class Product</td>
<td>Class Product</td>
</tr>
<tr>
<td>LifeLine r</td>
<td>LifeLine b</td>
<td>LifeLine s</td>
</tr>
<tr>
<td>LifeLine pr</td>
<td>LifeLine p</td>
<td>LifeLine p</td>
</tr>
</tbody>
</table>

Execution Occurrence
| BE | Message Occurrence SndD |
| SE | Message Occurrence SndS |
| SS | Execution Occurrence EndP |
| PE | Message Occurrence SndP |
| PS | Message Occurrence SndM |
| PE | Message Occurrence RcvS |

Figure 6.24: The result of the extended brokered sale.
6.2.2 Home Appliances Control System (HACS)

HACS [HTUK] is a system that integrates wireless devices with home appliances. Through the control of a HACS, wireless devices, e.g., cellular phones, can remotely control home appliances, e.g., microwave ovens. A set of use cases and Sequence Diagrams are provided in [HTUK], for example:

![Sequence Diagrams](image)

Figure 6.25: A login scenario $SD_{LI}$ of HACS and its refinement.

- **Login:** To authorize legitimate access to HACS and to prevent malicious use of the system, users have to provide login information to HACS. Then, the login information is validated against user information stored in a database, and finally legitimate users are granted access to HACS by showing the system menu. Fig. 6.25 (a) illustrates a positive login scenario $SD_{LI}$ such that a legitimate user is granted the access to HACS. The *login* message can be further refined by the messages *username* and *password* as shown in Fig. 6.25 (b).

- **Logout:** To further protect the system, users will be logged out from the system automatically after idling for a predefined period of time. Fig. 6.26 shows a logout scenario $SD_{LO}$. 
Check status: Users can select from a list of appliances through HACS, and then check the status of the appliance, such as whether the appliance is turned on or not. A check status scenario $SD_{CS}$ is given in Fig. 6.27.

Given the login, logout, check status scenarios and the refinement of login, the goal of this merge example is to generate a complete $getstatus$ scenario by integrating the four scenarios. Integration should proceed in such a way that the following requirements are satisfied:

(i) A user has to login to the system first, before he or she can check the status of
(ii) After login, the user will check the status of the appliance.

(iii) After detecting that the user has been idling for a while, the system should automatically logout the user.

(iv) The login message should be replaced by its refinement, i.e., the username and password.

Figure 6.28: The head Sequence Diagram $SD_h'$ and mappings for the sequential composition part of the HACS example.

The first three requirements suggest that one of the valid scenarios can be the merge result by following the simple composition pattern on $SD_{LI}$, $SD_{CS}$ and $SD_{LO}$. The head sequence diagram $SD_h'$ to support the simple composition is given in
Fig. 6.28 (d). The mapping pairs are given in Fig. 6.28\(^5\).

Figure 6.29: The head Sequence Diagram $SD_h$” and mappings for the refinement part of the HACS example.

The last requirement indicates that we can follow the interaction refinement pattern. The head for the refinement $SD_h$” and the mappings are given in Fig. 6.29.

Figure 6.30: The combined head $SD_h$ for the HACS example.

Instead of these two consecutive merge operations, we can also use a single merge with the head $SD_h$ in Fig. 6.30 and mapping pairs listed in Table 6.6 to produce

\(^5\)For conciseness, the mapping pairs between the lifelines and classes are not shown.
the same result shown in Fig. 6.31 as the stepwise merge. In general, the question to what extent the heads of serial consecutive merge operations can be combined to obtain a single merge that achieves the same result is a very interesting question for future research. In this particular case, the head for the single merge is constructed as some kind of simple union of the heads of the consecutive merge steps.
### Table 6.6: The mapping pairs required for the HACS example.

<table>
<thead>
<tr>
<th>Source: $SD_h$</th>
<th>Target: $SD_{L1}$</th>
<th>Target: $SD_{CS}$</th>
<th>Target: $SD_{LO}$</th>
<th>Target: $SD_{ref}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class User</td>
<td>Class User</td>
<td>Class User</td>
<td>Class User</td>
<td>Class User</td>
</tr>
<tr>
<td>Class HACSCtrl</td>
<td>Class HACSCtrl</td>
<td>Class HACSCtrl</td>
<td>Class HACSCtrl</td>
<td>Class HACSCtrl</td>
</tr>
<tr>
<td>Lifeline u</td>
<td>Lifeline u</td>
<td>Lifeline u</td>
<td>Lifeline u</td>
<td>Lifeline u</td>
</tr>
<tr>
<td>Lifeline c</td>
<td>Lifeline c</td>
<td>Lifeline c</td>
<td>Lifeline c</td>
<td>Lifeline c</td>
</tr>
<tr>
<td>Message Occurrence $\langle \text{derived} \rangle \text{SndM}$</td>
<td>Message Occurrence SndLI</td>
<td>Message Occurrence SndLI</td>
<td>Message Occurrence SndLI</td>
<td>Message Occurrence SndLI</td>
</tr>
<tr>
<td>Message $\langle \text{derived} \rangle \text{m}$</td>
<td>Message login</td>
<td>Message login</td>
<td>Message login</td>
<td>Message login</td>
</tr>
<tr>
<td>Message Occurrence $\langle \text{derived} \rangle \text{RcvM}$</td>
<td>Message Occurrence RcvLI</td>
<td>Message Occurrence RcvLI</td>
<td>Message Occurrence RcvLI</td>
<td>Message Occurrence RcvLI</td>
</tr>
<tr>
<td>Behavior Execution $\langle \text{derived} \rangle \text{ExeM}$</td>
<td>Behavior Execution ExeLI</td>
<td>Behavior Execution ExeLI</td>
<td>Behavior Execution ExeLI</td>
<td>Behavior Execution ExeLI</td>
</tr>
<tr>
<td>Execution Occurrence $\langle \text{derived} \rangle \text{EndE}$</td>
<td>Execution Occurrence EndLI</td>
<td>Execution Occurrence EndLI</td>
<td>Execution Occurrence EndLI</td>
<td>Execution Occurrence EndLI</td>
</tr>
<tr>
<td>Execution Occurrence U0S</td>
<td>Execution Occurrence EndS</td>
<td>Execution Occurrence EndS</td>
<td>Execution Occurrence EndS</td>
<td>Execution Occurrence EndS</td>
</tr>
<tr>
<td>Execution Occurrence U0E</td>
<td>Message Occurrence SndP</td>
<td>Message Occurrence SndP</td>
<td>Message Occurrence SndP</td>
<td>Message Occurrence SndP</td>
</tr>
<tr>
<td>Execution Occurrence U1S</td>
<td>Message Occurrence RcvS</td>
<td>Message Occurrence RcvS</td>
<td>Message Occurrence RcvS</td>
<td>Message Occurrence RcvS</td>
</tr>
<tr>
<td>Execution Occurrence U1E</td>
<td>Message Occurrence SndG</td>
<td>Message Occurrence SndG</td>
<td>Message Occurrence SndG</td>
<td>Message Occurrence SndG</td>
</tr>
<tr>
<td>Execution Occurrence U2S</td>
<td>Execution Occurrence EndD</td>
<td>Execution Occurrence EndD</td>
<td>Execution Occurrence EndD</td>
<td>Execution Occurrence EndD</td>
</tr>
<tr>
<td>Execution Occurrence U2E</td>
<td>Message Occurrence RcvL</td>
<td>Message Occurrence RcvL</td>
<td>Message Occurrence RcvL</td>
<td>Message Occurrence RcvL</td>
</tr>
<tr>
<td>Execution Occurrence C0S</td>
<td>Execution Occurrence EndP</td>
<td>Execution Occurrence EndP</td>
<td>Execution Occurrence EndP</td>
<td>Execution Occurrence EndP</td>
</tr>
<tr>
<td>Execution Occurrence C0E</td>
<td>Message Occurrence SndV</td>
<td>Message Occurrence SndV</td>
<td>Message Occurrence SndV</td>
<td>Message Occurrence SndV</td>
</tr>
<tr>
<td>Execution Occurrence C1S</td>
<td>Message Occurrence SndS</td>
<td>Message Occurrence SndS</td>
<td>Message Occurrence SndS</td>
<td>Message Occurrence SndS</td>
</tr>
<tr>
<td>Execution Occurrence C1E</td>
<td>Message Occurrence RcvG</td>
<td>Message Occurrence RcvG</td>
<td>Message Occurrence RcvG</td>
<td>Message Occurrence RcvG</td>
</tr>
<tr>
<td>Execution Occurrence C2S</td>
<td>Message Occurrence SndD</td>
<td>Message Occurrence SndD</td>
<td>Message Occurrence SndD</td>
<td>Message Occurrence SndD</td>
</tr>
<tr>
<td>Execution Occurrence C2E</td>
<td>Message Occurrence SndT</td>
<td>Message Occurrence SndT</td>
<td>Message Occurrence SndT</td>
<td>Message Occurrence SndT</td>
</tr>
</tbody>
</table>
Chapter 7

Related work

The composition of behavior models has already been explored in different contexts. For instance, UML2 offers Interaction Overview Diagrams (IOD) [Obj07], while MSCs have been generalized to high-level MSCs [IT00]. These diagrams are essentially graphs whose nodes represent scenarios and edges show the control flow between them. Thus, behaviors specified by nodes are considered non-overlapping. In contrast, we address the issue of how to specify overlap between scenarios, and then integrate them without duplication. A similar problem has been studied in semantic data modeling for a long time, where it is called schema or view integration (see the recent survey [AB07] and a widely cited paper [PB03]). Most work in view integration is non-generic in the sense that definitions and algorithms depend on the use of a particular modeling language. However, a general and data-model independent approach to the problem can be designed with category theory means [Dis05]. The categorical framework has been used for schema merging in the contexts of database design [CD96] and metadata management [Dis05], for early requirement engineering [SE05], and for software merge [NES05].
While view integration is well studied in the context of data modeling, only a few papers attempt to tackle the issue in behavior modeling [UC04, DFKM98], including a categorical approach to merging MSCs in [KBH04]. The most important distinction of our approach is that we work with views augmented with derived elements because information considered basic in one view may be derived in another. This phenomenon is crucial to the integration problem and gives rise to an effective notion of refinement. Another distinction is that we allow for the specification of new information not captured by views during the identification of their overlap. In [KBH04], categorical machinery is employed for merging partially-ordered multisets representing MSCs. Thus, the formalization in [KBH04] is “string-based”, rather than graph-based like ours. We argue that this change in representation makes the integration procedure less transparent and more difficult to learn. Finally, only injective morphisms are considered in [KBH04], which might be a serious yet not relevant restriction. On the other hand, [KBH04] considers also control structures in scenario modeling (high-level MSCs), which we leave for future work.

To facilitate the merge of scenarios based on amalgamated sum proposed in [KBH04], an algorithm to automatically compute the common parts between two MSCs is proposed in [HHC06]. A common part is similar to a head plus the arms in our approach that identifies the common elements between diagrams. The quality of a common part is based on the ratio between the size of the common part and the size of the smaller input diagram. To efficiently compute the best common parts, common elements are limited to events located on the same instances and having the same labels. The algorithm in [HHC06] could potentially be applied in our approach to generate heads and arms. Nonetheless, a pattern-based approach like the one proposed in Chapter 6
would allow users to perform much more general and flexible merge operations.

Recently, aspects weaving have been proposed for the integration of aspects into different kinds of models (e.g., MSCs, Class Diagrams, Sequence Diagrams, and state machines). In [KHJ06], pointcuts (scenarios specified as basic MSCs) are first created and semantically matched in Hierarchical-MSCs, and then occurrences of matched pointcuts are replaced by advices (also scenarios specified as basic MSCs). In [WJ07, WMA+07], a pattern language is used for the specification of the overlap between models and aspects and then the weaving is performed based on graph transformation. Compared to our work, the use of a pattern language to specify the relationships between models in [WJ07, WMA+07] or the specification and semantic matching of pointcuts makes the approaches more user-friendly. However, it also makes the approaches less general and versatile because possibly quite different pattern languages or specification languages need to be defined for different diagram types; since the languages need to strike a delicate balance between expressiveness and usability, this step may not be straight-forward. Finally, our approach does not exclude the use of patterns and in fact we summarized a list of high level merge patterns in Chapter 6. Similar to the composition operators suggested in [JK08], our patterns are the first step towards increasing the ease of use of our approach yet still allow users to perform more general and versatile merge operations.

Some work on the synthesis of state-based behavior models from scenarios involve the integration of scenarios as a by-product (see [LDD06] for a survey). These approaches allow for complex forms of integration, and the labeling mechanism in [KGSB99] is clearly akin to our idea of explicit mappings (“arms”). However, our approach also allows us to include new information in the correspondence “head”, which is not
contained in any of the integrated models. As our brokered sale example illustrates, this new information can provide a new useful context into which the merged scenarios are placed. In addition, our approach inherits some useful properties from the underlying colimit operation (traceability and universality), which the majority of scenarios-from-state-machines procedures do not seem to offer.

Tools supporting tasks related to model management, particularly, model merging already exist, e.g., AMW [FV07] and Epsilon [KPP06b]. Conceivably, they could have been used to aid in the construction of our prototype. However, we chose to implement it without the use of existing merge tools because we wanted to study our approach in its purest form, unencumbered by the issues that can arise when encoding one theory in another. Moreover, we wanted to see to what extent the theory and its properties would be leveraged on the implementation level. We acknowledge, however, that the manual specification of mappings supported by AMW is similar to ours and could have been used for our mapping editor. In future work, it would be interesting to investigate the combination of AMW’s automatic mapping generator or Epsilon’s rule-based matching with our categorical approach.
Chapter 8

Future work and conclusions

8.1 Limitations and future work

In this section we will briefly discuss some possible extensions and limitations of the theory and the implementation.

On the theory side, we consider the following as starting points for interesting future work.

- **Beyond the merge operation.** Other operations over typed graphs can be useful in scenario management. For example, finding the differences, or extraction of the common part from two or more scenarios. It is shown in category theory that this and other counterparts of ordinary set-theoretical operations can be performed with typed graphs as well [BMSW06].

- **Support more constructs.** Currently, our approach to integrate Sequence Diagrams is limited to some core constructs of Sequence Diagrams. Some additional constructs are relatively easy to support. For example, one advantage of
our approach is the explicit representation of typing information and therefore consistency checking on scenario-based behavior models is possible. This type discipline can be elaborated even more to support synchronized messages in our approach. More precisely, it is possible to introduce a special edge \textit{return} between edges, in order to specify relations between procedure calls and their return messages.

To support other constructs of Sequence Diagrams, our approach might require some significant changes. For example, to support interaction operators, e.g., alternative or parallel, the current three-layer SD-graphs might have to be replaced by a four-layer typed graph. The lower three-layer are SD-graphs. The top layer would be a graph similar to a UML Interaction Overview Diagram. The nodes of the graph are references to SD-graphs in lower layers, and the edges of the graph represent the operators. Moreover, the procedure to merge such four-layer graphs is likely more complex as we have to take the top layer into consideration.

Finally, to help validate the correctness of SD-graphs it would be desirable to support predicates or constraints by our approach. For example, so far we implicitly assumed that each local execution in a Sequence Diagram is started by a single occurrence, i.e., a start execution occurrence or a receiving message occurrence. To explicate this condition, we need to specify the corresponding predicate to the object graph. In general, multiple predicates/constraints may be incorporated into the SD-graph. The main difficulty with introducing constraints into the formalism is that they essentially affect its algebraic properties. Graphs with constraints are \textit{sketches}, which are much more expressive but much
harder to work with [DK03]. We plan to explore the possibilities and problems of using generalized sketches in scenario based modeling in the nearest future.

- **Heterogeneous view integration.** Scenarios specified in different languages can be mapped to a particular kind of typed graphs, e.g., SD-graphs, and then integrated. The language of graphs is sufficiently expressive to make this idea practically interesting. For example, because of the similarities between MSCs and Sequence Diagrams, MSCs (or at least the core constructs of MSCs) can be transformed to SD-graphs, and therefore we believe merging them should not be too difficult.

- **Inconsistent scenarios.** The annotation of scenarios with type information not only facilitates the integration operation presented in this dissertation, but also creates interesting possibilities for model management in general and the discovering of inconsistences in particular.

- **General merge operation.** Our approach to integration is quite general. Although we showed a two-way merge example, our merge process and prototype tool actually support multi-way merge with any number of scenarios and heads. Moreover, the merge process is not limited to Sequence Diagrams (or MSCs), but can be easily extended to other kinds of models, e.g., Communication Diagrams or Class Diagrams. In principle, every type of diagram representable as a directed, typed graph is amenable to our integration approach. In the future, we would like to evaluate and apply our general integration process on other kinds of diagrams, for example UML Class Diagrams.
On the implementation and application side, we would like to improve our prototype in the followings directions.

- **Merge patterns.** In Chapter 6, we introduced a list of high-level merge patterns. With the proposed patterns, merge of Sequence Diagrams can be constructed more automatically. In the future, we will modify the current prototype in order to support the patterns. For example, each pattern can be supported through a sub-wizard which can be invoked by the existing MergeWizard. Moreover, more graphical editors are likely needed as well in order to specify pattern related concepts, e.g., TimePoint.

- **Improved wizard.** Currently, users of our prototype have to have the mapping models ready before they can merge Sequence Diagrams, i.e., loading pre-specified mapping models first and then performing merges on Sequence Diagrams through the MergeWizard. This limitation could be removed by modifying the wizard such that multiple flows are supported by the MergeWizard. For example, rather than loading pre-specified mapping models, the MergeWizard could allow users to choose between either creating and loading new mapping models on the fly by using our MappingEditor, or using existing approaches to automatically generate mapping models, e.g., our proposed merge patterns or the AMW’s automatic mapping generator.

- **Expert mode.** In our current implementation, the concepts and the representation of SD-graphs are essentially hidden from the users. For example, as we mentioned earlier in Chapter 5, the users create mappings or correspondences between Sequence Diagrams by using the MappingEditor. However, if we put
the benefits of our current approach aside, it does limit the users to only those concepts defined in Sequence Diagrams, and potentially could be less efficient, e.g., mappings between Sequence Diagrams needed to be translated to correspondences between SD-graphs. Thus, we are looking forward to implementing an expert mode which allows users to work on SD-graphs directly. In this expert mode, the users will be fully exposed to SD-graphs. For example, the users will have the choice between creating mappings between SD-graphs, or having the benefit of examining the merged SD-graphs with the implemented SG Editor before they are translated back to Sequence Diagrams.

- **Scenario-based testing support.** In scenario-based testing, test scenarios are created to describe the behaviour of a system under test. Concrete test cases are then generated from the test scenarios, for example, the generation of test cases from MSCs in [Ebn04] and the generation of test cases from Sequence Diagrams in [LLQC07]. Like regular testing, scenario-based testing is subject to constant system evolution or feature addition and deletion. Thus, test cases need to be rapidly created in order to test the new features or be modified in order to test changed routines. In turn, test scenarios need to be created or updated. To support the requirement of rapid test scenarios generation or modification in scenario-based testing, our approach could be potentially applied. For example, new test scenarios could be created by merging parts from existing test scenarios, or modified test scenarios could be produced by inserting new information to the original test scenarios with the help of our integration approach and high level merge patterns.

With respect to the model transformation via TXL, in the future, we would like
to investigate the following topics:

- Our Ecore2Grammar generator currently does not support OCL constraints. However, we would like to investigate the possibility to generate TXL rules from OCL constrains. The concept of generating rules from OCL constrains has been discussed in [WTEK08], where a restricted form of OCL constraints can be translated to graph constraints. Such generated rules then can be utilized to perform well-formedness checks on models.

- The built-in tracing as part of TXL debug options shows the sequence of rules applied during a transformation. Although TXL does not support traceability between target model and source model directly, i.e., the links between target model elements and source model elements may be scattered and interleaved throughout the applied rules, traceability could be established by reorganizing the tracing information provided by TXL in a more human readable format.

- In Section 4.1, we provided automatic grammar generators. It would be desirable if the rules that operate on these generated grammars could be generated as well, even as skeleton rules or functions. We are interested in the following two different approaches. For the first approach, we could fuse a graphical rule editor with TXL. More precisely, we can take advantage of graphical rule specifications provided by some of the graph transformation approaches and then translate graphical rules to TXL rules. This hybrid approach could be further used to realize the generic model transformation based on category theory’s pull-back operation explained in [DD06]. In the second approach, we could follow the “rule by examples” principle established in TXL programming and
use heuristics to infer (possibly partial) rules from user provided transformation examples.

- The example in Chapter 4 revealed that current model transformations via TXL have to be written at a fairly low level of abstraction, i.e., at the XML/XMI tags level. We acknowledged that it is a major obstacle to using TXL for model transformation. In the future, we would like to investigate the possibility of creating a domain specific language (DSL) for TXL model transformation, such that developers of TXL model transformation could work with more abstract and familiar terminologies.

8.2 Conclusion

It is widely accepted within the software engineering community that the support for integration is necessary for requirement models. Several methodologies have appeared in the literature that are relying on some kind of integration. However, the need for proper model integration has not yet been met. In this dissertation, we discussed our solution to the problem. More precisely, we presented a general integration process for scenario-based models, particularly UML Sequence Diagrams, based on the colimit construction known from category theory.

The typed graphs, or in particular SD-graphs, merge algorithm contained in the general process is an extension of existing merge operations on sets and graphs. On the one hand, the merge algorithm ensures traceability and guarantees that everything is represented and nothing extra is acquired in the merge. On the one hand, our formalization of Sequence Diagrams as SD-graphs, i.e., three-layer typed graphs,
retains the graphical nature of Sequence Diagrams, yet is amenable to algebraic manipulations and applicable to the merge algorithm. Another important property of our process is that the process is applicable to other kinds of models as long as they can represented by typed graphs. Finally, any number of models can be merged.

A prototype tool following the approach has been implemented. The tool not only served as a test bed for our theory and provided feedback about our theoretical framework, but also can be used for the educational purposes, for instance, to demonstrate or explain some of the theory behind the prototype tool, e.g., category theory.

We believe our theory and tool are beneficial to both academia and industry, as the initial evaluation has shown that the ideas presented in this dissertation represent promising steps towards more rigorous management of requirement models. However, the support for discovery and specification of model relationships needs to be improved. One possible approach is using heuristic, e.g., high-level merge patterns presented in this dissertation.

Also in this dissertation, we presented an approach allowing TXL to be used as a model transformation tool. Our model transformation approach benefits from explicit rule scoping, flexible rule application strategy and user controllable rule scheduling provided by TXL, which are some of the essential properties that affect the performance and efficiency of model transformations. Moreover, our grammar generators significantly ease the task of creating model transformations through the automatic generation of TXL grammars from meta-models.
Bibliography


[GRe] GReAT. GReAT: Graph Rewriting And Transformation. Online: http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp.


Glossary

Category theory  A branch of mathematics in which mathematical structures and their relationships are studied.

Class Diagrams   A Class Diagram shows a collection of classes and the relationships between them.

Colimit         A colimit is the general description of some concrete constructions, such as pushouts, coproducts and coequalizers. The dual notion of colimit in category theory is limit.

EMF             The Eclipse Modeling Framework (EMF) is a simpler than MOF metamodeling approach provided by Eclipse.

GMF             The Eclipse Graphical Modeling Framework (GMF) is an open framework for developing new graphical editors.

MDA             The Model-Driven Architecture (MDA) is the Object Management Group (OMG)’s standard for MDE.

MDE             The Model-Driven Engineering (MDE) is a software development methodology which is based on the use of models.
Meta-modeling  Meta-modeling is an abstraction about models and allows the development of models of models.

Model transformation  A model transformation transforms an input model conforming to one meta-model to an output model conforming to another meta-model.

MOF  The Meta-Object Facility (MOF) is the OMG’s standard to support meta-modeling.

Object Diagrams  An Object Diagram shows a specific instance of a system described by a Class Diagram.

OCL  The Object Constraint Language (OCL) is an OMG standard for describing constraints using declarative rules.

Pushout  A pushout of two arrows $f: R \to S_1$ and $g: R \to S_2$ between objects $R$, $S_1$ and $S_2$ in a category is an object $P$ with two arrows $f': S_1 \to P$ and $g': S_2 \to P$ that satisfies the universal property.

RSA  IBM Rational Software Architect (RSA) is a commercial UML modeling tool.

Sequence Diagrams  A Sequence Diagram specifies the interactions between a set of objects indicated by lifelines over time.

TXL  TXL is a hybrid functional and rule-based programming language designed to support software analysis and source transformation tasks.
UML   The Unified Modeling Language (UML) is a general purpose modeling language standardized by the OMG.

UML2   The Eclipse UML2 framework provides the implementation of the UML 2.x meta-model as a suite of APIs for the Eclipse platform.

Universal property   Intuitively, the “most efficient solution” to a certain problem. For instance, in the case of pushout, the calculated result is the least upper bound of the input objects.

XMI   The XML Metadata Interchange (XMI) is the OMG’s standard for exchanging metadata information via the Extensible Markup Language.