Using Concrete Syntax in Graph-based Model Transformations

Doctoral Dissertation by

Roy Grønmo

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfillment of the requirements for the degree ph.d. in Computer Science

November 2009
Abstract

The emergence of large and complex software systems increases the interest in model-driven engineering, as a way to lower the cost of development and maintenance of software. Models allow us to hide irrelevant details, provide different model viewpoints, and isolate and modularize models of cross-cutting concerns of a system. The emerging technologies for aspect-oriented modeling and weaving provide a systematic way to handle cross-cutting concerns at the modeling level.

The success of model-driven engineering relies heavily on model transformations. This thesis describes how aspect-oriented modeling and many typical model transformations can be defined as aspects/rules that: (1) use the concrete syntax of the involved modeling languages, and (2) use graph transformation principles as its foundation.

The thesis presents two main results. The first main result is an aspect language for UML 2 sequence diagrams. The language takes advantage of a formal model for sequence diagrams, which makes the matching and weaving process semantics-based. For this language we provide a confluence theory.

The second main result is an approach to define many typical model transformations as graph transformations, where the transformation designer uses the concrete syntax of the involved modeling languages. Some typical model transformation examples are illustrated in this thesis, i.e. transformation from feature models to BPMN, UML activity diagram aspects, UML activity diagram refactoring, UML state machine refactoring, transformation from sequence diagrams to state machines, and transformations involving Petri nets. A collection operator has been introduced as a means to match and transform collections of similar subgraphs in graph transformations, using either concrete or abstract syntax. This allows for improved usability in transformations that would otherwise be complex or impractical to specify.
Acknowledgements

The work reported in this thesis has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), SINTEF and the DiVA project grant no. 215412 (EU FP7 STREP).

The people in the SWAT group have been of important value for my progress throughout the time of the thesis work: Jon Oldevik, Fredrik Sørensen, Eyvind W. Axelsen, Øystein Haugen, Stein Krogdahl and Birger Møller-Pedersen. First of all I thank my main supervisor, Birger Møller-Pedersen, who has always taken the time, even on short notice, to read my work, give feedback and have discussion meetings about the work. His expertise in the field of model-driven engineering was crucial for the successful research results. My supervisor Stein Krogdahl has also participated in several discussions and his thoroughness, ability to see weaknesses and to provide detailed comments have strongly improved the quality of the results. Supervisor Øystein Haugens feedback has been of great value, especially regarding his expertise on model-driven engineering and UML sequence diagrams. I am very thankful to Fredrik Sørensen who has co-authored two of the papers, involved himself in our discussions and contributed to ideas and implementation of the semantics-based aspect language.

I really appreciate the efforts by Ragnhild Kobro Runde who has co-authored the work on the confluence theory. She learned me a lot about how to formalize a theory and to write formal proofs. She also found several weak points in the work and improved the overall structure of the paper.

In addition to those mentioned above I want to thank my working colleagues at SINTEF and at the Department of Informatics, University of Oslo, for providing good work environments.

I am very thankful to all the anonymous reviewers who pointed out flaws, possible improvements and suggested new directions in my work. This also goes for the feedback on the conference paper presentations. A special thanks to Claudia Ermel and Enrico Biermann for inviting me to the leading graph transformation community at the Technical University of Berlin for fruitful discussions about parts of the thesis work.

My lovely wife, Beate, has supported me throughout the whole thesis work for which I am deeply grateful. Also, my two sons, Jesper and Magnus, have given me a lot of joy and happy moments unrelated to the work.
List of original publications

1. Roy Grønmo and Birger Møller-Pedersen.  
   *Aspect Diagrams for UML Activity Models.*  
   In proceedings *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, Revised Selected and Invited Papers*, pages 329-344, Lecture Notes in Computer Science, Springer, 2008

2. Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, and Stein Krogdahl.  
   *A Semantics-based Aspect Language for Interactions with the Arbitrary Events Symbol.*  

3. Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, and Stein Krogdahl.  
   *Semantics-Based Weaving of UML Sequence Diagrams.*  

4. Roy Grønmo, Birger Møller-Pedersen, and Gøran K Olsen.  
   *Comparison of Three Model Transformation Languages.*  

5. Roy Grønmo, Stein Krogdahl and Birger Møller-Pedersen.  
   *A Collection Operator for Graph Transformation.*  

6. Roy Grønmo and Birger Møller-Pedersen.  
   *Concrete Syntax-based Graph Transformation.*  
   Research Report 389, Dept. of Informatics, Univ. of Oslo, Norway, 2009

7. Roy Grønmo, Ragnhild Kobro Runde, and Birger Møller-Pedersen.  
   *Confluence of Aspects for Sequence Diagrams.*  
   Research Report 390, Dept. of Informatics, Univ. of Oslo, Norway, 2009

8. Roy Grønmo and Birger Møller-Pedersen.  
   *From Sequence Diagrams to State Machines – with help from Combined Fragments.*  
   Research Report 391, Dept. of Informatics, Univ. of Oslo, Norway, 2009

The publications 1-8 are available as Appendices B-I in Part II.
Contents

Abstract iii

Acknowledgements v

List of original publications vii

Contents ix

I Overview 1

1 Introduction 3
   1.1 Overview of the Artefacts 7
   1.2 Structure of the Thesis 8

2 Research Method 9
   2.1 From Problem Analysis to Artefacts 9
   2.2 Innovation 10
   2.3 Evaluation 10

3 State of the Art 13
   3.1 Model and Graph Transformation Languages 13
   3.2 Sequence Diagrams and Aspect-orientation 17

4 Problem Analysis 21
   4.1 Problem Statement 21
   4.2 Requirements 26

5 Contributions 29
   5.1 A Collection Operator for Graph Transformation 29
   5.2 A Semantics-based Sequence Diagram Aspect Language 32
   5.3 A Theory for Confluence Analysis of Sequence Diagram Aspects 34
   5.4 A Framework for Concrete Syntax-based Graph Transformations 37

6 Discussion 41
   6.1 Design Decisions 41
   6.2 Evaluation of the Artefacts with Respect to the Requirements 43
   6.3 Related Work 45
Contents

7 Conclusions and Future Work
   7.1 Achievements ................................................................. 49
   7.2 Future Work ................................................................. 50

Bibliography ........................................................................... 51

II Research Papers ................................................................... 61
   A Overview of the Papers ....................................................... 63
   B Paper 1: Aspect Diagrams for UML Activity Models .......... 67
   C Paper 2: A Semantics-based Aspect Language for Interactions with the Arbitrary Events Symbol ......... 85
   D Paper 3: Semantics-Based Weaving of UML Sequence Diagrams ....................................................... 103
   E Paper 4: Comparison of Three Model Transformation Languages ..................................................... 121
   F Paper 5: A Collection Operator for Graph Transformation ............................................................. 139
   G Paper 6: Concrete Syntax-based Graph Transformation ............................................................... 157
   H Paper 7: Confluence of Aspects for Sequence Diagrams .................................................................. 181
   I Paper 8: From Sequence Diagrams to State Machines
               – with help from Combined Fragments ............................................................. 247
Part I

Overview
Chapter 1

Introduction

Model transformation can be defined as a process that produces a target model from a source model. This simple definition can be generalized by allowing multiple source models and multiple target models. The source and target modeling languages may differ or be the same. If they are the same, the target model can replace the source model, which means that the transformation can be regarded as a manipulation of the source model.

Model transformations are crucial for the success of model-driven engineering. Relevant examples include transformations between models of the same reality seen from different viewpoints (e.g. UML sequence diagrams to UML state machines [93]), transformations from an abstract model to a more detailed model (e.g. UML class model to WSDL [31]), and model refactoring (e.g. transformation of workflow graphs [44]). In model-driven engineering the model transformations partially automate the transitions from one model to another, which otherwise would be fully manual work. By replacing the manual work by automation, the software engineering process becomes less error-prone and more efficient. A target model will often be subject for further manual refinement. Additional information is added compared to the information that can be derived from the source model.

A major motivation for our thesis work is to improve the model transformation languages with respect to the user-friendliness of specifying transformations. The current approaches (ATL [45], QVT [70] KerMeta [67], Epsilon [55]) provide both imperative and declarative specification styles, except KerMeta which is purely imperative. These approaches have several weaknesses. Firstly, they lack a formal foundation that enables us to discuss termination and confluence properties of declaratively specified transformations. Secondly, they are textual-based even for cases where the source and target are graphical models. Thirdly, the transformation designer needs a detailed knowledge of the often complex metamodels of the source and target languages to be able to specify a transformation.

Models often resemble graphs. This is why graph transformation has been promoted by several authors as a means to specify model transformations (four examples are illustrated in [96]).

A graph consists of nodes and directed edges that connect the nodes. For typed attributed graphs [37], nodes and edges have types. These types, at least for nodes, often include a set of named attributes with values that can vary from one node to another. The abstract syntax of a graph visualizes all nodes by similar graphical symbols, and all edges by similar graphical symbols. A common abstract syntax for graphs is to let a node be represented by a rectangle separated into two compartments. The first compartment denotes an instance identifier and node type, while the second compartment contains the list of attributes and their values. An edge is normally visualized with an arrow, where the edge type is placed next to the arrow.
Graph transformations have a formal foundation and an established theory with tool support. However, one of the weaknesses described above for model transformations also apply to graph transformations. Graph transformations are normally also specified in relation to the source and target metamodels, and hence the transformation designer also here needs detailed knowledge of these metamodels.

A metamodel defines concepts and relationships between these concepts, which constitute the language in which models are made. Graphs and models can share the same conceptual metamodel, although in practice they are normally represented differently since graph and modeling tools do not have a united way to represent a metamodel. In both cases, a metamodel is defined by a structural class-like diagram. For graphs, the metamodel is often referred to as 'the type graph'. Such a metamodel implicitly defines the abstract syntax without additional information necessary. On the other hand, the concrete syntax provides additional information that cannot be derived from a metamodel. The concrete syntax defines the visual representation for each type in the metamodel. The same model can be represented by abstract syntax or by concrete syntax.

A graph transformation rule is often displayed with a left hand side graph (LHS), a right hand side graph (RHS), and a number of negative application condition graphs (NACs). The LHS defines a subgraph to be matched, usually also with respect to types or attributes, within the graph to be transformed. A matched LHS within the source graph is replaced by the corresponding RHS. Figure 1.1 illustrates this principle. An interface graph specifies the shared elements between the LHS and the RHS, and this defines how the new elements in the RHS shall be connected to the remaining graph which is unchanged by the rule. The interface graph is usually not explicitly given, but is instead implicitly defined by the instance identifiers of nodes and edges. An element implicitly belongs to the interface graph if it has the same identifier in the LHS and the RHS. A NAC prevents application of the corresponding rule if the LHS combined with the NAC has a match. There can be multiple NACs associated with one rule. Further details and formalization of these concepts can be found in paper 5.

![Graph transformation rule](image)

Figure 1.1: Graph transformation rule (target graph = source graph, except for the indicated replacement)

Graph-based model transformation usually follows the approach shown in Figure 1.2, where a set of graph transformation rules have been specified based on the abstract syntax of graphs. First, the source model (in concrete syntax) is mapped to a source graph (abstract syntax). Then, the source graph is transformed by a graph transformation tool, according to the graph transformation rules, into a target graph. Finally, the target graph is mapped to a
model in concrete syntax of the target modeling language.

![Graph-based model transformation](image)

Figure 1.2: Graph-based model transformation

In our thesis work we have chosen to follow this promising research path of graph-based model transformations. However, our special approach is that we have investigated if graph-based model transformation rules can be specified directly in the concrete syntax of the source and target languages, instead of defining traditional abstract syntax-based rules. For languages with well-known concrete syntaxes, this has the potential to make the specification of model transformations more user-friendly since the transformation designer needs no detailed knowledge of the metamodels.

The idea of using concrete syntax in graph-based model transformations started with investigations of how to specify changes to models in an aspect-oriented way and how the associated weaving process should work. Aspect-orientation was originally defined for traditional programming, as a means to take care of ‘cross-cutting concerns’ in a modularized way. Cross-cutting code is specified in a separate module, called an aspect. An aspect contains a pointcut that specifies where the aspect code (also part of an aspect, and called the advice) shall be woven into the rest of the program code. Without aspect-orientation, cross-cutting code would traditionally have to be scattered and duplicated into multiple areas of the code.

Aspect-oriented modeling aims to do the same for the modeling domain. In a model weaving approach the aspects define cross-cutting concerns that can be woven with the main model. Aspect-oriented modeling and weaving can be seen as a special case of model transformations in which the source and target languages are the same. The aspect-oriented model corresponds to a set of transformation rules and the weaving corresponds to applying the transformation rules.

For aspect-oriented modeling and weaving it is common to use concrete syntax-based aspects, and in some of these approaches aspects are specified in a way which is quite similar to graph transformation rules [51, 101, 50].

To develop and validate the approach of using concrete syntax in graph-based model transformations, we have applied it on examples involving different diagram types, especially UML 2 diagram types. UML 2 sequence diagrams have been given special attention in our work, since they are non-trivial to handle in graph-based transformations. The challenges are due to the facts that: (1) the abstract syntax for a sequence diagram has a relatively complicated structure even for small sequence diagrams, and (2) there is a significant order of the events on a lifeline, while graphs have no order on a node’s incident edges. It turns
out that the latter fact has the consequence that concrete syntax-based graph transformation rules for sequence diagrams need special treatment.

Figure 1.3 illustrate the difference in concrete and abstract syntax for a simple sequence diagram. The left hand side of the figure shows the concrete syntax, and the right hand side of the figure shows a choice of corresponding abstract syntax. For larger sequence diagrams, especially those involving combined fragments (introduced in Chapter 3), the abstract syntax becomes quite complicated. The example sequence diagram has two lifelines, L1 and L2, and there are two messages a and b both going in the same direction from lifeline L1 to L2.

![Concrete vs. abstract syntax](image)

Figure 1.3: A sequence diagram in concrete syntax and a corresponding abstract syntax

Normally, the matching in graph-based model transformation is syntax-based\(^1\), and does not take the semantics of the source language into account. A syntax-based matching can potentially fail to match semantically equivalent, but syntactically different structures to a rule’s LHS. This limitation is why we have developed a semantics-based aspect language for sequence diagrams. In order to be semantics-based, this language uses a different matching strategy than for graph transformations. The matching relates to the semantics of a sequence diagram that can be defined as traces representing valid or invalid executions.

The commonalities of our sequence diagram aspect language and the other contributions of our thesis work are: (1) the use of concrete syntax in model transformations, and (2) aspects and transformation rules that are specified in a similar manner to graph transformation rules. However, the terminology is different. An aspect corresponds to a rule, a pointcut corresponds to a LHS, an advice corresponds to a RHS, and negative pointcuts correspond to NACs.

By regarding aspect-oriented modeling and weaving as a special case of model transformation, we can have a unified conceptual view where both fields can learn from the experiences and best practices of the other. Furthermore, we use graph transformation as a basic foundation for both these fields, while for each modeling language a tailoring and specialization is needed in order to make the approach as useful as possible.

Our overall goal is to improve model transformation engineering (where model transformation includes aspect-oriented modeling and weaving) by: (1) making model transformation languages more user-friendly, (2) finding ways to ensure the correctness of a transformation specification, and (3) ensuring termination and confluence of transformation specifications.

This thesis introduces four artefacts which contribute to the overall goal, mostly with respect to goals (1) and (3). We contribute to goal (1) by increasing the expressiveness of graph transformation with a collection operator, by making a semantics-based sequence

\(^1\)Here, syntax-based can refer to either concrete syntax or abstract syntax depending on what the graph transformation rules are based upon.
diagram aspect language, and by using concrete syntax-based graph transformation rules. We contribute to goal (3) by a theory that can be used to analyze if a set of sequence diagram aspects is confluent. An overview of the artefacts is provided below.

### 1.1 Overview of the Artefacts

The first subsection introduces a collection operator. The second subsection introduces a sequence diagram aspect language. The third subsection introduces a confluence theory for sequence diagram aspects. The fourth and last subsection introduces a framework for concrete syntax-based graph transformation rules.

#### 1.1.1 A Collection Operator for Graph Transformation

The artefact is a graphical construct for graph transformation rules, called **collection operator**, to match and transform collections of similar subgraphs. The matching and transformation is defined in relation to algebraic graph transformation, by dynamically instantiating a collection free rule according to the actual match size. This approach allows us to reuse much of the existing graph transformation apparatus. The collection operator is novel since it allows us to specify collection matching and transformation concisely by a single rule, and to allow general cardinalities of potential matches. The collection operator is useful in several graph transformation cases, and these cases are cumbersome to express without an operator like the collection operator.

#### 1.1.2 A Semantics-based Sequence Diagram Aspect Language

We define an aspect language that can be used to specify cross-cutting effects on a set of UML 2 sequence diagrams. The aspects and sequence diagrams are woven at the model level. By basing the weaving upon a formal trace semantics for sequence diagrams, we ensure that the weaving is semantics-based. We formally define the concepts of matching and weaving, and we also prove that the weaving, under a few reasonable conditions, does not lead to invalid sequence diagrams. Our **arbitrary events symbol** is a wildcard mechanism that can be placed on the pointcut lifelines. The symbol implies that zero or more events are allowed in the symbol position on a matching sequence diagram lifeline.

#### 1.1.3 A Theory for Confluence Analysis of Sequence Diagram Aspects

This artefact provides a confluence theory for sequence diagram aspects with respect to the expressiveness of the language in which aspects are specified. We show that confluence is undecidable for a set of aspects with negative pointcuts and the arbitrary events symbol. On the contrary, we show that confluence can be algorithmically checked for aspects without negative pointcuts and the arbitrary events symbol. The decidability algorithm is based on an extended version of a traditional critical pair analysis from term rewriting and graph transformation.
1.1.4 A Framework for Concrete Syntax-based Graph Transformations

We present the framework of a general purpose model-to-model transformation language, where the transformation modeler can concentrate on the intuitive concrete syntaxes of the source and target modeling languages. The approach has been tested on a number of modeling languages and we report the major findings of these case studies.

1.2 Structure of the Thesis

This thesis is based on 8 papers, which are attached as Part II. Part I describes a unified overview of the work:

- **Chapter 1 - Introduction** presents a short introduction, our four artefacts and the structure of the thesis.

- **Chapter 2 - Research Method** describes an iterative method over three steps (problem analysis, innovation, evaluation) upon which the research of this thesis is based.

- **Chapter 3 - State of the Art** describes the foundation for our work in this thesis. We give an overview of the most relevant literature for further reading on preliminaries to our work. Some of the work cited have chosen different approaches than we have, but they were candidate approaches before we chose our path. We highlight the shortcomings in state of the art that are addressed by this thesis.

- **Chapter 4 - Problem Analysis** describes the main problems and challenges addressed that are addressed by this thesis. The chapter also presents a list of requirements to possible artefacts that address the described problems.

- **Chapter 5 - Contributions** describes an overview of our main achievements within this thesis. The achievements consist of four artefacts.

- **Chapter 6 - Discussion** discusses some of our design decisions with respect to the artefacts, and it is discussed to what extent our developed artefacts fulfill the requirements introduced in Chapter 4. This chapter also describes the closest of the related works. It is described in more detail than any of the papers previously have.

- **Chapter 7 - Conclusions and Future Work** presents the main achievements of the thesis, and directions for future work are suggested.

The first chapter in Part II gives an overview of the papers by giving a short abstract, an identification of my contribution, and the publication status for each paper.
Chapter 2

Research Method

This chapter presents the research method that has been used in the thesis work.

As clarified by [90], much of the computer science research, including this thesis work, can be called technological research, where the aim is to create new or improved artefacts. In the field discussed here, such artefacts include languages, modeling constructs, security protocols, hardware processors, methods and theory.

The research method normally used for technological research is an iterative process, where each iteration consists of three steps: problem analysis, innovation, and evaluation. In the problem analysis, we choose particular topics within our field of interest that we will investigate. The investigation leads to the identification of certain needs, and we set up more concrete requirements for the technology. We continue by investigating the state of the art to see if there are shortcomings in all the current approaches with respect to the requirements. Next, we move on to the innovation step, in which we propose new constructs, hypotheses, implementations etc. that can improve the state of the art so that the requirements are better fulfilled. The evaluation checks to which extent the new artefacts fulfill the requirements. Such checks can be performed by e.g. experimentation, user evaluation, test examples, or formal proofs.

The next three sections explain how the three steps are instantiated in this thesis.

2.1 From Problem Analysis to Artefacts

In the initial work of this thesis, we investigated state of the art as described in Chapter 3. This state of the art investigation covered model and graph transformation approaches, and aspect-oriented modeling with different weaving approaches, the latter in particular related to UML sequence diagrams.

We tested graph transformation on a number of model transformation scenarios and discovered the lack of a construct to match collections of similar subgraphs, which led to our first artefact.

We reviewed the existing sequence diagram aspect languages. Since all these aspect languages had shortcomings, we designed a new language as our second artefact.

A confluence theory specialized for sequence diagram aspects is, to our knowledge, not addressed in any previous work. The existing confluence theories from string rewriting or graph transformation were not directly applicable to sequence diagram aspects, and hence we provide a new theory as our third artefact.

Most of the existing model transformation approaches use abstract syntax. In the few
approaches that use concrete syntax there were several open questions and challenges that were not addressed or discussed, including the flexibility to switch between concrete and abstract syntax, and especially handling sequence diagrams properly. These limitations are partly addressed by our fourth artefact.

### 2.2 Innovation

Our innovation is the introduction of the four artefacts that were shortly described in Chapter 1. We here shortly explain the innovation step in relation to these artefacts.

We have introduced a collection operator for graph transformation (artefact 1) that makes it relatively easy to define transformations that are cumbersome to define without a collection operator.

The sequence diagram aspect language (artefact 2) has been developed by adopting many strong principles and design decisions from other languages, and extending the language based on test examples and a case study presented in paper 2.

We explored how the expressiveness of the aspect language effected the confluence theory (artefact 3) from two sides: (1) we tried to minimize the expressiveness of the aspect language and see how long confluence remained undecidable, and (2) we tried to maximize the expressiveness of the aspect language and see how long we still could find an algorithmic way to check confluence.

An early hypothesis in the thesis work was: 'Graph transformation is suitable to solve aspect-oriented modeling and weaving'. The outcome of this was concrete syntax-based graph transformation rules as a means to define aspect-oriented models. The rules were restricted to endogenous transformations, i.e. transformations where the source and target languages are the same. After successful results, a new generalized hypothesis emerged: 'Concrete syntax-based graph transformation is suitable to solve more general model transformations than endogenous transformations' (artefact 4). The goal is to allow as many source and target languages as possible, and to explore when the approach is not applicable. The thesis has only some initial results in answering that question.

In Section 6.2 we explain that our artefacts fulfill many of the requirements identified in Section 4.2. Further research is needed to find solutions (if they exist) that fulfill all the requirements, and also to identify new requirements based on case studies and practical usage.

### 2.3 Evaluation

Evaluation has been performed by example scenarios, tool implementation and formal proofs. Example scenarios range from full scale case studies (paper 2 and 4) to small examples, mostly taken from the research literature.

The first artefact, the collection operator, has been applied in several examples including UML activity model refactoring examples (paper 1 and paper 4), simulation of the firing of a transition in Petri nets (paper 5), UML state machine refactoring (paper 5), transformation from UML sequence diagrams to UML state machines (paper 8), and transformation from feature models to business process models (paper 6).

The second artefact, the aspect language, has been applied on a sequence diagram aspect example in paper 2. Some important properties of the aspect language have been formally
proven in paper 7. This includes the soundness of weaving that ensures that the woven diagrams are valid. Finally, large parts of the aspect language and a weaving tool have been implemented and tested on numerous examples.

The third artefact, the confluence theory, has been validated by formal proofs of several lemmas and theorems including an undecidability theorem and a critical pair theorem to check confluence.

The fourth artefact has been applied in several examples in papers 1, 4, 5, 6 and 8. In paper 4, a transformation case study shows several benefits (including conciseness and less effort needed) of the fourth artefact compared to two of the leading model transformation languages ATL [45] and AGG [95] for a particular transformation task. However, future work to conduct a user evaluation would complement our work.
Chapter 3

State of the Art

In this chapter we describe the state of the art that constitutes the foundation for our thesis work. Furthermore, we point out some limitations and shortcomings that we have addressed in the thesis. However, we refer to Section 6.3 for the most closely related works and work carried out in the same time interval as our work. The related work sections in the attached papers are also complementing this chapter.

This chapter is organized in two sections, where the first describes state of the art of model and graph transformation languages, and the second describes state of the art of aspect-oriented approaches related to sequence diagrams.

3.1 Model and Graph Transformation Languages

Czarnecki and Helsen [13] classify the nature of model transformation languages. Along one axis, a language is either declarative, imperative or a mix of declarative and imperative (called hybrid). Along another axis, the language can offer ways to specify bidirectional or unidirectional transformations. If the source and target languages are the same, then the transformation is called endogenous, otherwise it is called exogenous.

A number of model transformation languages (ATL [45], QVT operational [70] KerMeta [67], Epsilon [55]) all have the following characteristics: (1) a transformation is specified in relation to the metamodels of the source and target languages, (2) the transformation language has a textual syntax, (3) the transformation language is either imperative or a hybrid of imperative and declarative, and (4) the transformation language rely strongly on OCL-like constructs [69].

The user-friendliness of OCL-based transformation languages can be questioned since Stein et al. [91] conclude that OCL 'quickly leads to complex query statements even for simple queries'. Another case study where OCL-expressions are used to specify a complex model refactoring for business process models, illustrates that OCL-expressions to specify transformations also can get very complicated [54]. Stein et al. propose a graphical UML-based notation for matching UML model extracts instead of OCL, called Join Point Designation Diagrams (JPDD). The work is restricted to matching and does not cover the transformational part.

Many of the model transformation languages have a very general applicability since they can be applied to any modeling language that can be defined by a MOF metamodel [71]. However, the model transformation languages mentioned above have not been formalized and used in a theory that provides ways to analyze confluence and termination properties.
Furthermore, the transformation specifier needs detailed knowledge about the involved metamodels. Even for cases when the source and target models have graphical concrete syntaxes, it is often necessary to specify the transformation in pure textual code.

Models can be represented by graphs, and a metamodel can be defined by a type graph in a notation similar to UML class diagrams, which is analogous to metamodel definitions in model transformation. Graph transformation (GT) can be used to specify model transformations, and GT is based on a mathematically precise formalism in contrast to many of the model transformation approaches.

**Algebraic GT**

A main branch within GT is *Algebraic GT* [95, 25, 22, 57]. Algebraic GT is based on category theory and rules are declaratively defined. There are two main approaches of algebraic GT: double pushout and single pushout. In the double pushout approach, a transformation rule is not applied on a match when it leads to dangling edges in the resulting graph. This is a safe way to define the transformation, since no deletion is carried out without explicitly stating so. In single pushout all potentially dangling edges will be deleted implicitly (e.g. the GROOVE tool [48]).

Algebraic GT first used the plain graph concept, where a graph can be defined by a set of typed nodes and typed edges that connect the nodes. This graph concept has been extended by many works including [20, 37, 25, 15]. In the work by Heckel et al. [37], nodes can have attributes, and in the work by Ehrig et al. [25], the notion of E-Graph allows also edges to have attributes. In [15], de Lara et al. introduce node inheritance. Hyperedges from Drewes et al. [20] allow a single edge to connect an arbitrary number of nodes.

An algebraic GT rule is basically defined by three main graphs: a LHS graph, a RHS graph, and an interface graph (I) to show the shared elements between the LHS and the RHS. The elements in the interface graph are to be preserved, the elements in LHS \ I are to be deleted, and the elements in RHS \ I are to be added. Thus, the interface graph defines how the RHS graph shall be connected to the remaining graph.

There are three notation styles to represent an algebraic GT rule: (1) the three graphs mentioned above, (2) two graphs (LHS and RHS), or (3) a single graph. The first notation style directly corresponds to the theoretical foundation. It can be found in several research papers, e.g. [78], but is rare in tool implementations. In the second notation style, shared identifiers, between the LHS and the RHS, implicitly define the interface graph (e.g. [95]). The third notation style uses some kind of marking (e.g. tags, colors) to denote the elements to be added and deleted respectively (e.g. [48, 101]). Untagged/uncolored elements represent elements that are neither added nor deleted, but they are elements that need to be matched.

The three notational styles are in general equally expressive and the choice is a matter of taste. It should also be feasible to make automatic conversion between these three different alternatives so that the transformation designer can have access to all three notation styles.

Regardless of the notational style, a transformation rule can have an arbitrary number of negative application condition graphs (NACs) [34]. If a rule matches any of its NACs, then the rule cannot be applied.

**Triple Graph Grammars**

Another branch within GT is *Triple Graph Grammars (TGG)* [87], where the transformation rules are always bidirectional. A correspondence graph holds the relationship between the
source and target graphs. All the rules have a clear separation between source, target and correspondence elements.

AToM [16] is a tool that supports TGG in addition to algebraic graph transformation. AToM rules are partly graphical for the main nodes and relations, while it uses textual pre- and post-conditions to define attribute conditions, e.g. a transformation from UML sequence diagrams to state charts [93].

OMG’s Query Views and Transformations (QVT) [70] provides QVT relational as a graphical transformation language in addition to the textual language (QVT operational), already referenced in the beginning of this section. QVT relational is based on the principles of triple graph grammars discussed above, and hence only bidirectional transformations can be expressed. Abstract syntax in the rules is used to represent metamodel concepts in the two modeling domains that take part in a relation. A keyword not, that can only be assigned to single elements (a restricted variant of a NAC), specifies that this element cannot exist in order to apply the transformation. OCL expressions are used in QVT relational to express conditions for when a transformation is applicable.

**Controlling the GT rule execution order**

In the algebraic GT the set of rules shall by default be applied non-deterministically. The notion of layers [80, 11] has been added to algebraic GT in the AGG tool to give some control over the rule execution order. First, all the rules in layer 0 are applied non-deterministically until no more rules in this layer are applicable. Then, we follow the same principle with layer 1, layer 2 and so on. The rule designer may then choose to loop over the layers or not. If layers are not sufficient, then the only alternative within the AGG tool is to write Java code to handle the rule execution order.

In VIATRA [99], MOLA [47, 46], GReAT [5], Fujaba [29] and PROGRES [86, 88], the principles of algebraic graph transformation are combined with advanced mechanisms for controlling the rule execution order. These tools follow a hybrid style, where the rules are declarative and the rule execution ordering statements are imperative. While VIATRA has a textual syntax for the rule ordering, Fujaba, MOLA and PROGRES provide a graphical syntax which is quite close to UML activity diagrams. This includes control flow of the following types: sequence, conditions, parallel and loop. GREaT uses a data-flow graph instead of a control flow graph. VIATRA rules have rule parameters for nodes and edges, which are intended for better reusability.

Bottoni et al. [10] allow to define replacement units as a way to define transactional support for the execution of algebraic GT rules. The transaction contains rules and a rule execution order defined textually to be a mix of sequential and as-long-as-possible control flow, where as-long-as-possible means a loop over a set of rules until no more rules are applicable. Bottoni et al. provide sufficient criteria to state that such replacement units terminate.

**Confluence and termination in GT**

This subsection describes some main results regarding the desired functional properties of a graph transformation system: confluence and termination. We will focus on GT systems, but a few results from term rewriting systems are also mentioned since much of the theory for graph transformation systems has been inspired by it.

Plump [78] uses the term direct derivation to denote the application of a single graph transformation rule on a single match. A derivation is a sequence of zero or more direct
derivations. A *normal form* is a graph on which there are no possible direct derivations. Two graphs are *joinable* if there exists derivations leading to a common (up to isomorphism) graph.

The concepts above can be used to define confluence and termination. A set of graph transformation rules is *terminating* if and only if there exists no infinite derivation sequence for any graph. A set of graph transformation rules is *confluent* (globally confluent) if and only if all derivations from the same graph are joinable. It is important to note that this shall hold for any graph. A set of graph transformation rules is *locally confluent* if and only if all direct derivations from the same graph are joinable. Newman’s Lemma [64] proves that local confluence and confluence are equivalent for terminating systems.

There are several strong theoretical results available for algebraic GT, including termination theory [77, 56, 58, 76, 9] and confluence theory [37, 78, 56]. AGG [95] is a tool that supports algebraic graph transformation, and to our best knowledge it is the only graph transformation tool that provides a termination and confluence analysis.

For term rewriting systems, all critical pairs are joinable if and only if there is confluence [53]. For GT on the other hand, Plump [78] has shown that confluence is undecidable for terminating GT systems. A few authors have identified sufficient criteria for GT confluence. Confluence holds if all critical pairs are *strongly joinable* [78, 37], where strong joinability is joinability and the additional requirement that all nodes preserved in the two derivations of the critical pair, must be maintained and be isomorphic in the joined result.

Critical pairs are systematically produced from each pair of transformation rules according to possible dependencies between two rules. For GT there are two cases of dependence as defined by Lambers et al. [57]: *use-delete* conflict and *produce-forbid* conflict. A use-delete conflict occurs when one rule deletes something in the LHS of the other rule. A produce-forbid conflict occurs when one rule produces something that is matched by a negative application condition in the LHS of the other rule.

Huet and Lankford [40] has proven that it is undecidable if a term rewriting system terminates, while Plump [77, 76] has proven that it is undecidable if a GT system terminates. Even though termination is undecidable, several authors have defined sufficient criteria to claim confluence for GT systems [23, 1, 10, 9]. For certain restricted versions of term rewriting, termination is decidable [17, 30, 68].

**Extensions of the GT expressiveness**

Some GT tools, like AGG, allow us to use E-Graphs and to associate an icon or graphical shape to each node type. This can make the visualization of the rules appear closer to the concrete syntax, than of pure abstract syntax of the source and target languages. However, there is still a gap from having rules defined fully upon the concrete syntax.

In order to make GT more user-friendly there have been several proposals to extend the basic expressiveness by high-level constructs [60, 33]. These constructs allow us to specify powerful match expressions and associated transformations in a concise manner. The *star operator* [60] allows us to match patterns of arbitrary depth, and *recursive rules* [33] are introduced to handle transformation tasks of a recursive nature.

Fujaba, QVT relational and PROGRES have support for matching collections of single nodes. The nodes are displayed as a multirectangle symbol, and the incident edges are implicitly defined to occur multiple times in rule matches. A match will contain the same number of an incident edge as there are occurrences of the node in the matched collection. Each item in a collection is a single node only, and there is no support for matching collec-
3.2 Sequence Diagrams and Aspect-orientation

A group operator, introduced by Balasubramanian et al. [4] and implemented in the GREaT tool, enables arbitrarily large subgraph matches that can be copied, moved or deleted. However, the subgraph matches cannot be modified.

Amalgamated rules described by Taentzer [94], can simulate subgraph matching and transformation. This cannot be specified by a single rule diagram. Instead there will be one subrule to capture the rule part outside of all subgraphs, and one elementary rule for each subgraph to be matched and transformed.

3.2 Sequence Diagrams and Aspect-orientation

Figure 3.1a shows a UML 2 sequence diagram [69] with two lifelines L1 and L2, and two messages with the signals a and b. A lifeline, visualized with a rectangle and a dashed line below, represents an interacting entity on which events take place in an order from top to bottom on the dashed line.

Each message is represented by two events, a send event (!) and a receive event (?). Thus, our example diagram has four events, !a and !b on lifeline L1, and ?a and ?b on lifeline L2.

Sequence diagrams impose a partial order of events given by: (1) the send event must come before the receive event of the same message (this is referred to as the message invariant), and (2) all events are ordered from top to bottom on each lifeline. An intuitive idea behind this partial order is that messages are sent asynchronously and that they may happen in any order on different lifelines, but sequentially on the same lifeline. Figure 3.1b shows the four partial order requirements of the sequence diagram.

STAIRS [85] gives the semantics of a sequence diagram using traces that represent possible execution runs. More precisely, the semantics of a sequence diagram can be described as a set of positive traces and a set of negative traces. Positive traces define valid behavior and negative traces define invalid behavior, while all other traces are defined as inconclusive. Our example diagram in Figure 3.1a has no negative traces and two positive traces as shown in Figure 3.1c.

While STAIRS provide a denotational semantics for sequence diagrams, there are also works that present operational semantics for sequence diagrams. Klein et al. [52] and Grosu and Smolka [32] use automata-based approaches, while Lund [61] uses rewrite rules. Although these approaches are different, they seem to provide similar semantic interpretations of sequence diagrams.

In UML version 2, combined fragments were introduced to model optional behavior (opt), conditional behavior (alt), and loops (loop). The combined fragments can have
guard expressions and they can be arbitrarily nested. Figure 3.2 illustrates usage of combined fragments to model the behavior of the yatzi game. A player sends a rollDice message to the YatzySystem. The outer loop iterates once for each dice to roll, which is ensured by the guard more dice. The body of the loop contains an alt operator with six operands, one for each alternative outcome of a die. A combined fragment spans across the involved lifelines, and its the operands are separated by a dashed line. A loop operator has always exactly one operand.

Before we present aspect-orientation for sequence diagrams, we shortly introduce the main concepts of aspect-orientation. The most successful aspect-oriented language to date is AspectJ [49], which works on Java programs. AspectJ defines an explicit joinpoint model for where AspectJ cross-cutting concerns can be inserted. An aspect consists of a pointcut and an advice. The pointcut defines a pattern to be matched in the base program, and the matches must be a subset of all possible joinpoints. The advice defines changes or additions in the match positions of the base program. AspectJ further defines the possible actions to be taken: before, after, and around. New code is then inserted before, after or around the matched code.

Many aspect-oriented approaches follow the AspectJ way of aspect-orientation, which in some cases may be too limited, at least for defining aspect-oriented models. Firstly, the AspectJ joinpoints are all atomic, i.e. a joinpoint consists only of a single event, as opposed to a single joinpoint consisting of a series of events. A method call and a variable assignment are examples of atomic joinpoints. Secondly, there is no notion of state in the aspects to control when an aspect matches a pointcut. This is why Douence et al. [19] have introduced stateful aspects. Thirdly, the explicit joinpoint model only allows to match predefined places in a program and not unforeseen places. Fourthly, AspectJ has no notion of distribution or parallel behaviors. The latter limitation has been addressed in [97].

Several diagram types have recently been targeted by aspect-oriented modeling approaches, where it is common practice to specify the aspects based on the concrete syntax of the targeted diagram type [50, 27, 101, 42, 43, 12, 18, 51, 89, 72]. These address mostly UML diagram types including structural diagrams of classes and components, and behavioral diagrams of sequence diagrams and state machines.
The aspect-oriented modeling approaches can be classified as symmetric and asymmetric. In the symmetric approaches, there is no difference between an aspect and a base. The notion of a base is irrelevant and everything is called aspects or concerns, which are woven together.

In the asymmetric approaches, there is a base and a set of aspects to be woven upon the base. For behavioral diagrams, asymmetric aspect approaches have dominated, while for structural diagrams both symmetric and asymmetric aspect approaches are common. France et al. [27] represent a symmetric aspect approach for UML classes.

A single aspect can even be specified by several diagram types as proposed by Kienzle et al. [50]. There, one aspect consists of three UML diagram views: class, state machine and sequence diagram each consisting of pointcut and advice. The maintenance of consistency between the different views is an important part of the framework.

The MATA approach by Whittle et al. [101, 42, 43] follows a GT-based approach to aspect-orientation. MATA supports aspect-oriented modeling and weaving for UML sequence diagrams, state machines and class diagrams.

The MATA approach is promising for a number of reasons. Firstly, any model pattern can be a joinpoint and therefore there is no need to define an explicit joinpoint model. Explicit definitions of joinpoints may be too restrictive for unexplored aspect cases. Secondly, the matching and weaving use well-founded principles from GT. Thirdly, it allows for the analysis of termination and confluence properties of aspects. In addition, the MATA approach uses concrete syntax-based aspect definitions as a layer on top of the abstract syntax-based GT rules.

However, compared to our aims there are some shortcomings in the MATA work. The aspect language is syntax-based, and the described work provide no formalization of the match concept. This makes it impossible in general to decide if a base model has a match and to determine certain other properties of the language.

There have been a number of proposals on aspect diagrams for UML 2 sequence diagrams (abbreviated as aspect diagrams) [12, 18, 51, 89, 101]. Aspect diagrams at the model level define cross-cutting effects on a base model. Some of the proposals pursue a model weaving approach [51, 89, 101], while others intend to postpone the weaving to the program level [12, 18].

In most of the sequence diagram aspect proposals we have seen, the aspect definitions are based on the concrete syntax of sequence diagrams. On the other hand, most of the more generic model and graph transformations listed in the previous section, restricts us to use abstract syntax in their transformation rules.

Mehner et al. [63] analyzes if a set of aspects may be properly woven with the base model by considering possible conflicts and dependencies. Pre- and post-conditions expresses the effects of each activity in the AGG tool where automated analysis is carried out. The aspect definitions proposed in their paper are limited to inserting new behavior either before, after or as a replacement of some previous activity.

Avgustinov et al. [2] perform a run-time matching and weaving as opposed to static weaving. The matching is based on traces and since this happens during run-time, the aspects are restricted to additive parts that are inserted entirely after the already executed match part. Also with run-time weaving, performance becomes a major issue.

Join Point Designation Diagrams (JPDD) [92] allows us to express pointcuts by using sequence diagrams, state machines and class diagrams. Such pointcuts can then be mapped to AspectJ pointcuts. The approach does not cover the advice part of an aspect, and JPDD focuses on the synchronous communication.
Deubler et al. [18], Solberg et al. [89], and Jayaraman et al. [42] all define syntax-based approaches for sequence diagrams. Deubler et al. match single events only and provide no model weaving or mapping to a concrete aspect language. Solberg et al. either use special tags on the base elements that shall be woven with an aspect, or use a separate binding model to express this.
Chapter 4

Problem Analysis

This chapter describes the main problems and challenges that the thesis addresses. The first section presents an overall problem statement and the last section presents the requirements for what we consider as successful outcome of the different problems. The requirements have emerged from concrete examples, have been adopted from related work, have been identified based on shortcomings and limitations in review of existing work, and have emerged when trying to extend and generalize our initial results.

4.1 Problem Statement

The problem statement is divided in two subsections, one for sequence diagram aspects and the other for graph transformation.

4.1.1 Sequence diagram aspects

Aspect-oriented modeling and weaving help to modularize, isolate and better maintain cross-cutting concerns at the model level. This has the potential of improving model-driven development. The research on aspect-oriented modeling and weaving at the model level is still in its infancy and more research is needed to clarify how to optimally utilize this approach.

As explained in Chapter 3, there have been a number of proposals on aspect diagrams for UML 2 sequence diagrams [12, 18, 51, 89, 101]. Figure 4.1 shows an example of such an aspect and how it can be woven with a base model. The aspect pointcut consists of an a message followed by a b message, both going in the same direction from the lifeline of type L1 to the lifeline of type L2. The normal interpretation is that there is a match of this pointcut within the figure’s base model. These two matched messages in the base model will then be replaced by the advice of the aspect, which in this case means that the two messages a and b are kept, while a c message is introduced in between them.

The weaving process may vary between different approaches. One strategy is to define a set of aspects and to non-deterministically apply one aspect at a time until no more aspects are applicable, i.e. there are no more matches of any of the aspects. Klein et al. and Whittle et al. [51, 101] argue that there are situations where we should allow matches when there are unspecified events in between the events of the explicit pointcut events. By allowing such unspecified events, the woven model in Figure 4.1 has another match where the events of the already woven c message plays the role of such unspecified events, and the weaving process continues. In fact, the weaving process will in this case never terminate since a pointcut
match is part of the advice. Obviously non-termination is an undesired property. Thus, we should either disallow such aspects, or find another way to ensure termination in such cases.

Another weaving strategy, chosen by Klein et al. [52, 51], identifies all the matches in a base model, weaves in all these positions and terminates the weaving process. Such a weaving process will always terminate, and the woven model in Figure 4.1 cannot be woven anymore if the shown aspect is the only aspect.

Let us assume that our aspects are guaranteed to terminate, and that the matching allows for unspecified events between the explicitly defined events of the pointcut. Such matching of unspecified pointcut events leads to some questions that are addressed by the thesis. Where should messages introduced by the advice be placed in relation to these unspecified events? Can matched unspecified events be deleted? Can unspecified events be allowed in certain positions and not in others? For our example this could mean that unspecified events could be allowed as part of the match on lifeline L1, but not on lifeline L2.

Let us now assume that the matching does not allow to match unspecified events in between the explicit pointcut events. There are still cases where there is a challenge to determine where the advice events shall be placed. It is common practice to place the advice events in relation to the matched pointcut events. This makes the placement of advice events trivial for lifelines where the pointcut has events to be matched. For advice events on a lifeline that is not part of the pointcut, it is an open question where to place these events in relation to already existing base events on such a lifeline.

Figure 4.2 raises more questions with respect to the matching. Except for the c message, the base diagram is identical to the pointcut diagram. Is there a pointcut match in the base model? None of the two c events splits any of the events of the a and b messages with respect to one specific lifeline. Before we conclude that there is a match, we shortly discuss this example with respect to the underlying semantics of a sequence diagram.

The semantics of a sequence diagram can be described by a set of traces, where each trace describes a possible execution run. In any possible trace, the events of the c message will split the events of the a and b messages, so in this regard we may be tempted to say that there is no match.
Syntax-based matching in general has a problem of missing to match semantically equivalent, but syntactically different patterns in a base model. Figure 4.3 illustrates an example where there is no syntax-based match. One possible execution trace, however, chooses the second operands of the two alt operators which then corresponds to the pointcut. This means that semantically there is a match. How can we define a semantics-based matching?

Figure 4.3: Syntax-based matching fails

We now discuss a desired property of the woven diagrams. A sequence diagram is invalid if the imposed partial order relation is circular, as illustrated by Figure 4.4. Assuming that the pointcut, advice and base models are all valid sequence diagrams, then it would be undesirable that the weaving process produces an invalid woven sequence diagram. Is it possible to define the matching and weaving such that we can guarantee valid woven sequence diagrams? Does the weaving guarantee valid woven sequence diagrams for both syntax-based and semantics-based matching, and even if we allow unspecified events in the match?

Figure 4.4: Invalid sequence diagram

If there are multiple aspects to be applied on the same base model representing a set of sequence diagrams, then there is a need to investigate if there are dependencies between the aspects. If there are dependencies, then the order of applying the aspects matters. A confluence theory is well-known from graph transformation and term rewriting [37, 57, 78, 53, 39]. A set of terminating rules which is confluent will always yield the same result when applied non-deterministically on the same initial graph or term, i.e. a confluent set of rules have no dependencies or conflicts. The notion of confluence can be adopted also for sequence diagram aspects, and is closest related to the existing confluence theories if we use the former weaving strategy mentioned above of applying the aspects as long as possible.

Figure 4.5 shows two aspects that are not confluent. The application of one aspect excludes the application of the other aspect on the same potential match, since they both delete an a message which is part of the other aspects pointcut. A base model counter example consists of a single a message, and will either end up as a diagram with a single b or a single c message, depending on which aspect we apply.
The discussion above leads to some of the questions that are addressed by the thesis. Does any existing confluence theories apply to sequence diagram aspects? How is the confluence theory affected by different match definitions and weaving strategies? How much expressive power can we include into sequence diagram aspects and still have decidability of confluence?

4.1.2 Graph transformation

We now turn to graph transformation rules as a means to define model transformations. As we have seen above, sequence diagram aspects are specified with the concrete syntax of sequence diagrams. Weaving according to such aspects can be seen as a special kind of model transformations. Can graph transformation rules in general be based on the concrete syntax?

As already discussed in Chapter 3, graph transformations have been proposed by several authors as a means to perform model transformations [8, 24]. The graphical way to define graph transformations, the available tool support [95, 29, 88], and the well-established theory including termination and confluence analysis [58, 78] make graph transformation appealing.

The graph concept is based on nodes and directed edges in terms of which we can represent models. Many model transformations can then be defined by a set of graph transformation rules.

In the following we consider an example transformation to refactor activity models. If there are two guarded control flows in the same direction between the same two activities, then these two control flows can be combined into one control flow. The two guard expressions will be joined into operands of an introduced or operator.

Figure 4.6 shows three alternative graph transformation rules to solve the refactoring task, where the topmost rule is based on concrete syntax. Shared identifiers denote common elements between the LHS and the RHS, and are placed next to an element. Variables to hold matched guard values are prefixed by question marks (?g1 and ?g2) and placed in the normal positions of guard values.

The other two rules are based on abstract syntax, and we have defined two versions to illustrate that the rules depend on the choice of metamodel for activity models (the metamodels are not explicitly shown, but the rules reveal parts of them). An activity is represented by a node of type actNode in the first metamodel and by activity in the other metamodel. In the first metamodel, a control flow is represented by a node of type flow with an attribute fType="control", where data flow is represented by the same node but different fType value. In the second metamodel, a control flow is represented by a node of type CFlow.
of the first metamodel, an activity node has edges (typed incoming and outgoing) going to its incoming and outgoing control flow respectively. In the other metamodel, the edges between activities and corresponding control flow go in the other direction to denote the source and target of the control flow. As we can see, the rule designer has to have detailed knowledge about the chosen metamodel in order to specify rules based upon abstract syntax.

The discussion above leads to some of the questions that are addressed by the thesis. Can rules be defined on the concrete syntax for many typical modeling languages? What if we need to say something about properties without a predefined graphical representation? Are there languages and language constructs for which it is problematic to define the rules on concrete syntax?

Next, we illustrate by a Petri net example that there is a need to extend the basic expressiveness of graph transformation rules.

The simple nature of graph transformation is probably a key factor to its success, since this makes it relatively easy to implement tools and to establish theory on its concepts. For the graph transformation designer, on the other hand, the lack of higher level constructs reduces the usability of graph transformation. This is why some authors have proposed to raise the level of abstraction by introducing new and powerful graph transformation mechanisms, e.g. the star operator [60] and recursion [33]. Our experience on a number of graph transformation examples reveals an often occurring need to match collections of similar subgraphs, as we will illustrate by an example that uses Petri nets.

A Petri net model consists of places, transitions and directed arrows. The directed arrows goes from a place to a transition or from a transition to a place. A transition $T_1$ has a preset of places which is the set of places that have a directed edge to $T_1$, and $T_1$ has a postset of places which is the set of places that have a directed edge from $T_1$. At any moment a number of tokens are assigned to each place, and each token is assigned to exactly one place.

In our concrete syntax, the tokens are drawn as small, filled circles, places are drawn as
larger, unfilled circles, and transitions are drawn as rectangles. An example is shown in the left part of Figure 4.7, where we have a single transition consisting of two places in the preset and three places in the postset. The places in the preset have one and two tokens respectively. The places in the postset have one, zero and zero tokens respectively.

![Figure 4.7: Example: The effects of firing a transition on a Petri net model](image)

A transition is enabled when all the places in the preset of a transition have at least one token. The transition within the leftmost model in Figure 4.7 is thus enabled, and we can fire a transition. When firing a transition we shall remove one token from each place in the preset and add one token to each place in the postset. The resulting model after firing the transition is shown in the right part of Figure 4.7.

Figure 4.8 shows a transformation rule that can express transition firing, but only when we have a fixed number of two preset places and three postset places. The NAC excludes application of the rule if we can match an additional preset place. There is a need for a new mechanism in order to express transition firing by a single rule for arbitrary numbers of preset and postset places.

![Figure 4.8: Rule to fire transition for a fixed number of preset and postset places](image)

### 4.2 Requirements

Based on the problem statement above, we list a set of requirements. These requirements are used to evaluate the artefacts presented in the next chapter. The requirements are based on experience from the thesis work, and this list of requirements have gradually evolved.

1. **It shall be possible to match and transform collections of similar subgraphs by specifying a single rule.** A rule traditionally matches a fixed number of elements. This requirement allows some elements to be fixed, while others represent a set of similar subgraphs. The choice of graph transformation examples we have investigated have led to this requirement, which is one desirable way to extend the graph transformation formalism. Other examples could have led to other extension needs.
2. The mechanism to match and transform collections of similar subgraphs shall be applicable to rules using both concrete and abstract syntax. The notation of the mechanism must be applicable to all kinds of graphical elements, and not be limited to the generic elements representing nodes and edges in the abstract syntax. This requirement must be seen in connection with requirement number 11.

3. A sequence diagram aspect shall define if and where the match can contain other events than those explicitly given by the pointcut. This requirement allows a single aspect to allow arbitrary events in some positions and not in others.

4. A sequence diagram aspect shall be able to match base sequence diagram extracts that have equivalent traces as the pointcut even for cases where the syntactic structure is different between the pointcut and the base model. This requirement allows the aspect modeler to concentrate on what the aspect shall match, and not how possible matches are represented in the base model.

5. The semantics-based weaving of sequence diagram aspects must be tractable. The weaving must be performed within reasonable time and not have exponential growth relative to the size of the diagram.

6. A sequence diagram aspect shall be able to match arbitrarily large pointcut structures (sequence of events) as opposed to single events only in AspectJ. This requirement allows for more precise aspects. This is especially natural for sequence diagrams where a series of events must occur before there is a match of the aspect.

7. (Soundness of weaving) Given that the pointcut, advice and base diagrams are all valid sequence diagrams, then the weaving shall guarantee a valid woven sequence diagram. It is desirable that the weaving does not produce woven diagrams that are invalid sequence diagrams.

8. A theory shall define independence criteria for sequence diagram aspect derivations that guarantee commutativity of the aspect derivations. Independence criteria are helpful for establishing other confluence results regarding undecidability and sufficient criteria to guarantee confluence. It is also helpful in an analysis to see if two pairs of aspects are dependent or in conflict.

9. A theory shall determine if confluence is undecidable for our sequence diagram aspect language with full expressiveness or with limited expressiveness. The theory shall seek to minimize the expressiveness of the aspect language with respect to when we can prove that confluence is undecidable.

10. A theory shall provide sufficient criteria or possibly an algorithmic way to determine confluence for our sequence diagram aspect language with full expressiveness or with limited expressiveness. The theory shall seek to maximize the expressiveness with respect to how confluence can be checked. Requirements 9 and 10 together provide knowledge of when we can find algorithmic ways to check confluence and not, with respect to the expressiveness of a sequence diagram aspect language.

11. It shall be possible to specify graph transformation rules based on the concrete, graphical syntax of the source and target modeling languages. The transformation designer
Problem Analysis

does not need to have any knowledge of the metamodels of the source and target languages. This is a very ambitious requirement, for which we expect that an approach only meets the requirement for a subset of all possible modeling languages. The next requirement is strongly related to this one.

12. **Concrete syntax-based rules shall be applicable to a majority of the most used modeling languages used as source and/or target language.** To be a successful candidate for general model transformations, it shall support a large number of typical modeling languages.

13. **Concrete syntax-based rules shall be based upon the well-established principles of algebraic graph transformation [37].** By basing the approach on algebraic graph transformation we can utilize the existing matching, transformation, termination and confluence results.

14. **Concrete syntax-based rules shall be at least as expressive as traditional abstract syntax-based graph transformation rules.** This means that all the abstract syntax elements must be available in some way in the concrete syntax-based environment. Otherwise, there will be abstract syntax-based rules that cannot be expressed as concrete syntax-based rules.

15. **A framework to specify and execute concrete syntax-based rules shall be supported by a set of tools.** Tool support is essential to make the approach convincing to a large community and to be used in practice.
Chapter 5
Contributions

This chapter provides an overview of the four artefacts that constitute the main contributions. For each artefact we explain the main achievements, while further details, formal definitions and proofs can be found in the research papers of part II.

Figure 5.1 shows how the eight papers contribute to the four artefacts. Artefact 3 builds upon the definition of artefact 2, while the other two artefacts have no such dependencies. All of the papers and artefacts concern the common theme of using concrete syntax in graph-based model transformations.

Figure 5.1: Overview of papers and artefacts

5.1 A Collection Operator for Graph Transformation

As observed in the previous chapter, the field of graph transformation lacks a construct to match and transform collections of similar subgraphs. Without such a mechanism, graph transformation is complex or even impractical to use in a number of cases. To address this, we first introduced the collection operator in paper 1. The collection operator can be used in a graph transformation rule to match and transform a set of similar subgraphs in one step. In paper 1, we translated a collection rule into a set of collection free rules for execution in the
AGG tool [95]. While this worked fine for our first examples, it became quite complicated in the general case. There was a need to execute the set of collection free rules stemming from a collection rule, as a transaction, handle cardinality restrictions, and ensure that the set of collection free rules shares the common context.

Due to this complicated simulation of a collection rule by a set of collection free rules, we proposed instead, in paper 5, to give a formal definition of the collection operator as an extension to algebraic graph transformation (GT). This enabled us to reuse the existing algebraic GT matching and transformation definitions and the associated tools.

A collection operator can be represented by a graph where the collection subgraph is designated by a node of type \texttt{coll}. This node has \texttt{min} and \texttt{max} as cardinality attributes, and it has a set of edges targeting all the collection subgraph nodes. The set of all collection operators in a rule \( p \) is referred to as \( \text{Coll}_p \). We use \( \psi \) to denote a function that maps each collection operator in a rule \( p \), to a number within its cardinality range, i.e. \( \psi : \text{Coll}_p \rightarrow (\mathbb{N} = \{0, 1, 2, \ldots\}) \), where \( \forall c \in \text{Coll}_p : \psi(c) \in [c.\text{min}, c.\text{max}] \).

Each possible \( \psi \) for a rule represents a collection free rule where a collection content has been copied (with fresh identifiers in each copy) the same number of times as \( \psi \) denotes. Although there are infinitely many \( \psi \) assignments when the upper cardinality of a collection is unbounded, obviously it has an upper bound when it is to be matched to a finite graph. In fact, the collection free rule can be made dynamically as part of the matching process. We illustrate how the collection operator definition works by using it to specify a fire transition rule for the Petri nets that we introduced in the previous chapter.

With two collection operators (identified as \( c_1 \) and \( c_2 \)) we can define the firing of a transition by a single rule (Figure 5.2). Collection \( c_1 \) expresses that we remove one token from each place in the preset, while collection \( c_2 \) expresses that we add one token to each place in the postset. The NAC ensures that there are no preset places without a token. This is because the \( c_1 \) collection is matched to all preset places with at least one assigned token, and the preset place of the NAC must be a different preset place than all those matched by the collection. Elements in the NAC can be the same as those in the LHS only if they share identifiers. Furthermore, two elements with shared identifiers must also be placed within two collections having the same identifier or both elements placed outside of all collections.

![Figure 5.2: Transformation rule with collections to execute the fire transition behavior](image)

The rule with collections represent infinitely many rules for all combinations of \( \text{pre}^* \text{post} \), where \( \text{pre} / \text{post} \) denotes the number of preset / postset places respectively of a transition. It is complicated to express the firing of a transition with a finite set of rules without the collection operator.

The matching and transformation process goes through the following steps:

1. Non-deterministically identify a match of the rule’s LHS, where each collection operator has been replaced by the minimum number of content copies. The minimum
number of content copies equals the minimum cardinality of each collection operator. In our example, zero is the lower cardinality for both collections. So, we simply look for a match of any transition. The outcome of this step will be a match of the single transition of our model.

2. Extend the match of each collection as much as possible. This means that the c1 collection is expanded to two collection content matches, and the c2 collection is expanded to three collection content matches. The outcome of this step is a final assignment of $\psi$, i.e. $\psi = \{c1 \to 2, c2 \to 3\}$.

3. Use the $\psi$ to dynamically build a collection free rule for $\psi$, as shown in Figure 5.3, and where the match is the one produced in the previous step.

4. Check that all NACs and dangling conditions are satisfied.

5. Apply the final collection free rule on the identified match.

Figure 5.3: A dynamically built collection free transformation rule to execute the fire transition behavior

Some of the strengths of the collection operator can be summarized as follows. Arbitrary cardinalities can be assigned to each collection operator; The collection operator works on both concrete and abstract syntax rules; One or more collection operators can be concisely expressed within a single transformation rule; The collection operator is formalized as a straightforward extension of algebraic graph transformation.

The collection operator has dramatically reduced the complexity of the rules needed in several transformation examples:

- activity diagram aspects and refactorings in papers 1 and 4,
- state machine refactoring in paper 5,
- transformation from sequence diagrams to state machines in paper 8, and
- from feature models to BPMN in paper 6.

The useful collection cardinalities so far have been: 0..*, 1..*, 0..1, and 2..*.
5.2 A Semantics-based Sequence Diagram Aspect Language

Our aspect-oriented language for sequence diagrams shares some design principles with graph transformation, but the matching is different as explained below. This makes the language highly generic by being applicable to specify traditional cross-cutting aspects as well as other transformations of sequence diagrams, e.g. refactoring. Furthermore, we need no explicit joinpoint model, since a match in a base sequence diagram is defined as a relation between an arbitrarily large ‘LHS’ diagram and the base.

In this setting, an aspect is defined by one pointcut (corresponds to LHS in GT), one advice (corresponds to RHS in GT) and an arbitrary number of negative pointcuts (correspond to NACs in GT). All these three diagram parts of an aspect are based on the concrete syntax of sequence diagrams. The aspect language extends sequence diagrams by adding identifiers to messages and lifelines. In addition, the message signals and lifeline types can have wildcard expressions, e.g. * that matches any value.

Our language is semantics-based in the sense that the matching and weaving is defined in relation to a trace-based formal model for sequence diagrams. Informally, there is a match if there exists a pointcut trace, which is contained within at least one base trace. This means that there will be a pointcut match in the base model in Figure 4.3 from the previous chapter. The weaving is defined in relation to the match. The main principle is that advice events on a lifeline replaces the matched events on the same lifeline. This weaving principle is formally defined in paper 7. Papers 2 and 3 define weaving also for an aspect which introduces events on lifelines that are not part of the pointcut. Such aspects are not allowed in paper 7. This is because they normally lead to non-confluence, and thus they are not so interesting in the confluence theory.

Theoretically the semantics-based matching on traces works fine as a definition. But in practice, and even for relatively small diagrams, a matching based on complete trace sets is intractable. This problem is due to the growth in number of traces, which is exponential relative to the number of messages. In papers 2 and 3, we first define lifeline-based matching which is equivalent to semantics-based matching, and which does not have such an inherent intractability. To identify if there is a lifeline-based match, it is sufficient to: (1) identify matches on each lifeline in isolation, and (2) ensure that there are no match blocking partial orders, i.e. no unmatched events that the base diagram requires to happen between two of the matched events. The definition of lifeline-based match was imprecise in the papers 2 and 3, and instead we refer to paper 7 for a complete definition and the associated proof. The flaw in the previous papers was that it only considered the partial orders imposed by single messages (i.e. the send event comes before the receive event) to be potentially match blocking, and not general partial orders.

To keep the aspect language fairly simple, our pointcut diagrams can only use lifelines and messages, while the base and advice can use arbitrary additional combined fragments except strict, e.g. alt, par, or loop. We have only described the approach for base diagrams with the operators seq, alt and loop, but it is also applicable to other operators if they can be rewritten to compositions of seq and alt. This means that operators such as opt and par are indirectly supported. In general only loops with a finite upper bound can be woven, but paper 3 describes a method to support weaving also for many typical loops of unbounded cardinality.

The aspect language includes support for decomposition of the aspects in a similar way as in ordinary sequence diagrams. A decomposed lifeline which occurs in both the pointcut and the advice leads to two levels of diagrams both for the pointcut and the advice. The UML
2 specification in [74] defines some rules for decomposition, and these rules also apply to our aspects with decomposition. Paper 2 shows how decomposition of aspects works on a concrete example. The decomposition is only used to structure the diagrams, and is not visible at the trace level where only the non-decomposed lifelines appear. Thus, our trace-based match and weave definitions need no extensions to support decomposed diagrams.

So far a match requires that a sequence of events on a pointcut lifeline must be a continuous subsequence of the corresponding base diagram’s lifeline event sequence. However, it is recognized by other authors that this match concept in some cases is too strict [100, 51], for instance in cases where multiple aspects are to be woven into the same base diagram.

As opposed to the other authors, we have introduced an explicit graphical symbol, called the arbitrary events symbol (\(\parallel\)). This symbol is to be placed in lifeline positions of the pointcut where an arbitrary number of interfering events are allowed in the match. This has several benefits: (1) the same pointcut can define that arbitrary events are allowed in some positions and not in others, (2) the placement of the advice events can be precisely defined in relation to the arbitrary events symbols.

Figure 5.4 shows an aspect that uses two arbitrary events symbols. The pointcut defines that we are looking for matches of an a message followed by a b message, and the arbitrary events symbol used on both the lifelines indicate that there may be arbitrary events in between the send a event (\(!a\)) and the send b event (\(!b\)) on lifeline L1, and between receive a (\(?a\)) and receive b (\(?b\)) on lifeline L2. The corresponding advice adds an ad message with an explicit position relative to the arbitrary events. The send event of ad, \(!ad\), shall be inserted directly after the \(!a\) event (and before all the arbitrary events) on lifeline L1, and the receive event, \(?ad\), shall be inserted directly before the \(?b\) event (and after all the arbitrary events) on lifeline L2.

The arbitrary events symbols must be repeated in the advice. This is to avoid problems such as producing woven diagrams with only the send or receive event of a message. Our theory assumes that all the pointcut, advice and base diagrams only have complete messages, i.e. both (or none of) the send and receive events of a message must be part of the diagram. Furthermore, multiple arbitrary events symbols on a pointcut lifeline must maintain the same order in the advice, which means that no identifiers are needed to pair corresponding arbitrary events symbols of the pointcut with the advice. More details about the arbitrary events symbol are found in paper 2.

Paper 7 provides formal match and weave definitions for sequence diagram aspects including negative pointcuts and arbitrary events symbols. If we consider the subclass of aspects where advice events are not introduced on lifelines without pointcut events, then we

---

1Whittle et al. [101] have in parallel with our work introduced a combined fragment of type any in the MATA tool, which is an alternative to our arbitrary events symbol. However, the MATA tool uses syntax-based matching as opposed to our semantics-based matching.
get a strong result which we call *weaving soundness*. Given that the pointcut, advice and base diagrams are all valid sequence diagrams, then the woven result will always be a valid sequence diagram (proof is given in paper 7). A sequence diagram is invalid if the imposed partial order relation is circular, as illustrated by Figure 4.4 from the previous chapter. The weaving soundness also trivially holds for negative pointcuts, since they only restrict the cases for which we can weave. For arbitrary events symbols, the weaving soundness also holds if we add the restriction that ‘the advice cannot introduce a partial order with the receive event before an arbitrary events symbol on one lifeline and the corresponding send event after an arbitrary events symbol on another lifeline’, where ‘before’/‘after’ is not necessarily immediately before/after.

Our semantics-based weaving does not lead to woven diagrams which are nicely structured. This is because the weaving first rewrites a base diagram with combined fragments into a set of base diagrams without combined fragments. This is required by the lifeline-based matching, which is only defined for diagrams without combined fragments. A matching and weaving is performed on all the base diagrams. Finally, we produce a woven diagram with one outermost alt operator and one operand for each woven base diagram. Therefore our semantics-based weaving must only be applied in cases where we do not intend to continue work on the woven diagrams. Our semantics-based weaving is intended for automatic generation of other artefacts, e.g. to test if a sequence diagram is a correct refinement of another sequence diagram [61], or to test if a system specified by UML state charts, class diagrams and object diagrams is consistent with sequence diagram specifications [75].

We have a tool which also includes weaving of the above mentioned supported class of unbounded loops. The tool uses the Eclipse-based SeDi sequence diagram editor v.1 [59] to define base, pointcut and advice diagrams. The weaving has been tested to behave correctly on several examples, by manually investigating the woven textual interactions. There is currently no support in the tool for the arbitrary events symbol and decomposition (decomposition is first introduced in SeDi v.2). We believe it is straightforward to also support these parts in a future version based on SeDi v.2.

5.3 A Theory for Confluence Analysis of Sequence Diagram Aspects

In paper 7 we describe a confluence theory for our sequence diagram aspect language including negative pointcuts and the arbitrary events symbol. A confluence analysis is helpful in order to automatically detect if there are dependencies among aspects. Non-confluent aspects often means that it is necessary to specify an explicit weaving order, redesign some of the aspects, or exclude one or more aspects.

Our aspects can be seen as analogous to term rewrite rules or graph transformation rules. There already exists a well-established theoretical foundation on confluence for graph transformation systems (GTS) [37, 57, 78], and confluence for term rewrite systems [53, 39]. These confluence theories are, however, not directly transferable to our aspects. The concrete syntax of sequence diagrams and aspects defined upon these are quite different from graphs and GTS rules. For a node in a graph, there is no order among its incoming and outgoing edges. In sequence diagrams, on the other hand, the events are partially ordered. The partial order also makes sequence diagrams different from term rewrite systems, where the elements in a term are totally ordered. The semantics-based matching and weaving further complicates the relation to the well-known existing confluence theories.
From Newman’s lemma [64] confluence equals local confluence for terminating rules / aspects. As already defined in Section 3.1, local confluence means that two direct derivations from the same base diagram, are always joinable. We have not provided a termination theory for our aspects, but we assume that the set of aspects are terminating. This allows us to focus on the joinability of all possible pairs of direct derivations from the same base diagram.

We define two derivations from the same base diagram to be independent if and only if these two derivations can be directly joined by commuting the two derivations. While Lambers et al. [57] identify two different conflict types that can make two derivations dependent in GT, we identify five types of conflicts for sequence diagrams of which two are variants of those from GT. Two of the new conflict types (produceMB-blocked and deleteMB-forbid) relate to match blocking partial orders, which can be seen as a kind of fixed negative pointcut for an aspect. A match blocking partial order is a partial order between two of the unmatched base events, and such that the partial order of the base diagram requires these two events to happen between two of the matched events. The partial order is match blocking since it prevents a pointcut trace to be a continuous subsequence in a base trace, i.e. it prevents a match.

Our five conflict types are:

- **use-delete.** This conflict type occurs when one derivation deletes a direct relation in the partial order which was part of the other derivation’s match. Analogous to a GT conflict type.

- **produce-forbid.** This conflict type occurs when one derivation adds something that leads to a match of one of the other aspects negative pointcuts. Analogous to a GT conflict type.

- **produceMB-blocked.** This conflict type occurs when one derivation adds a match blocking partial order for the other derivation’s potential match.

- **produce-produce.** This conflict type occurs when the commutation of the two derivations leads to an unequal order of events on a lifeline.

- **deleteMB-forbid.** This conflict type occurs when one derivation deletes what previously was a match blocking partial order for one of the other aspects negative pointcuts.

The conflict type produce-produce is due to the ordering of events on a lifeline. Produce-produce conflicts do not occur in GT, since there are no orders among the incoming or among the outgoing edges of a node.

Figure 5.5 shows an example of a produceMB-blocked conflict. The MakeMB aspect produces a match blocking partial order (the message mb) for the other derivation, and thus the two derivations are dependent. Notice that although the messages a and bm are crossing in the base model, there is still a match of the A1 aspect. The message bm is not a match blocking message since none of the bm events has to happen between any of the a events.

The following theorem states an important result for one of the most expressive forms of aspect diagrams:

**Theorem 5.3.1 (Undecidable).** It is undecidable to determine confluence of an arbitrary finite set of terminating aspect diagrams with negative pointcuts and the arbitrary events symbol.
The theorem above is proven in paper 7. As defined in Section 3.1, confluence means that all derivations from the same base diagram must be joinable, and there is only confluence if this holds for any base diagram. The joinability of all derivations from a fixed base diagram is trivially decidable if we have a finite set of terminating aspects. With a fixed base diagram we can perform all possible derivations, of which there are only a finite number.

If we do not allow aspects with negative pointcuts or arbitrary events symbols, paper 7 proves a theorem that can be used to algorithmically decide if confluence holds or not. The theorem is based on analysing a set of critical pairs. As opposed to term rewriting systems and GT, the joinability of what we call minimal context critical pairs is not sufficient to claim confluence. A minimal context critical pair consists of two dependent derivations, where the base diagram consists only of matched messages from the two pointcuts of the two derivations.

In our confluence check, we first calculate all the minimal context critical pairs. Each minimal context critical pair is then extended in a systematic way into what we call an extended critical pair such that this critical pair is maximally difficult to join. This is accomplished by reducing the applicable aspects in two ways: (1) the introduction of new events in between the two matches of the critical pair, and (2) the introduction of match blocking partial orders. Both these extensions may prevent matches and hence the application of aspects that are used to join the two derivation results in the critical pair. At the same time we can only insert messages that do not prevent the two derivations of the critical pair.

Figure 5.6 shows an example where all minimal context critical pairs are joinable, but we still do not have confluence since an extended critical pair is not joinable. The three aspects are plain additive and they all add an adv message. The A1 – A2 derivations for a base diagram in the middle part of Figure 5.6 shows the only minimal context critical pair. This critical pair is joinable by applying the A3 aspect on the two woven diagrams in the critical pair.

In the bottom part of Figure 5.6, we show the extended critical pair. The base diagram of the extended critical pair extends the minimal context critical pair base diagram by adding messages with signal $\chi$. Messages with signal $\chi$ shall not occur in any of the pointcut diagrams. The $\chi$ message does not prevent the first A1 and A2 derivations. However, the A3 aspect is no longer applicable since $(\chi, \chi)$ becomes a match blocking partial order. We reach two normal forms which are different, and we have shown that the set of aspects is not confluent.

The main result of the confluence theory is given by the following theorem:
5.4 A Framework for Concrete Syntax-based Graph Transformations

Algebraic graph transformation can be used to specify model transformations [96]. Such transformations have a general applicability in that they can be defined for most modeling languages. A disadvantage is that the transformation designer has to use the abstract syntax, in which a detailed knowledge of the metamodels of the source and target languages is needed.

This section provides an overview of a framework that allows the user to define the LHS/RHS/NACs of a rule by using source and target model elements in concrete syntax. Further details are given in paper 6. This framework, which we call concrete syntax-based graph transformation (CGT), has the following properties:

- **Concrete syntax.** The transformation modeler can think entirely in the concrete, graphical syntax of the source and target modeling languages, i.e. no knowledge of the source or target metamodels is needed.

- **Graph transformation.** CGT rules are automatically compiled into algebraic graph transformation rules in abstract syntax, which means that the well-established algebraic graph transformation theory [22, 78, 57, 58] and tools [95, 7, 6] can be directly applied.

---

**Theorem 5.3.2 (Confluence).** A set of terminating aspect diagrams (without the arbitrary events symbol and without negative pointcuts) is confluent if and only if all the extended critical pairs are joinable.

---

**Figure 5.6:** Joinable minimal context critical pairs does not imply local confluence
Contributions

- **Generic.** CGT is applicable to a majority of the most commonly used source and target languages. CGT shall strive to make concrete syntax-based rules appropriate for the most common modeling languages. Abstract syntax can still be used for non-graphical modeling languages or if there are problematic concrete syntax constructs.

Figure 5.7 explains the process that an editor designer performs in order to configure and automatically generate a concrete syntax-based rule editor. The resulting editor allows a model transformation designer to specify concrete syntax-based rules, compile these automatically into abstract syntax-based rules and execute the transformations on a source model to produce target models. The model transformation designer can concentrate on the concrete syntax to specify the model transformation, while the actual implementation behind the scenes uses the abstract syntax and a traditional graph transformation tool to do the transformation.

![Transformation modeler tasks to configure the rule editor](image)

The editor designer starts by providing the definitions of the source and target metamodels, i.e. the abstract syntax. For commonly used modeling languages, such metamodel definitions may be publicly available. Assuming our framework supports the available format, it is sufficient to import the definition. Otherwise the editor designer needs to define the abstract syntax definition(s) from scratch.

The next step is to associate graphical representations with the abstract syntaxes, i.e. to define the concrete syntaxes. In this step, the editor designer specifies a bidirectional mapping between the concrete and abstract syntax, for both the source and target languages. Again, for commonly used modeling languages, it may be sufficient to import such mappings from a publicly available registry. The Graphical Modeling Framework (GMF) [21] is one example of a tool where an editor designer can define the concrete syntax and mapping to a corresponding abstract syntax. The steps of defining source or target concrete syntaxes can be skipped, but then only abstract syntax will be available in the rule editor.

When the source and target abstract and concrete syntaxes are defined, a fully automated tool can generate a rule editor. Our approach of using concrete syntax-based rules that are automatically compiled into abstract syntax-based rules means that we can reuse existing matching and transformation engines, as well as switching to graphical, abstract syntax when this is better suited.

When using algebraic graph transformation to perform model transformations, there was also prior to our approach a need to define the mappings between the models (i.e. concrete syntax) and the graphs (i.e. abstract syntax). The basic idea of our approach is that these mappings can be reused when mapping from concrete syntax-based rules to abstract syntax-based rules. Papers 1,4,5 and 6 show that this principle works fine for many transformation
scenarios. However, special treatment is necessary to handle (1) order-significant models and (2) merged graphical elements. These concepts are explained in the following paragraphs.

Order-significant elements. The problem with order-significant elements occurs when an element has a significant order among its connected elements, e.g. the order of the events on a sequence diagram lifeline. In algebraic graph transformation there is no order among the incident edges of a node.

Merged graphical elements. When multiple abstract syntax elements are represented by concrete syntax elements that are overlapping, very close to each other or even merged into combined concrete syntax elements, then there may be difficulties to use our approach without special treatment. This challenge occurs for an \texttt{alt} operator and its operands, and for a UML state machine and its regions. Such constructs can be difficult to match and transform, and in future work we plan to introduce special treatment for such constructs.

Paper 8 proposes a way to handle sequence diagrams, which needs special treatment in CGT since the language has both order-significant elements and merged graphical elements. We need to stress that using sequence diagrams as the source and target language in CGT is quite different from our sequence diagram aspect language. This is clarified in Section 6.1 of the next Chapter.
Chapter 6

Discussion

This chapter is organized in three sections. The first section discusses some design decisions and how the aspect language for sequence diagrams is complementary to concrete syntax-based graph transformation applied to sequence diagrams. The second section evaluates our artefacts against the requirements identified in Section 4.2. The third and final section discusses related work to our thesis work.

As a shorthand we use the following numbering of the artefacts. Artefact 1 = a collection operator for graph transformation; Artefact 2 = a semantics-based sequence diagram aspect language; Artefact 3 = a theory for confluence analysis of sequence diagram aspects; and Artefact 4 = a framework for concrete syntax-based graph transformations.

6.1 Design Decisions

In this thesis we have presented two quite different transformation approaches for sequence diagrams: (1) the sequence diagram aspect language, and (2) concrete syntax-based graph transformation. The latter is used in paper 8 to transform from sequence diagrams to state machines. Our aspect language to do transformations on sequence diagrams is semantics-based, but not structure-preserving in the sense that all the structure of combined fragments etc. is broken down in the weaving process. The graph transformation approach on the other hand, is purely syntax-based, but structure-preserving.

In some cases a semantics-based approach may be better suited. This goes for cases where the pointcut consists of several elements that can found in the base model in different syntactical representations, so that there is a risk to miss out relevant matches. This is not the case for the transformation from sequence diagrams to state machines, where we systematically match and treat each topmost element on the lifeline. Instead the syntax-based approach is better suited for that transformation, since structure-preservation of the intermediate models is important.

There are several graph transformation approaches and tools, so why did we choose the algebraic graph transformation and the AGG tool? This is because of the strong theoretical theory regarding termination and confluence analysis, which is also supported by AGG. While AGG is useful as a research tool, it is not so suited in many industrial settings since the performance and scalability is better addressed by other tools, e.g. Viatra [99] and PROGRES [86, 88]. Also, AGG does not support the ’set nodes’ supported by Fujaba [29] and PROGRES, which can be used as an alternative to the collection operator for simple cases.
Still, as discussed in Chapter 4, the set nodes are too restrictive in the general case.

Why have we chosen to use the two-graph notation style for our rules, while other authors have chosen the more compact single-graph notation style [101, 42, 43, 73, 81]. These two notation styles have advantages and disadvantages. We see the following advantages of the two-graph notation style compared to the single-graph: (1) the two-graph notation style is closer to the algebraic GT theory since the theory is based on three graphs, (2) the LHS shows clearly everything to match and nothing else, and (3) the graphs are not cluttered by special tags to indicate the elements to be added, deleted and those belonging to NACs.

The two-graph notation style has a disadvantage compared to the single-graph style in that there is need to use identifiers to denote shared elements between the LHS and the RHS. The elements to be deleted are clearly marked by deletion tags (e.g. del) in the single-graph style, while this correspond to LHS elements without identifiers in the two-graph notation style. The single-graph style has an advantage by being the most concise of the two notation styles. As mentioned also in Section 3.1, a tool that allows to switch freely between the single-graph and the two-graph notation styles may be an ideal solution.

Why have we chosen to use a trace-based formal model, instead of an automata-based formal model, for our semantics-based sequence diagram aspect language? The UML 2 specification [74] explains the semantics of a sequence diagram by traces of events and in STAIRS [85, 36] a trace-based formal model for sequence diagrams is defined. By basing our aspect language on that formal model, we can utilize the existing formalism from STAIRS that includes semantics for combined fragments and a refinement theory.

A trace-based formal model gives a denotational semantics as opposed an automata-based formal model which represents operational semantics. It is easier to formalize the matching and weaving upon a denotational semantics than with operational semantics. This makes it easier to derive formal proofs for confluence, termination etc. A disadvantage with denotational semantics in general may be that implementations based directly upon it may be inefficient. This is the case also for the trace-based formal model, and instead our matching and weaving tool has been based on an equivalent, more efficient strategy which we have called lifeline-based matching.

We have only considered sequence diagrams with asynchronous messages, which is common practice also in many other works which involve sequence diagram-like specification languages, e.g. for message sequence charts [62, 52], and for sequence diagrams [85, 36, 41, 51].

When sequence diagrams use synchronous messages, there is an additional requirement on the event orders, but there are variations on how to interpret the semantics. Harel and Maoz [35] say that a message sent from a lifeline L must also be received (normally on a different lifeline) before the next event occurs on lifeline L. Another even stricter interpretation by Amstel [98] is that a synchronous message is a single action where no events can take place between the send and receive events of that message. In addition to different interpretations, Dan et al. [14] show that some precaution is needed when modeling sequence diagrams with synchronous messages, in order to get the intended trace semantics.

With any of the semantic interpretations of synchronous messages, the traces imposed by a sequence diagram with synchronous messages will be a subset of the traces imposed by the same sequence diagram where one or more of the messages are changed to asynchronous messages. This means that our trace-based match definition still applies even for synchronous messages. With synchronous messages only, crossing messages will not be allowed and becomes an additional criteria when ensuring that our weaving produces valid woven sequence diagrams.
Although all our examples in the thesis are behavioral models, our approach of using concrete syntax-based graph transformation rules shall be applicable also to structural models for which algebraic GT has many successful applications.

6.2 Evaluation of the Artefacts with Respect to the Requirements

This section discusses to what degree the requirements listed in Section 4.2 have been met by our artefacts. Requirement 1 is addressed by artefact 1; Requirement 2 is addressed by artefact 1 and 4; Requirements 3-7 are addressed by artefact 2; Requirements 8-10 are addressed by artefact 3; and Requirements 11-15 are addressed by artefact 4.

1. It shall be possible to match and transform collections of similar subgraphs by specifying a single rule. This is achieved by introducing the dashed line frame of the collection operator. When used in the LHS, it expresses matching of similar subgraphs. When used in the RHS, it expresses transformation of similar subgraphs.

2. The mechanism to match and transform collections of similar subgraphs shall be applicable to rules using both concrete and abstract syntax. This is partially achieved by having a notation that can be placed around arbitrary elements. For some constructs, where several elements are merged into the same graphical construct (e.g. combined fragments of sequence diagrams, state regions of a state machine), a new, generic graphical construct was introduced in paper 8. For those modeling languages we have not investigated, there may be other graphical elements for which it is difficult to use the collection operator in an intuitive way. So far we have mainly investigated many of the UML diagram types and Petri nets.

3. A sequence diagram aspect shall define if and where the match can contain other events than those explicitly given by the pointcut. This is achieved by using the arbitrary events symbol on the relevant positions of pointcut diagram lifelines. These positions are the only positions where additional events, not explicitly given by the pointcut, are allowed in base diagram matches.

4. A sequence diagram aspect shall be able to match base sequence diagram extracts that have equivalent traces as the pointcut even for cases where the syntactic structure is different between the pointcut and the base model. This is achieved by defining the matching in relation to traces for a trace-based formal model.

5. The semantics-based weaving of sequence diagram aspects must be tractable. Calculating the entire trace sets is not tractable even for relatively small sequence diagrams. Instead we use a lifeline-based matching. In paper 7 we prove that a lifeline-based matching and the absence of match blocking partial orders is equivalent to a trace-based matching. The latter matching process is tractable.

6. A sequence diagram aspect shall be able to match arbitrarily large pointcut structures (sequence of events) as opposed to single events only in AspectJ. This is achieved by allowing the pointcut to be an arbitrarily large sequence diagram.
7. (Soundness of weaving) Given that the pointcut, advice and base diagrams are all valid sequence diagrams, then the weaving shall guarantee a valid woven sequence diagram. This is proven by the proof of Lemma 3 in paper 7 for aspects without the arbitrary events symbol, and by the proof of Lemma 6 in paper 7 for aspects with the arbitrary events symbol.

8. A theory shall define independence criteria for sequence diagram aspect derivations that guarantee commutativity of the aspect derivations. The independence criteria are given in Lemmas 7 and 8 in paper 7.

9. A theory shall determine if confluence is undecidable for our sequence diagram aspect language with full expressiveness or with limited expressiveness. The proof of Theorem 1 in paper 7 proves that confluence of our sequence diagram aspects is undecidable, when we use full expressiveness.

10. A theory shall provide sufficient criteria or possibly an algorithmic way to determine confluence for our sequence diagram aspect language with full expressiveness or with limited expressiveness. This criterion is met by restricting the aspects from using negative pointcuts and the arbitrary events symbol. The proof of Theorem 2 in paper 7 means that we have an algorithm to check confluence for such aspects. It is sufficient to check the joinability of what we call extended critical pairs. As with ordinary critical pairs, such a check can be time-consuming. The performance issue for the extended critical pairs has not been addressed by this thesis.

11. It shall be possible to specify graph transformation rules based on the concrete, graphical syntax of the source and target modeling languages. For some of the most commonly used modeling languages, we have shown that transformation rules can be based on the concrete, graphical syntax of the source and target modeling languages.

12. Concrete syntax-based rules shall be applicable to a majority of the most used modeling languages used as source and/or target language. This criterion is not sufficiently tested. All the languages we have experimented with can be used within the framework. However, we needed to introduce some extensions for several of the languages. When the framework is applied to other languages, more extensions to the framework may be necessary.

13. Concrete syntax-based rules shall be based upon the well-established principles of algebraic graph transformation [37]. This is achieved by compiling the concrete syntax-based rules into traditional abstract syntax-based rules, and by following the main principles of graph transformation also on the concrete syntax level.

14. Concrete syntax-based rules shall be at least as expressive as traditional abstract syntax-based graph transformation rules. This is achieved by requiring that the mapping from abstract to concrete syntax is sufficiently comprehensive, and that the generated rule editor provides property views for non-visualized elements. Hence, all the metamodel properties of the source and target models become editable in the rule editor, directly as graphical elements or within additional property views.

15. A framework to specify and execute concrete syntax-based rules shall be supported by a set of tools. This is not covered by the thesis work. We have hardcoded some configurations of the framework in prototype tools (papers 1 and 4), but the main tool
6.3 Related Work

In this section we describe related work for each artefact. We cover the closely related work in more detail than in the attached papers of Part II, while the attached papers complement with additional discussion of related work. Since there are no other approaches providing a comparable confluence theory for sequence diagram aspects, the section for that artefact describes our results in relation to graph transformation and term rewriting systems.

Related Work for Artefact 1

There are some works carried out in parallel with our thesis work that propose alternatives to the collection operator [81, 28, 38, 65]. In all these alternatives, a single rule can match collections of subgraphs. Figure 6.1 shows a rule to simulate transition firing in Petri nets, expressed using different notations.

The topmost rule is expressed using our proposed collection operator, which is the only operator defined to work also on the concrete syntax of the source and target languages, and not only on the abstract syntax of graphs. Rensink [81] uses exists (\(\exists\)) quantifiers in a frame to denote elements outside of collections, and for all (\(\forall\)) quantifiers in a frame to denote
elements inside a collection. Notice also that Rensinks rules use a single graph style where the LHS are merged with the RHS. Special markers denote elements to be added (add), or deleted (del), or elements belonging to NAC conditions (neg). Fuss and Tuttlies [28] use a multi-rectangle symbol to surround the collection content. Minas and Hoffmann [38, 65] use the same symbol, but it can only be attached to single nodes. Shared identifiers among such nodes mean that they belong to the same collection, and all the incident edges of collection nodes are defined to belong to the collection.

Fuss and Tuttlies [28] and Rensink [84] goes beyond our collection operator in that they support nested ‘collection operators’. None of the related works support collection cardinalities beyond 0..* and 1..*. The other approaches focus only on applying their collection operators for the abstract syntax. Still, the notations proposed by Rensink [81] and as sketched by Fuss and Tuttlies [28] have a nature which makes them appropriate to be introduced on the concrete syntax. This is not the case for the notation proposed by Minas and Hoffmann [38, 65].

Our formalization of the collection operator is different from those mentioned above. When a match is found we reduce the collection rule into a collection free rule. Each collection is replaced by the same number of collection content copies as the actual match. Then we apply the dynamically produced collection free rule as an ordinary rule within the framework of algebraic graph transformation using the double-pushout approach.

Related Work for Artefact 2

Klein et al. [52, 51] describe a semantics-based aspect language for two closely related diagram types: Hierarchical Message Sequence Charts (HMSC) and UML 2 sequence diagrams. Like in our language the semantics of the diagrams in Klein et al. is based on asynchronous messages. Although Klein et al. use a finite automata-based formal model and we use a trace-based formal model, the semantic interpretation of sequence diagrams seems to be equivalent.

In their approach all the non-overlapping matches of an aspect are identified and all are woven in one step, while we apply an aspect on one match at a time and continue until there are no more matches. As opposed to their approach, our matches can be overlapping. Furthermore, the application of an aspect in our approach may introduce new matches that were not present in the initial base model. For plain additive aspects where the advice contains the pointcut, we prevent non-termination by requiring that the same match cannot be used again with the same aspect.

Klein et al. [51] define four different match strategies of which we directly support two. On the other hand their match strategy is more restricted since their chosen match strategy must apply to an entire pointcut. We can freely mix our two supported match strategies in the same pointcut since we have introduced the arbitrary events symbol. Additional events in the base diagram are only allowed in positions where an arbitrary events symbol is placed. The need for the two match strategies from Klein et al. that we don’t support, is not justified by any examples, and the usefulness thus remains to be shown.

They have two benefits compared to our approach. Firstly, they allow arbitrary nesting of loops and alts in the based model, while our weaving does not support loops and alts within unbounded loops. Secondly, they can always ensure that matches are treated as early as possible, while we only support a random match strategy if there are unbounded loops.

On the other hand, our approach have some benefits compared to their approach. The automata-based approach cannot handle unbounded loops that lead to irregular trace expres-
6.3 Related Work

sions. Such loops impose no problem in our approach. Although an unbounded loop represents infinite trace sets, the weaving can always be performed on a bounded structure. This bounded structure is calculated in relation to the size of the pointcut. The old unbounded loop will be rewritten to a new loop where the loop body is finally woven. Neither the approach by Klein et al. nor our approach consider weaving of unbounded loops involving arbitrary events.

Klein et al. do not define how advice events are to be placed in relation to arbitrary events. If this is randomly selected or if the designer can choose freely, then there is a risk of producing invalid woven sequence diagrams. In paper 7 we ensure valid woven sequence diagrams by restricting where the new advice events can be placed in relation to the arbitrary events symbols.

Our aspect language goes beyond Klein et al. by introducing negative pointcuts and decomposition for the aspects.

Related Work for Artefact 3

Independence of our aspect derivations is more complicated than the independence of graph transformation rules. Leen et al. [57] have identified two cases that can make two derivations dependent, use-delete and produce-forbid conflicts. A use-delete conflict occurs if one derivation deletes elements that are matched by the other derivation. A produce-forbid conflict occurs if one derivation adds elements that produce a match for the other derivations NAC. With our aspects we get three additional conflict types which are caused by the partial order property of trace events.

For term rewriting systems all critical pairs are joinable if and only if there is confluence [53]. This is not the case for graph transformation. Confluence of graph transformation rules, even without NACs, is undecidable as proven by Plump in [78]. Similarly, we prove in paper 7 that confluence of sequence diagram aspects is undecidable, but only when the aspects can use negative pointcuts and the arbitrary events symbol.

When limiting the expressiveness of our aspects by excluding negative pointcuts and the arbitrary events symbol, confluence can be algorithmically checked based on our notion of extended critical pairs. We prove that a set of terminating aspects is confluent if and only if all extended critical pairs are joinable. In graph transformation and term rewriting, the critical pairs are completely constructed from merging elements from the LHSs of a pair of rules. In our extended critical pairs we introduce new events, with a reserved message signal $\chi$, in special positions of the extended critical pair.

Related Work for Artefact 4

AToM³ [16] is an existing graph transformation tool in which you can define the rules by using concrete syntax-based rules that are compiled into python code with their own matching and transformation engine. This makes it best suited for transformations where both the source and target languages have a concrete, graphical syntax. For other languages that lack a concrete, graphical syntax, AToM³ doesn’t provide a generic abstract syntax. Hence, transformations involving such languages needs to be written in python code. In contrast our approach of using concrete syntax-based rules that are compiled into abstract syntax-based rules means that we can reuse existing matching and transformation engines, as well as switching to graphical, abstract syntax when this is better suited. Furthermore, confluence and termination analysis can be directly applied if our rules are compiled into AGG rules.
The TIGER tool from Biermann et al. [7, 83, 26, 82] allows transformation rules to be defined using the concrete syntax. Just as in our approach, these rules are then compiled into abstract syntax rules, where the actual transformation takes place.

Their strength is that they already have a tool implementation, but their approach is so far limited compared to ours in that the approach only works for endogenous transformations. All attributes are placed in so called 'property views'. In our approach this is the last resort only to be used for attributes without a defined visualization. The examples shown so far are all about simulating behavior, e.g. the simulation of the Ludo game.

Baar and Whittle [3] show how to express concrete syntax-based rules that are equally expressive as QVT graphical rules [70]. A rule has a LHS and a RHS, but uses OCL expressions in a \textit{when-clause} instead of graphical NACs. They support transformations where the source and target language are different, although the source and target language is the same in all their examples (which are all about model refactorings).

It is not trivial to see how Baar and Whittle could express our example transformation from feature models to BMPN (see paper 6) without support for our generic nodes/edges and the collection operator. Their refactoring rules for UML class diagrams should be possible to express in our approach, where most of the complexity lies in defining NACs that correspond to their OCL when-clauses.

In the SmartAdapters tool described by Ramos et al. [79] and by Brice et al. [66], the notion of \textit{model snippets} corresponds very closely to our concrete syntax-based LHS rules and NACs. The transformation part, which is expressed entirely by RHS models in our approach, needs to be specified by textual composition directives in their approach.
Chapter 7

Conclusions and Future Work

This chapter is separated into two sections. The first section summarize the main achievements, and the second section sketches directions for future work.

7.1 Achievements

We have introduced the collection operator for graph transformation as a way to easily specify the matching and transformation of collections of similar subgraphs. The notation is concise and applicable to all kinds of graphical elements. The collection operator differs from parallel work from other authors in that it allows for arbitrary cardinalities and by its straightforward extension of algebraic graph transformation. Our experience in a number of transformation examples is that a graphical construct similar to a collection operator is essential to be able to define the transformations in a user-friendly manner.

An aspect language for sequence diagrams can introduce many benefits to the model-driven development. The set of basic sequence diagrams does not need to be cluttered with repetitive interaction fragments if we use sequence diagram aspects. Sequence diagram aspects help to modularize, isolate and better maintain cross-cutting concerns at the model level. Our language and the language by Klein et al. [52, 51] are the only semantics-based sequence diagram aspect languages. We use a trace-based formal model while they use an automata-based formal model. To our best knowledge our arbitrary events symbol and the property of guaranteeing valid woven sequence diagrams are novel. We have implemented a tool to define aspects and to perform weaving for a large part of the aspect language (excluding negative pointcuts and the arbitrary events symbol).

Confluence analysis. No other works have introduced a confluence analysis for sequence diagram aspects. In paper 7 we have explained why the confluence theories from string rewriting and graph transformation are not directly applicable. We prove that confluence is undecidable with high expressiveness in the sequence diagram aspects. By restricting the expressiveness to exclude negative pointcuts and the arbitrary events symbol, we show that confluence can be algorithmically checked. The joinability of traditional critical pairs is not sufficient to claim confluence. By introducing the notion of extended critical pairs we can check if we have confluence or not.

In the thesis work we have experimented with several model transformation scenarios of which we use concrete syntax instead of abstract syntax. This is unproblematic for some modeling languages, while special treatment is needed for some other modeling languages.
Conclusions and Future Work

7.2 Future Work

In this section possible directions for future work are described.

There are reported cases where a nesting of collection operators is needed [84]. The formalization of nested collection operators would be a natural extension to the formal definitions of the collection operator, and current investigations indicate that this is feasible. Tool support within an algebraic graph transformation tool would also make the operator available to the graph transformation community. An extended theory is needed in order to use the confluence and termination theory for algebraic graph transformation in combination with the collection operator.

We have not considered combined fragments with guards in our sequence diagram aspect language. Since guards are important in sequence diagrams, future work should ensure that the aspect language can support base diagrams with guards. Some investigation is needed to see how our aspect language can be extended to also work on negative behavior. Additional case studies are needed to find out if the aspect language is expressive enough or if additional constructs should be introduced.

Experiments with traditional graph transformation in the AGG tool [95] reveal that even relatively small rules may lead to severe performance problems for the confluence analysis. Hence, in order to apply our theory for confluence analysis, it is useful to investigate the computational complexity to see if there are inherent scalability problems.

We have proven that confluence of aspects, for the class of aspects with arbitrary events symbols and negative pointcuts, is undecidable. Although it is undecidable for that class of aspects, one can still identify sufficient criteria to guarantee confluence, which is a natural extension of our confluence theory. Our thesis work also leaves unanswered questions about the decidability of the following two classes of aspects:

- aspects with arbitrary events symbols and without negative pointcuts
- aspect with negative pointcuts and without arbitrary events symbols

Confluence theory depends on termination since our results only hold for terminating sets of aspects. A termination theory for sequence diagram aspects will be a complement to the confluence theory in addition to having value on its own.

Tool support is crucial to get wide acceptance for any model transformation approach. Full tool support for our transformation framework requires considerable effort and goes far beyond the scope of this thesis. As a first step, a lightweight implementation that supports only the directly applicable languages, could be more realistic. A promising path to follow is the Graphical Modeling Framework (GMF) in Eclipse, since the principle is quite similar. Both our framework and GMF is based on defining and linking the concrete, graphical syntax to the abstract syntax (metamodel).

The development of the framework is an emerging task. It is necessary to continue testing new languages and transformation examples to see if the framework is still applicable. If not applicable, there are two possible paths: 1) extend the framework to cope with such languages, or 2) define that languages with certain properties are not supported by the framework. It would be a large benefit to potential users of the framework if future work could determine exactly under which conditions the framework can be applied, i.e. if there are certain graphical constructs that makes the language unsupported by the framework. Furthermore, it is interesting to find out if is possible to develop general extension mechanisms to make the approach applicable to unforeseen languages.
Bibliography


Part II

Research Papers
Appendix A

Overview of the Papers

The main results of the thesis are documented in the papers in Part II. The papers can be read independently. Paper 1 contains some early work on the collection operator which is improved and extended in paper 5. Papers 1, 4 and 8 can all be seen as configurations of the generalized approach described in paper 6. The formalization of the semantics-based aspect language, described in papers 2 and 3, has been improved and more thoroughly covered in paper 7. However, papers 2 and 3 still contain contributions that are not found in paper 7.

In this chapter each paper is listed with the relevant publication information, an identification of my contribution and a short introduction to the topic.

Paper 1: Aspect Diagrams for UML Activity Models


Publication details: This paper was accepted as a full presentation paper among 12 papers out of 47 submissions (26% acceptance rate). The paper was also nominated for best paper award.

My contribution: Roy Grønmo is the main author and contributor to all aspects of this paper (e.g. the ideas, the writing, the tool development, all topics of the paper), responsible for about 90% of the work.

Topic: The paper presents an aspect-oriented language to perform general transformations where both the source and target are activity models. The transformation language defines declarative rules in the familiar concrete syntax of activity models. These rules are then mapped to traditional graph transformation rules for execution in the existing AGG tool. This work has later been generalized towards general model transformations by paper 6, and the aspect language for activity models can be seen as one configuration of that framework. We also introduce the collection operator for the first time. The collection operator has later been improved and formalized within paper 5.
Overview of the Papers

Paper 2: A Semantics-based Aspect Language for Interactions with the Arbitrary Events Symbol


Publication details: The conference had an acceptance rate of 38%.

My contribution: Roy Grønmo is the main author and contributor to all aspects of this paper except the tool development (e.g. the ideas, the writing, all topics of the paper), responsible for about 70% of the work. Grønmo also contributed strongly to the tool development, where Sørensen was the main contributor.

Topic: This paper presents a semantics-based aspect language for interactions represented by UML 2 sequence diagrams. As opposed to paper 1, this paper is not a configuration of the framework for concrete syntax-based graph transformation. Instead we provide our own definitions of match and weaving based on a formal trace model for sequence diagrams. As for all the other approaches, rules are still defined declaratively following the principles of graph transformation. Also here, the rule use the concrete syntax of the modeling language which in this case is sequence diagrams. The core of the semantics-based aspect language is also described by paper 3. This paper goes beyond paper 3 by covering more details of the language including the arbitrary events symbol as a wildcard mechanism to express that matches can contain zero or more unspecified events on a sequence diagram lifeline.

Paper 3: Semantics-Based Weaving of UML Sequence Diagrams


Publication details: The conference had an acceptance rate of 34%.

My contribution: Roy Grønmo is the main author and contributor to all aspects of this paper except the tool development (e.g. the ideas, the writing, all topics of the paper), responsible for about 70% of the work. Grønmo also contributed strongly to the tool development, where Sørensen was the main contributor.

Topic: This paper shares the core of the semantics-based aspect language for interactions represented by UML 2 sequence diagrams with paper 2 (see the previous description). Paper 2 does not cover weaving of unbounded loops, which is covered here. For many typical cases the weaving can be performed on a finite structure, although the base model structure is infinite. It turns out that the matches repeat after a certain number of loops depending on the length of the pointcut. This enables us to rewrite a loop into a new finally woven loop.

Paper 4: Comparison of Three Model Transformation Languages

Roy Grønmo, Birger Møller-Pedersen, and Gøran K Olsen. In proceedings Model Driven Architecture - Foundations and Applications, 5th European Conference, pages 2-17, Lecture

**Publication details:** The conference had an acceptance rate of 30%.

**My contribution:** Roy Grønmo is the main author and contributor to all aspects of this paper (e.g. the ideas, the writing, the implementation, all topics of the paper), responsible for about 90% of the work.

**Topic:** This paper is a case study to compare our framework for general model transformations (CGT) (covered by paper 6) with two other widely used model transformation languages, based on the configuration for activity models described by paper 1. The other two transformation languages are: (1) Attributed Graph Grammar (AGG) representing traditional graph transformation, and (2) Atlas Transformation Language (ATL) representing model transformation. Our case study is a fairly complicated refactoring of UML activity models. The case study shows that CGT rules are more concise and requires considerably less effort from the modeler, than with AGG and ATL. With AGG and ATL, the transformation modeler needs access to and knowledge of the metamodel and the representation in the abstract syntax. In CGT rules on the other hand, the transformation modeler can concentrate on the familiar concrete syntax of the source and target languages.

**Paper 5: A Collection Operator for Graph Transformation**


**Publication details:** The conference had an acceptance rate of 22%. A special issue of the journal Software and Systems Modeling will contain extended versions of the best papers, where we have been invited to submit an extended version based on this paper.

**My contribution:** Roy Grønmo is the main author and contributor to all aspects of this paper (e.g. the ideas, the writing, all topics of the paper), responsible for about 90% of the work.

**Topic:** This paper improves the initial work in paper 1 on the collection operator to match and transform collections of similar subgraphs. Our original attempt in paper 1 simulated the collection free rule by a set of collection free rules. In general this becomes quite complicated and this paper proposes an improved matching and transformation. We also extend on the work in paper 1 by allowing arbitrary cardinalities, multiple collection operators in the same rule and by formalizing the matching and transformation. The matching and weaving is defined in relation to algebraic graph transformation, by dynamically instantiating a collection free rule according to the actual match size. This allows us to reuse much of the existing graph transformation apparatus.

**Paper 6: Concrete Syntax-based Graph Transformation**

Roy Grønmo and Birger Møller-Pedersen. Research Report 389, Dept. of Informatics, Univ. of Oslo, Norway.

**My contribution:** Roy Grønmo is the main author and contributor to all aspects of this paper (e.g. the ideas, the writing, the implementation, all topics of the paper), responsible for about 90% of the work.

**Topic:** This paper gives an overview picture of artefact 4 which is a framework to specify concrete syntax-based transformation rules. Such rules are compiled into traditional algebraic graph transformation rules for execution. Practical use of the framework is described
by an example transformation from feature models to business process models. The framework has been tested on a number of modeling languages and we report the major findings of these case studies.

**Paper 7: Confluence of Aspects for Sequence Diagrams**


**My contribution:** Roy Grønmo is the main author of this paper, responsible for about 70% of the work. Grønmo is the driving force behind the ideas and the main contributor of the writing. Grønmo worked out the early proof sketches. Runde was the main contributor to the final formalization and proof writing where Grønmo also contributed.

**Topic:** This paper describes artefact 3 in which we establish a confluence theory for sequence diagram-based aspects. It is proven that confluence is undecidable with high expressiveness in the aspect language. For another less expressive, but still interesting, version of the aspect language we show that confluence can be algorithmically checked. This is achieved by an extended version of a traditional critical pair analysis from term rewriting and graph transformation. An important part of the algorithm is to investigate five types of conflict situations that can make two aspect applications dependent.

**Paper 8: From Sequence Diagrams to State Machines – with help from Combined Fragments**

Roy Grønmo and Birger Møller-Pedersen. Research Report 391, Dept. of Informatics, Univ. of Oslo, Norway.

**My contribution:** Roy Grønmo is the main author and contributor to all aspects of this paper (e.g. the ideas, the writing, the implementation, all topics of the paper), responsible for about 90% of the work.

**Topic:** This paper proposes a transformation from UML sequence diagrams to UML state machines. The transformation is helpful within a described modeling process, which is based on an existing refinement theory. We take advantage of the added expressiveness in UML 2 where combined fragments (e.g. conditional behavior, loop) can be used to define more precise sequence diagrams than in previous UML versions. The main contribution of the paper is a set of transformation rules that are defined with the concrete syntax of sequence diagrams and state machines. The paper describes one configuration of the framework representing artefact 4, where we introduce tailored transformation support for the combined fragments from sequence diagrams.
Appendix B

Paper 1: Aspect Diagrams for UML Activity Models
Aspect Diagrams for UML Activity Models

Roy Grønmo1,2 and Birger Møller-Pedersen1

1 Department of Informatics, University of Oslo, Norway
2 SINTEF Information and Communication Technology, Oslo, Norway
{roygr,birger}@ifi.uio.no

Abstract. Aspect-orientation has gained increasing popularity, especially within the programming domain, with textual-based approaches such as AspectJ. Aspect-orientation provides an approach to the organization and management of code that cross-cut elements of the base program or library. Cross-cutting aspects is also an issue within the modeling domain, and it is therefore likely that modeling languages can benefit from the aspect-oriented approach. This paper proposes activity aspect diagrams for UML 2 activity models. Activity aspect diagrams are defined directly in the concrete syntax of activity models in order to enable a user-friendly way of specifying aspects. The activity aspect diagrams and base activity models are transformed into the abstract syntax of algebraic graph transformation systems, where the model weaving is carried out using the well-established AGG tool. The approach is demonstrated by two examples and a proof-of-concept aspect diagram editor has been implemented.

1 Introduction

Activity models [12] is a popular tool to model workflow systems, service-oriented models and business processes. An activity model consists of activities that are connected/linked by means of control-and data flows in a graph-layout. An activity may range from a human step such as contact-the-boss to an automated service such as a call to a Web service. Control flow includes support for sequential, choice, parallel and events. Activities may be grouped in subactivities and can be nested at arbitrary levels.

In aspect-oriented programming the base program is the main program upon which one or more aspects may define some cross-cutting code as additions or changes. An aspect is defined by a pair (pointcut and advice), where the pointcut defines where to affect the base program and the corresponding advice defines what to do in the places identified by the pointcut. Analogously we term our main activity diagrams as the base models, and we define an aspect diagram to consist of a pointcut diagram and an advice diagram, both based upon the concrete syntax of activity models. From the base model and an aspect, an aspect weaver can produce a woven result in the form of a new model.

We have chosen to use the aspect terminology instead of the more general model transformation terminology. This is because our aspects, the source models and the target models are all based upon the same language (activity models), and because we define a transformation as a pair of pointcut and advice.
The need to transform activity models include model refactoring, model checking, quality-of-service aggregation [5] etc. Another important application of aspects at the model level is to achieve good separation-of-concern. A base model may for instance model the functional parts, while a set of aspect models may define non-functional aspects such as exception handling, security and quality-of-service properties. In many situations the updated model or aggregated result shall be viewed by the modeler. In other cases the transformation may only simplify or restructure the model so that the model can be interpreted by other processing tools that require a specific structure. Assume there is a transformation script that can produce BPEL code [14] for execution, but it requires that subactivities are not used. In such a case, we may define a first transformation which removes all subactivities to a collapsed structure, and which preserves the execution semantics.

Traditional model transformation approaches suffer from being either textual and/or working at the abstract syntax also known as the metamodel level. With our proposed activity aspect diagrams, the modeler can define the model-to-model transformation rules directly upon the already familiar environment of the graphical, concrete syntax of activity models. The hypothesis is that defining graph-based transformation rules operating directly on concrete syntax would provide the transformation modeler with a better tool for defining model transformations.

The base models upon which aspects can be applied, cannot in general predict the aspects that one wants to apply. Thus, the base model specification should be independent of the aspect model specification. This is called obliviousness and is one of the key factors behind the success of AOP. We want to apply the same principle to our activity aspect diagrams. Furthermore, the aspect models should be easy to specify and understand, so that many typical cross-cutting properties are expressible in a simple manner. We propose to introduce high-level operators to be able to express transformation needs using a single rule, where multiple rules otherwise needs to be defined within traditional graph transformation approaches.

2 Examples

We will demonstrate our approach by two examples. In the first example we assume that the base activity model has been used to model a Web service composition [16]. In a service composition there are several calls to distributed services, in general provided by external parties. In such a scenario it is a typical problem that individual services become temporarily (or permanently) unavailable due to network problems, server problems etc. A service composition will fail if any of its individual services fail.

We propose an exception handler aspect, based on timeouts, to improve the reliability of the service composition. It is assumed that it is more reliable to terminate with a proper timeout message instead of being a non-responding service which only hangs. For all the services we specify a timeout value indicating
unreasonable long time to process the service. If no timeout value is defined we may use a default value such as 20 seconds. The timeout value is specified by a **tagged value** (*tagged value* is a name-value pair which is used to extend the UML metamodel to make user-defined UML profiles). Notice that by omitting all the timeout annotation and applying default timeouts to all the services, we will achieve full obliviousness if desired.

An activity model uses rounded rectangles to represent activities, diamond symbols to represent DecisionNodes (or-split) and MergeNodes (or-join), bars to represent parallel flow (and-split/and-join), a filled circle represents InitialNode, a circle with a smaller filled circle inside represents FinalNode, and directed edges represent control flow.

The left part of Figure 1 shows the proposed transformation to be applied on the base model which consists of two activities. Each time an activity is executed (call to a Web service in this case) a timer (displayed as hourglass) is started in parallel, with the timeout value taken from the activity. There is now a racing condition between these two actions, where the first one to terminate should enforce the termination of the other. This is achieved by placing an **InterruptibleRegion** (dashed rounded rectangle) around these two parallel activities. An interrupt control flow leaves the ordinary activity and will by definition terminate all other flows inside the **InterruptibleRegion**. The timer activity is immediately followed by a send timeout signal.

The timeout signal is received by a global **acceptEventAction**. One **acceptEventAction** is produced for each activity. This is to make the exception message specific to the activity which had the timeout. Thus, the **acceptEventAction** can be immediately followed by **throwException** activity which reports back the name of the activity and associated timeout value that caused the exception. The **throwException** activity is followed by a **FinalNode** which will terminate all other flows within the entire activity model.

In this example the resulting model will be very cluttered and hard to read if the timeout exception is included. In this case, the new model should not be used for viewing, only as an intermediate step prior to execution.

**The second example** is a model refactoring example taken from Eder et al. [6] who present a number of model refactoring rules for workflow models. We have adopted his **WFT-JC1** example as our second example for activity models. The right part of Figure 1 shows an example base model before the rule is applied.

---

**Fig. 1.** Examples: Exception handler (left) Redundant DecisionNode (right)
applied (top part), and after the rule has updated the model (bottom part). The rule expresses that two consecutive DecisionNodes can be merged into one by combining the guards of the first and second DecisionNodes. The result of the transformation is that the inner pair DecisionNode/MergeNode is removed and the guards are joined by an AND-operator.

The example is shown using two alternative paths inside the inner DecisionNode, but a solution should be capable of handling an arbitrary number of alternative paths. We do however restrict the example, so that only a single activity is allowed within each alternative path of the inner DecisionNode/MergeNode pair.

3 Architecture of the Approach

Figure 2 shows the architecture of our approach. The base model is specified within an existing UML 2 activity modeling tool. Our proposed graphical language, called activity aspect diagrams, define aspects to be applied on activity models. To support the approach we need to develop a new editor for the activity aspect diagrams. One or more activity aspect diagrams may apply to the same base model. An activity aspect diagram uses the concrete syntax, in this case activity models, and it is based upon algebraic graph transformation.

Since both the base model and the transformation rules are defined using a concrete syntax, one cannot directly use existing graph transformation tools, as these are based upon transformations on abstract syntax. So, in order to perform rule analysis (correctness, termination, confluence) and the actual weaving, we must either implement all this from scratch, or provide a mapping between the concrete and abstract syntax. We choose the latter to benefit from existing well-established graph transformation tools.

We need to transform UML 2 activity models into graph representation and back again. The graph representation will be a typed attributed graph, where nodes and edges are assigned to types, and the nodes and edges can have associated attributes. Similarly the aspect diagrams need to be transformed into graph transformation rules. The transformations from the concrete syntax of base models and aspect model should be fully automatic, and the modeler should not need to see or worry about the graph transformation tool operating on the abstract syntax.

Fig. 2. Approach: From aspect diagrams to graph transformation rules
The graph transformation tool performs the weaving by applying the generated transformation rules on the generated abstract syntax representation of the base model. The result is an abstract syntax representation of the new activity model. The new model will be translated back to concrete syntax and presented to the user in a UML modeling tool.

4 Activity Aspect Diagrams

The activity aspect diagram consists of two parts: pointcut diagram and advice diagram. The pointcut diagram is shown on the left hand side and the advice diagram is shown on the right hand side. The weaving semantics of the aspect diagram follows the basic principles of a traditional graph transformation system, where the pointcut diagram models an activity fragment for which we are looking for potential matches (often referred to as a morphism within graph theory) within the base model. The advice diagram instructs how a base model shall be changed relative to the match. We require that matches of the pointcut are injective, meaning that every separate element defined in the pointcut needs to be mapped to separate elements in the match.

Elements appearing in the pointcut and not in the advice are to be deleted, while elements appearing only in the advice, are to be added. Elements appearing in both the pointcut and advice are to be unchanged or they may change their properties or relationships to other elements. Furthermore we adopt the double-pushout approach which excludes application of rules deleting nodes that are attached via edges to nodes in the remaining graph. The precise meaning of this depends on the mapping from concrete syntax to abstract syntax (section 6).

Elements will only be matched if they have the exact same context in the base model as within the pointcut diagram. Thus all relations need to be present also in the base model. Assume that the pointcut diagram defines an activity with an attached note and a single outgoing control flow reaching the finalNode. In such a case both these relations need to be associated with the matching activity within the base model. The base model element may however have additional relations and still be a match, such as incoming control flow (most likely) and data flow leading into its input pins.

Both the pointcut and advice modeling languages build upon activity models. In the sequel of this paper we will use the shorter term 'aspect diagram' for 'activity aspect diagram' as the context is given to be activity models.

In the simplest form a pointcut diagram is an ordinary activity model fragment. A pointcut diagram extends activity models with property matching expressions. The property matching expression goes into the exact same place as the corresponding property of an ordinary activity model. Properties in an activity model include names, stereotypes, tagged values, guard expressions etc. Each property matching can use any legal Java string expression combined with the two wildcards (*, ?). The star matches an arbitrary sequence of characters and the question mark matches any single character.
Identifiers are defined with a question mark prefix and may be used to identify both elements and properties. The identifiers and property matching are combined in a syntactic pair such as ?actId «Service» placed in the name property of an activity. actId will be the identifier of a matching activity that has a name ending with Service. Identifiers in the name position is by default an element identifier.

Property assignment is available to update the property values in the advice. A property assignment is defined directly at the value place of the the property, and it can be any legal Java expression that evaluates to a string. The Java expression can use the identifiers as variables holding the value of properties matched by the pointcut. An identifier will be bound to the matched property value (the name in case of an element identifier).

We extend the pointcut modeling language with boolean operators (not applicable to the advice diagram). All selection elements in the pointcut model are implicitly joined by and-operators. In addition there are or-, xor- and not-operators available to use in other cases. These are displayed as {operator} and are attached to its operands via dotted lines. The not-operator has one or more operands, while the others have two or more operands. One element can only be an operand of one operator.

The boolean operators may also be used as part of the property matching expression. Example: We want to match all Activities with stereotype Service or Webservice. This can be expressed as «Service» {or} «Webservice».

In order to make the aspect diagrams better suited to specify transformation rules in a simple manner, we propose to introduce a few, but powerful high-level operators. We will see in the transformation section that the use of high-level operators typically needs to be translated into a set of basic graph transformation rules. This also motivates the use of such high-level operators, because the rule modeler can define a single rule using a powerful, but intuitive high-level operator instead of defining several basic transformation rules. Due to limited space, we will only present one high-level operator, the collection operator, in this paper.

An aspect model consists of a set of aspect diagrams which can be non-deterministically applied or they may be applied according to some rule control structure. To define the control structure of the rules, one alternative is to use activity models.

5 Aspect Diagram Examples

After having introduced the aspect diagram language, we will now show how aspect diagrams can solve the two examples from section 2. Before solving the timeout exception handler case, we assume for simplicity that an activity can have at most one incoming control flow and at most one outgoing control flow. This is without loss of generality, since many incoming control flows represent an implicit join node, while many outgoing control flows represent an implicit fork node. Thus, any activity model can be translated into a semantically equivalent
activity model with our proposed restriction. In fact, the modeler may quite easily use aspect diagrams to define such a translation.

We also assume that each activity has an associated tagged value, \texttt{timeout}, indicating the timeout of the service it invokes. Again, an aspect diagram could easily introduce this tagged value with a default value for all activities without such annotation.

The left-most aspect diagram in Figure 3 shows the proposed solution. For each activity with incoming and outgoing control flow and a timeout value (shown at the top of the pointcut), we add all the structure necessary to introduce the exception handling. The pointcut definition includes the usage of two \texttt{not}-operators attached to control flows of the activity identified as \texttt{?outermost}, which ensure that no incoming and no outgoing control flow is associated with the outermost activity. This condition will only hold for the outermost activity. By placing all the exception handlers in the outermost activity we will terminate all flows within the entire activity model when we go to the finalNode.

We need to ensure the aspect is only applied once for each activity. The throw event signal named \texttt{timeout} + \texttt{?S} (\texttt{?S} is bound to \texttt{activityName}) is added by the advice. The same signal is therefore added to the pointcut with a \texttt{not}-operator attached to it, meaning that it cannot be present for the aspect to be applied. The aspect should only apply to activities that are leaf activities and not subactivities. For simplicity we assume that only leaf activities have the timeout values. If this condition does not hold, we could strengthen the aspect diagram by inserting an initialNode connected to a \texttt{not}-operator inside the \texttt{?S}-activity.

In the second example we introduce the high-level collection operator. Remember that in the example there is an inner, redundant \texttt{decisionNode}, which can have an arbitrary number of alternative paths. The right-most aspect diagram in Figure 3 shows usage of the collection operator, where the dotted rectangle surrounds the collection elements and a cardinality is provided next.
to it. The cardinality has the same form as ordinary UML cardinalities. The elements inside the rectangle of the collection operator indicate that there may be an arbitrary (cardinality in this case is 1..*) number of matches for this part. Each match however, needs to be linked to the rest of the graph exactly as specified by the relationships to the parts outside of the collection rectangle. This means that the same DecisionNode C2 is linked to all the matches within the collection operator.

If the collection operator had been extended to include the DecisionNode C2, then we would have had distinct DecisionNodes for all our matches (remember that the matches are injective, which also applies to the collection operator matches). Identifiers inside the collection pattern denote different elements or properties for each match, such as the guard in the example. The collection operator is normally used also in the advice to indicate the changes to the matches in the collection. If the collection operator is absent in the advice, it implies a request to delete all the collection elements. No boolean operators are allowed inside the collection operator. Within this paper we also assume that only a single collection operator is used within the same aspect diagram. By this we avoid a lot of complexity which we do not have space to cover here.

Figure 4 shows the relationship between the collection operator in the pointcut and possible matches. For the illustration only we use a circle to denote some element A and B (eg. InitialNode, DecisionNode, JoinNode) that can be the source or target of a control flow edge. In case a) only the edge is inside the collection while the source and target elements are outside the collection operator. This means that possible matches will have a set of edges between the same A and B elements. In case b) the source A is also inside the collection which means that a match will contain a set of distinct A elements with edges leading to the same B target element. In case c), both the source and target is inside the collection, which means that a match will contain a set of distinct A and B elements each having their own control flow. We require that a collection match shall be maximal, meaning that the largest set of elements, limited by the upper bound cardinality, must be gathered before the advice is applied to the match.

6 Transformation between Concrete and Abstract Syntax

The AGG tool [18] is chosen as the graph transformation tool, and parts of the mapping is tailored for this purpose. We need to transform both ways between the concrete syntax of activity models and the abstract syntax of graphs. For
this purpose we define a one-to-one correspondence, which is quite straightforward for activity models since they are very close in nature with graph representation.

Activity, InitialNode, FinalNode, DecisionNode, MergeNode, ForkNode, JoinNode and data objects appear as nodes in the activity model and we choose to represent these as nodes (with different types) also in the graph representation. The control and data flow edges of activity models are also represented as nodes (with different types), in the graph representation, with two directed outgoing edges labelled src and trg. By this circumstantial mapping of the activity model edges, missing edge sources or targets, at the concrete syntax, will be translated into rules where the source and target of an edge are always present at the abstract syntax. We discuss this further in section 7 after all the transformation rules are presented.

Properties of the different UML types are mapped to node attributes of the corresponding graph node. An activity name is mapped to a name attribute belonging to the activity graph node, while a control flow guard is mapped to a guard attribute of the control flow graph node. The definition below provides the one-to-one relation between Activity model and graph representation.

The operator $\leftrightarrow$ defines the one-to-one relation between Activity model elements and graph elements. It uses an overloaded mapping function $\phi$ which is either $\phi : Id \rightarrow Id$ or $\phi : Attrs \rightarrow Attrs$, to map from activity element ids/properties to graph ids/attributes. To the left we show the Activity elements and on the right we show the corresponding graph elements. Nodes are given by a triple $(Id, Type, Attrs)$ and edges are given by a quintuple $(Id, Type, srcId, trgId, Attrs)$:

$$\text{Activity}(aId, attrs) \leftrightarrow \text{node}(\phi(aId), "Activity", \phi(attrs))$$

$$\text{cFlow}(cId, AId, BId, attrs) \leftrightarrow \begin{cases} 
\text{node}(\phi(cId), "cFlow", \phi(attrs)) \\
\text{edge}(\text{genId}(), "src", \phi(cId), \phi(AId), \epsilon) \\
\text{edge}(\text{genId}(), "trg", \phi(cId), \phi(BId), \epsilon)
\end{cases}$$

The mapping of InitialNode, MergeNode, ForkNode, JoinNode, DecisionNode and FinalNode is similar to the Activity mapping, and the mapping of dFlow is similar to the cFlow mapping. $\epsilon$ denotes an empty set of attributes, Ids are suffixed by Id, and genId() makes a new id. Due to limited space, we do not present a full mapping of the activity models as graph representation within this paper.

We define a one-way transformation from the aspect diagrams to graph transformation rules in the abstract syntax. Often a single aspect diagram will be mapped to several graph transformation rules. Since the aspect diagrams are designed as extended activity models, the mapping concerning the activity models can follow the mapping defined by $\leftrightarrow$. In an aspect diagram without high level operators, the pointcut diagram will be mapped to the left part(s) of one or more graph transformation rule(s), and the advice diagram will be mapped to the right part(s).
For the property matching and assignment, the identifiers are mapped to AGG identifiers and variables, while Java string expressions can be read directly by AGG. We omit wildcard expressions (\ast,\?) since it is not supported by AGG.

We need to map the boolean operators of the pointcut language. All elements not explicitly defined as an operand, will implicitly belong to the global and-operator. This is directly supported by normal graph transformation rules, and no additional mapping is needed.

Each not-operator will be mapped to a Negative Application Condition (NAC) associated with the corresponding mapped rule of the aspect diagram. The occurrence of a matching negative application condition in combination with left part matches of the rule, prevents the application of the rule. Algorithm 6.1 shows pseudocode on how to map a single not-operator into a NAC. The not-operands will be removed from the pointcut and inserted into a NAC rule instead. If the not-operand is an edge, then its source and target nodes (retrieved by the directAssoc method) will be copied into the NAC rule (and not moved from the pointcut). The rules left, right and NAC definitions are finalised by translating to abstract syntax with the toAbsSyntax method.

Algorithm 6.1: TransformNotOper(AD : aspectDiagram)

\[
\text{notExpr} = AD\text{.getNotExpr}; \quad \text{NAC} = \text{new NACRule}
\]
\[
\text{for } i \leftarrow 1 \text{ to } \text{notExpr.numOperands} \text{ do }
\]
\[
\begin{cases} 
\text{notElem} = \text{notExpr.operand}(i) \\
\text{AD.pointcut.remove(notElem)} \\
\text{NAC.add(notElem + notElem.directAssoc)}
\end{cases}
\]
\[
\text{NewRule} = \text{new Rule}; \quad \text{NewRule.left} = \text{AD.pointcut.toAbsSyntax} \\
\text{NewRule.right} = \text{AD.advice.toAbsSyntax} \\
\text{NewRule.addNAC(NAC.toAbsSyntax)}
\]

The three not-operators in the exception handler aspect diagram of Figure 3 will be mapped to three different NAC rules associated with a single graph transformation rule. The first NAC will ensure that the rule is only applied once (timeout signal as not-operand), and the latter two not-operands ensure that the outermost activity will not have incoming nor outgoing control flow.

The or-operator leads to several copies of the rule, one for each operand of the or-operator. Algorithm 6.2 shows pseudocode for an aspect diagram with a single or-operator. outsideOrExpr retrieves all elements that are not part of the or-expression.

An xor-operator will be mapped in the same way as an or-operator with a special metanode to be produced by each rule generated by the xor-expression. The metanode acts as a flag to indicate that one of the rules has been applied. The metanode is added as a NAC rule associated with each of the rules generated for the xor-expression, which ensures that at most one of the xor-rules will be performed.
Algorithm 6.2: TransformOrOper(AD : aspectDiagram)

\[
\begin{align*}
\text{orExpr} &= \text{AD.pointcut.getOrExpr} \\
\text{for } i &\leftarrow 1 \text{ to orExpr.numOperands} \\
&\quad \begin{cases} 
\text{NewRule} = \text{new Rule} \\
\quad \text{NewRule.left} = \text{orExpr.operand}(i).\text{toAbsSyntax} + \\
\qquad \text{AD.pointcut.outsideOrExpr.\text{toAbsSyntax}} \\
\quad \text{NewRule.right} = \text{AD.advice.\text{toAbsSyntax}} \\
\quad \text{AllRules.addRule(NewRule)}
\end{cases}
\end{align*}
\]

Now we map the high-level collection operator. For simplicity we assume that only two cardinalities are available for the collection operator: 0..* or 1..*. The collection operator can be represented by an ITER-rule and a FINAL-rule.

**ITER-rule. Intention:** The iteration rule shall be applied to a single match in a collection of matches and it shall be applied for as long as possible. That means that it shall be applied the same number of times as there are individual matches of the collection pattern. **Mapping:** Simply remove the collection operator marking (the rectangle and cardinality) in the pointcut. Construct the advice by combining the outside of the collection operator from the original pointcut with the inside of the collection operator of the original advice. This will ensure that the necessary changes are applied to each match of the collection. At the same time we preserve the elements outside the collection rectangle, so that all individual collection matches get an equal chance to be matched. To preserve the intended semantics at the end, we let the FINAL-rule sort this out.

**FINAL-rule. Intention:** The final rule shall be applied only once after all matches in the collection have been applied with the ITER-rule. **Mapping:** Both the pointcut and advice is constructed by removing both the collection operator (the rectangle and cardinality) including its inside content. This will result in a rule that finally does all the adding and deletion of elements outside of the collection part. These changes cannot take place before all possible applications of the ITER-rule.

The transformation of the collection operator can be summarized by the pseudocode of algorithm 6.3. The algorithm assumes there is exactly one collection operator in the input. The toAbstractSyntax method will transform from concrete to abstract syntax. In this step the inside of the collection operator will contain all node elements resulting from any elements inside the collection of the concrete syntax. The removeCollOper method will remove the Collection operator but keep all the elements inside and outside of the collection. The outsideCollection and insideCollection will keep only elements outside or inside the collection operator. The + operator produces a new graph where the node/edge set is the union of the nodes/edges of the operands.
Algorithm 6.3: TransformCollOper(AD : aspectDiagram)

\[
AD = AD.\text{ToAbsSyntax}; \text{Iter} = \text{new} \ \text{Rule}; \text{Final} = \text{new} \ \text{Rule}
\]
\[
\text{Iter.left} = AD.\text{pointcut}.\text{removeCollOper} \\
\text{Iter.right} = AD.\text{pointcut}.\text{outsideCollection} + AD.\text{advice}.\text{insideCollection} \\
\text{Final.left} = AD.\text{pointcut}.\text{outsideCollection} \\
\text{Final.right} = AD.\text{advice}.\text{outsideCollection}
\]

Figure 5. Aspect diagram of redundant decisionNode example mapped to AGG rules

Figure 5 shows the result of transforming the redundant DecisionNode aspect diagram into AGG graph transformation rules. An aspect diagram may have an arbitrary number of potential matches in the base model. We need to ensure that all these matches are found by forcing the ITER-rule and FINAL-rule to be repeated as long as possible. By doing so we will achieve the desired behavior as if the pointcut had a maximal matching (section 5) upon which the advice was applied. The top of Figure 5 shows how the rule control structure can be defined using activity models. AGG does not support all the control structure power of activity models, so in general we need a scheduler component on top of AGG. The AGG layered approach will however support the two paper
examples. Notice that the gluing condition (double-pushout) ensures that the example FINAL-rule is applied after all corresponding ITER-rule applications are finished, but this is not the general case.

7 Discussion

An activity model looks quite like an abstract syntax graph representation with nodes and directed edges, and one could question if the abstract syntax could directly represent the concrete syntax. However, since activity models have elements that contain other elements (e.g. subactivity, expansionRegion and interruptibleRegion), this is not possible. In addition, activity inputPins and outputPins pose a problem as they have a specialized notation in that they are displayed on the border of the owning activity.

Representing control flows as edges also in the abstract syntax would make things much more difficult. It would not be possible to define a rule without the source or target of a control flow edge. In the aspect diagram, however, we use explicitly that a missing source or target expresses a wildcard node. Without this possibility in the exception handler example, we would have to explicitly express all the different node options to be source of the ?in-labeled edge, and similarly for the target of the ?out-labeled edge. We could at least have the options of InitialNode/FinalNode (?in/?out), activity, subactivity, ForkNode/JoinNode (?out/?in) and DecisionNode/MergeNode (?out/?in). To cope with all combinations we would have to make 25 (5x5) different graph transformation rules.

An interesting question is to what extent our approach can be generalized: Is the approach also appropriate for other kinds of models than activity models, so that we could introduce the same kind of aspect diagrams for UML sequence diagrams, class diagrams etc.? This remains to be investigated, but an observation is that the closer the modeling language concrete syntax is to a typed attributed labelled graph, the easier it will be to follow the graph transformation approach. Class diagrams are close to such graphs, while sequence diagrams are different kinds of graphs. In a mapping from the concrete syntax of a sequence diagram into an abstract syntax as graphs, we get an explosion in the number of elements. This means that it is highly questionable if the approach, presented in this paper, is suitable for sequence diagrams.

The property matching, property assignment, boolean operators and rule control structure are general mechanisms and should be applicable to other kinds of models. The generality of high-level operators, and which high-level operators are needed, may vary with the kind of model. To achieve an intuitive and easy to comprehend aspect diagram language, we believe it should be tailored to the actual source and target modeling language.

We have developed a proof-of-concept Eclipse GMF-based [3] editor for the aspect diagrams. It currently supports Activity, DecisionNode and MergeNode of Activity models, in addition to the use of single collection operator, which was enough to successfully demonstrate the redundant DecisionNode example.
The transformation from aspect diagrams to AGG rules has been implemented using the MOFScript language ([11]). For other examples, including the second paper example, we have manually followed the mapping definitions described between abstract and concrete syntax, and tested the graph transformations in AGG upon several base activity models with successful results.

8 Related Work

Several approaches (QVT [13], Zhang et al. [20], graph transformation approaches such as AGG [18], Ehrig et al. [4] and PROGRESS [15]) provide model transformation languages and tools that can define transformations between general source and target modeling languages, and where one transformation may operate on different source and target modeling languages. In cases where the source and target languages are both activity models, they suffer from using abstract syntax instead of the more intuitive concrete syntax on which our aspect diagrams are defined.

Lindqvist et al. [9] propose the star operator which can be used in a pointcut language to find repetitive occurrences of a specific modeling pattern and is complementary to our collection operator. The star operator is limited to repetitive occurrences that constitute a sequential path, and is thus not strong enough to model the collection operator. It is only proposed within a pure query part (like our pointcut), and has no associated advice part as we have defined. Furthermore, the star operator is presented on abstract syntax only. However, the authors share our opinion with respect to concrete syntax: In a tool environment, however, creating the queries using the concrete syntax of the modeling language can be beneficial.

Aspect-oriented behavior modeling approaches so far have been dominated by UML sequence diagram attempts [19] [1] [2] [17]. Solberg et al. [8] and Whittle and Araújo [19] perform weaving at the model level as in our approach. Deubler et al. [2] and Stein et al. [17] use sequence diagrams to model the aspects at a conceptual level to be mapped to some aspect programming language such as AspectJ [7]. Stein et al. [17] focus only on the conceptual modeling of the pointcut, and do not cover advice modeling. Klein et al. [8] propose semantic-based weaving of Hierarchical Message Sequence Charts (close to UML sequence diagrams). Their approach focuses on the weaving algorithm that takes the execution semantics into account in the weaving process. This is a benefit compared to our approach since the aspect diagrams and the graph transformation system performs pure syntax-based weaving. To illustrate the aspect definition, they provide an example of a pointcut and advice which is similar to the graph transformation principle aspect diagrams are built upon. However, the aspect definition given only explains the example, and no further attempt to define an aspect-oriented modeling specification language is given.

Mehner et al. [10] analyzes if a set of aspects may be properly woven with the base model by considering possible conflicts and dependencies. Pre- and
post-conditions expresses the effects of each activity in the AGG tool where automated analysis is carried out. The aspect definitions proposed in their paper are limited to inserting an entire new use case before, after or as a replacement of some previous activity. None of our two example aspects are expressible with this definition.

9 Conclusions and Future Work

We have proposed activity aspect diagrams as a way to define aspects upon activity models. The approach is built upon transformation of both activity- and activity aspect diagrams in concrete syntax to an abstract syntax of a graph transformation tool, which then performs the weaving. A major benefit is that the aspect modeler can operate directly within the familiar syntax of activity models instead of the more general graph transformation rules of traditional graph transformation approaches. At the same time we can benefit from analysis of confluence and termination in the graph transformation tool.

Transformation approaches have been dominated by textual languages, even though the source and target languages may be graphical languages. One reason is that textual programming languages have been widely used for decades with lots of practical experience and improvements. While the earlier attempts used low-level, non-comprehensible constructs, todays textual programming languages use several high-level constructs (e.g. while-loops, inheritance, recursion) to allow for user-friendly programming. By introducing high-level operators also for graphical transformation languages, we believe that the graphical languages can learn from the history of textual programming language development. One such high-level operator, the collection operator, has been introduced in this paper to demonstrate the principle.

As future work we may look into nested collection operators, and investigate more practical examples to see if we need additional high-level operators. We would also like to explore the relationship between termination and confluence criteria at the concrete vs. the abstract representation of the transformation rules.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project).

References
Appendix C

Paper 2: A Semantics-based Aspect Language for Interactions with the Arbitrary Events Symbol
A Semantics-Based Aspect Language for Interactions with the Arbitrary Events Symbol

Roy Grønmo\textsuperscript{1,2}, Fredrik Sørensen\textsuperscript{1}, Birger Møller-Pedersen\textsuperscript{1}, and Stein Krogdahl\textsuperscript{1}

\textsuperscript{1} Univ. of Oslo, Dept. of Informatics, Norway
\textsuperscript{2} SINTEF ICT, Norway
\{roygr,fredrs,birger,steinkr\}@ifi.uio.no

Abstract. In this paper we introduce an aspect language that can define cross-cutting effects on a set of UML 2.0 sequence diagrams. The aspects and sequence diagrams are woven at the model level. By basing the weaving upon a formal trace model for sequence diagrams, we ensure that the weaving is semantics-based. We propose the arbitrary events symbol as a wildcard mechanism to express zero or more events on a sequence diagram lifeline. The approach is explained by a real-life example, and a weaving tool partially implements the approach.

1 Introduction

Aspect-orientation for programming has emerged as a promising way to separately define cross-cutting parts of programs, in order to achieve separation of concern. We believe that the same potential is there also for modeling. This paper explores aspect-oriented modeling for UML 2 sequence diagrams [14].

In aspect-oriented programming the base program is the main program upon which one or more aspects may define some cross-cutting code as additions or changes. An aspect is defined by a pair (pointcut and advice), where the pointcut defines where to affect the base program and the corresponding advice defines what to do in the places identified by the pointcut. Analogously we term our set of sequence diagrams as the base model, and we define an aspect diagram to consist of a pointcut diagram and an advice diagram, both based upon the concrete syntax of sequence diagrams.

In this paper we assume that the sequence diagrams are used to automatically produce executable test code, e.g. to test if a sequence diagram is a correct refinement of another sequence diagram [12], or to test if a system specified by UML statecharts, class diagrams and object diagrams is consistent with sequence diagram specifications [15]. Thus, we need to weave the aspect and the base model before generation of test code. The aspect diagram defines the cross-cutting model elements to influence the base model, so that an aspect weaver can produce a new model which is the base model woven with the advice.

The woven model is not intended to be viewed (except for debugging) or further updated by the modeler. This means that the structure of the result is not a primary focus. It suffices that the woven model is correct with respect to our formal model.

Many aspect-oriented approaches suffer because they rely on a pure syntactic pointcut matching and weaving. Syntactic-based matching has a problem because it does not
capture all the matches it conceptually should do. The pointcut of Figure 1 expresses that the message m1 from the lifeline L1 to the lifeline L2 is followed by the message m2 from L1 to L2. The base model has two consecutive alt operators. An alt operator defines a choice of different alternatives, where the alternatives are given as operands separated by a dashed line. If we try to find matches of the pointcut within the base model with pure syntactic matching, then we do not find any matches. However, one possible execution trace chooses the second operands of the two alt operators, which then should result in a match of the specified pointcut.

Our main contribution consists of: 1) an aspect language which is sufficient to handle our case study, and 2) semantics-based matching and weaving. We devote much attention to the arbitrary events symbol that makes it possible to define pointcuts with an arbitrary number of events in specific positions. This symbol enables us 1) to make the pointcut more robust with respect to some kinds of changes in the base model, and 2) to define more relaxed matching criteria leading to additional matches in cases with irrelevant base model differences.

The paper is organized as follows; Section 2 introduces the STAIRS formal model for sequence diagrams in relation to our case study; Section 3 presents the aspect diagram language including the arbitrary events symbol; Section 4 defines the semantics-based matching; Section 5 explains how the weaving algorithm works; Section 6 compares our approach with related work; and finally Section 7 provides the conclusions and suggests future work.

2 Sequence Diagrams and STAIRS

STAIRS defines a formal model where the semantics of a sequence diagram is understood as a set of execution traces. The syntax of a UML sequence diagram, called interaction, follows the EBNF in the right part of Figure 1 [16]. We focus on the operators seq and alt. These two operators are chosen because they are the basic operators from which we also may define several other operators.

Each message is represented by two events, a transmission event (!) and a reception event (?) (the transmitter and receiver lifelines are omitted for readability when this information is unambiguously defined by associated diagrams). An event takes place on a lifeline L1 if it is a transmission event on L1, e.g. !(signal,L1,L2), or a reception event on L1, i.e. ?(signal,L2,L1). We require that the messages are complete (i.e. contain both events ! and ?) within each alt operand, pointcut, advice and each sequence diagram in the base model.
The weak sequence operator, \( \text{seq} \), of sequence diagrams imposes a partial order of events given by: 1) the transmission event must come before the reception event of the same message, and 2) all events are ordered for each lifeline in isolation. An intuitive idea behind this partial order is that messages are sent asynchronously and that they may happen in any order on different lifelines, but sequentially on the same lifeline. The \( \text{alt} \) operator defines alternative interactions.

Sequence diagrams do allow crossing messages, i.e., two messages \( a \) and \( b \) are crossing only when they have events on the same two lifelines, and \( a \) has an event before \( b \) on one lifeline, while \( b \) has an event before \( a \) on the other lifeline, e.g. \( \text{seq} [ !(a, L1, L2), !(b, L1, L2), ?(b, L1, L2), ?(a, L1, L2)] \).

Figure 2 shows an extract of a base model sequence diagram for the ICU (I see you) buddy system. Prototype implementations have been used as test cases within computer science courses at the University of Oslo. ICU is an SMS-based service where the users have access to different positioning services including the positioning of users who have accepted to be on the buddy list. The services have been partially specified by sequence diagrams. The sequence diagrams have been manually interpreted to produce conforming state machines for which Java code has been produced by the JavaFrame code generation framework [5]. Future versions may automate the production of state machines from sequence diagrams [10].

There are four diagram levels due to lifeline decomposition of which we will concentrate on the first two levels as this will be sufficient to explain our approach. The number of decomposition levels for a particular scenario is chosen by the modeler.

In the position user service (Figure 2), an SMS message is sent from User to PATS, at level 1, to request the positioning of a buddy. The message is forwarded from PATS to the decomposed (indicated by the keyword \( \text{ref} \)) lifeline PosUser. In the decomposed diagram, at level 2, we see that the same message is received by the Request lifeline. The Request lifeline translates the message into an internal message, posUser, which is sent to the decomposed lifeline Core. Its internal lifelines and the messages at level 3 and 4 are omitted from the illustration. The posUser message will finally reach the proper session object at level 4. The session object will initiate a message, posRequest, to the positioning service PATS via Response, to position the buddy. The fourth lifeline at level 2 without any messages in the figure, DB, stores all the persistent data and provides querying services upon these data. The syntax representation for the position user service is shown in the middle part of Figure 2.
We briefly explain the semantics operator, $\llbracket \llbracket$, while a precise definition is given in [16]. The semantics of an interaction $i$ is $\llbracket i \rrbracket = (p, n)$, where $p$ is the set of positive traces and $n$ is the set of negative traces. Positive traces define valid behavior and negative traces define invalid behavior, while all other traces are defined as inconclusive. In this paper we concentrate on aspects that only affect positive traces, and we therefore use a simplified model without negative traces: that is $\llbracket i \rrbracket = p$. A trace is a sequence of events which we display as $(e_1, \ldots, e_n)$, where $e_i$ are events for all $i \in 1..n$.

The $\llbracket \llbracket$ operator produces one trace for each valid permutation of events that satisfy the two partial order requirements as explained for the seq operator above. The $\llbracket \llbracket$ operator produces the union of the traces for each operand of the $\llbracket \llbracket$ operator. Each message in the trace is dynamically given a unique identifier, which is shared between the transmission and reception events of the message.

We define one trace to be partial order equivalent (POE) to another trace if they are both permutations of the same set of events with the same order on each lifeline. The $\llbracket \llbracket$ operator is used to define POE since it is defined to produce all such permutations:

**Definition 1.** We say that two traces $t^A = (t^A_1, \ldots, t^A_n)$ and $t^B = (t^B_1, \ldots, t^B_n)$ are partial order equivalent (POE) if and only if:

$$\llbracket \text{seq}\left[ t^A_1, \ldots, t^A_n \right] \rrbracket = \llbracket \text{seq}\left[ t^B_1, \ldots, t^B_n \right] \rrbracket$$

We let the function, $\text{POE}: \text{Trace} \rightarrow \text{TraceSet}$, calculate all the POE traces of a given trace, $\text{POE}(\langle t^A_1, \ldots, t^A_n \rangle) = \llbracket \text{seq}\left[ t^A_1, \ldots, t^A_n \right] \rrbracket$. The pointcut diagram in Figure 1 has two traces which are POE: $\langle !m_1, ?m_1, !m_2, ?m_2 \rangle$ and $\langle !m_1, !m_2, ?m_1, ?m_2 \rangle$. The function $\text{POE}(\cdot)$ of either of these two traces returns the set of both traces.

### 3 Aspect Diagram

The aspect diagrams are inspired by graph transformation [3] where the left part, the pointcut diagram, defines a pattern for which we are looking for matches or morphisms in the base model. The right part, the advice diagram, defines a replacement of the matches within the base model. This implies that messages present only in the pointcut and not in the advice, will be deleted, while messages present only in the advice and not in the pointcut, will be added. Both the pointcut and advice diagrams are based upon the graphical elements of sequence diagrams so that the modeler can think in terms of an already familiar notation.

Property names (lifelines, messages or message parameters) to be matched in the pointcut may be defined by its full name or by a mixture of fixed characters and the wildcard symbol *: Identifier variables may be shown explicitly in the diagram as part of the property name: $\#id: \text{propertyName}$. The $\#id$ will be bound to the actual identifier of a matching element. Ids that are repeated in the advice must be placed on an identical kind of element, so it is not allowed to repeat the id from a pointcut message in the advice and to change either the message name, transmission or reception lifeline. Repeated elements in the advice may use the full form $\#id: \text{propertyName}$, where all parts need to be identical from the pointcut, or we may as a shorthand omit either the
id part or the property name. We allow to use messages with signatures equivalent to those defined in AspectJ [7] where ( . . ) means an arbitrary number of parameters.

The pointcut diagram is restricted so that it can only use events and the seq operator. We avoid the alt operator in the pointcut since it is not obvious how to combine this with the advice. To avoid possible confusion for the aspect modeler, we require that such alternatives are modeled explicitly by several aspect diagrams instead of a single, more compact aspect diagram.

We continue the presentation of the aspect diagram language in two subsections. First we show an aspect diagram example for our case study, and then we introduce the arbitrary events symbol.

3.1 Example

We will now investigate the aspect model example in Figure 3 which shall ensure that the user is registered in the ICU system before using the available services. This is a cross-cutting functionality since it shall be applied to all the ICU services (including the position user service in Figure 2), except the register user service. So instead of augmenting all but one of the sequence diagrams at four levels for each of the services (about 10 services have been used in the ICU system), we would like to apply a single aspect definition for easier specification and maintenance.

The aspect diagram will follow the same decomposition principles of ordinary sequence diagrams, so that we may define a pair of pointcut and advice for each level in the base model. In the figure we only show the first two levels of aspect diagrams (as we did with the base model).

The pointcut at level 1 defines that we are looking for matches of all incoming SMS messages with any content (wildcard * ) except for the service to register the user. The exception is defined as a negative pointcut (called Negative Application Condition (NAC) in graph transformation) where the #smsIn identifier of the pointcut is reused from the ordinary pointcut to restrict the possible matches. There may be an arbitrary number of negative pointcut diagrams associated with each level of pointcut diagrams. Pointcut matches are excluded if at least one negative pointcut also provides a match.

The two messages identified by the pointcut (#smsIn and #smsIn2) are repeated in the advice and will thus remain in the result. The advice contains an alt operator which shall be added to the base model.

We propose a new operator, insertRest, which in the example is placed in the else operand of the alt operator. The insertRest operator defines that the remaining part of the base model shall go into the alt operand, overriding the default behavior of being placed after the full alt operator. An insertRest operator must span over all lifelines and may be placed anywhere in the advice. Multiple insertRest operators are allowed in the advice.

Notice that there are two messages to be added by the advice diagram at level 2, isUserReg and its reply, which were not visible at level 1 since they are only between internal lifelines. Wildcard matching symbols * are used in all places where the base sequence diagrams have different values to be matched: the content of the incoming SMS messages (#smsIn and #smsIn2), the name of the decomposed lifeline (#icu), and the internal message (#service) initiated by the SMS message.
Our aspect diagrams follow ordinary UML decomposition rules. The alt of the advice introduced at level 1 that spans over the decomposed lifeline, must be repeated at all the lower decomposition levels. A reception event on a decomposed lifeline has the decomposed diagram frame, in the next level, as its source, and one of its internal lifelines as its target, e.g. ?(#smsIn2,PATS,#icu) at level 1 corresponds with !(#smsIn2,#icu,Request) at level 2. A transmission event on a decomposed lifeline has the decomposed diagram frame, in the next level, as its target, and one of its internal lifelines as its source, e.g. !(sms("Not registered"),#icu,PATS) at level 1 corresponds with ?(sms("Not registered"),Response,#icu) at level 2. The ordering of events on the decomposed lifeline must be maintained within the decomposed diagram, e.g. #smsIn2 must come before the alt operator both at level 1 and 2.

3.2 Arbitrary Events Symbol

Our pointcut in Figure 3 defines that there are some events that have to happen directly after each other: ?(#smsIn) followed by !(#smsIn2) on the PATS lifeline at level 1, and ?(#smsIn2) followed by !(#service) on the Request lifeline at level 2. These positions in the pointcut are fragile with respect to base model evolutions. Consider that we decide to send a request confirmation back to the user directly after each incoming SMS message on the PATS lifeline. Such a base model change may be registered directly in the base model or as another aspect. In either case there will be an intermediate event in between ?(#smsIn) and !(#smsIn2) which unintentionally prevents our aspect to be applied.

The two aspects (A and B) in the left part of Figure 4 share the same pointcut, and application of one aspect will prevent application of the other aspect at the same position.
In many cases it is desirable to allow pointcut matches even with intermediate events, while in other cases such intermediate events should not be allowed.

Klein et al. [8] suggest that the matching strategy of allowing intermediate events or not should be a user-configurable property of the matching tool, and will thus not be visible in the aspect diagrams. We propose on the other hand that the matching strategy is explicitly defined as part of the pointcut with the arbitrary events symbol (`/\`). This provides the benefit that we may easily define how to merge the intermediate events with the advice inserted events. We may also use different matching strategies within the same aspect. In between two events we may forbid other events, while for other event pairs we may allow intermediate events.

The arbitrary events symbol is placed on the lifeline of a pointcut or advice diagram to indicate the presence of an arbitrary number of events (including zero events). An arbitrary events symbol used in the pointcut has to be preserved from the pointcut, stay on the same lifeline and remain in the same order relative to the other arbitrary events symbols in the advice diagram. Due to this restriction we do not need identifiers for the arbitrary events symbols, but we will use the symbol `/\_{i}` to denote the `i`'th arbitrary events symbol on a lifeline `L` numbered from top to bottom. An arbitrary events symbol belongs to a specific lifeline, and it cannot be placed on a decomposed lifeline.

The pointcut in the right part of Figure 4 defines that we are looking for matches of an message followed by a message, and the arbitrary events symbol used on both the lifelines indicate that there may be arbitrary events in between `!a` and `!b` on lifeline `L1` and between `?a` and `?b` on lifeline `L2`. The corresponding advice adds an `ad` message with an explicit position relative to the arbitrary events. The transmission event of `ad`, `!ad`, shall be inserted directly after the `!a` event (and before all the arbitrary events) on lifeline `L1`, and the reception event, `?ad`, shall be inserted directly before the `?b` event (and after all the arbitrary events) on lifeline `L2`.

Figure 5A shows an illegal aspect since the arbitrary events symbols in the pointcut are not preserved in the advice. We do not allow to delete the arbitrary events since this may be harmful, and it may also produce illegal sequence diagrams where messages do not have both a transmission and reception event. We do not allow directly consecutive arbitrary events symbols in the pointcut such as in Figure 5B since this would be redundant. This disallowed redundancy case for pointcut diagrams of Figure 5B may however occur in the advice and still be allowed when explicit pointcut messages are deleted. Figure 5C illustrates that the arbitrary events symbol represents events and not messages and it is thus meaningful to specify that `!a` and `!b` may have arbitrary events in between, while `?a` and `?b` cannot have any events between them.

Since we have introduced the arbitrary events symbol and the insertRest operator, we need to extend the syntax for sequence diagrams. Both the pointcut and advice diagrams get additional EBNF clauses (the insertRest operator is only allowed for advice
Fig. 5. Rules for the arbitrary events symbol

The arbitrary events symbol has a superscripted L to indicate its owner lifeline. A possible syntax representation for the advice diagram in the right part of Figure 4 is:

\[
\text{seq}\left[\text{seq}\left[!a, ?a\right], !ad, \curvearrowleft^{L_1}, \text{seq}\left[\curvearrowright^{L_2}, ?ad, \right], !b, ?b\right]
\] (1)

Notice that there are multiple alternatives for the syntax representation, since all permutations that follow the partial orders of each lifeline are valid (and the `\text{seq}` operator may be nested arbitrarily).

4 Semantics-Based Matching

In order to make the matching semantics-based, we define matches directly on the base model traces. Remember that all messages are given unique identifiers which are shared between the transmission and reception events of the message, meaning that all the events in the pointcut trace have different identifiers from the events in the base trace.

We need an injective mapping function, \(\phi: \text{Event} \rightarrow \text{Event}\), which maps from pointcut events to base events. For each event, \(\phi\) only maps the identifier, while it preserves all the other event properties (kind, signal, transmitter, receiver). The \(\phi\) mapping function will be one-to-one between the match part of the base trace and the pointcut trace.

**Definition 2.** For a pointcut without any arbitrary events symbols we have a **match** if and only if a base trace contains a pointcut trace, where each event in the pointcut trace is mapped by \(\phi\)

In theory we may calculate all the pointcut and base traces to find matches. In practice this is an intractable problem since the number of traces may have an exponential growth relative to the number of events in the diagram. In our first test implementation we were not able to handle a relatively small base model, consisting of eleven consecutive messages in the same direction between the same two lifelines, since there are as much as 58,786 traces.

In an optimized weave algorithm we avoid calculating all the traces by instead working on the POE (last paragraph of Section 2) equivalence classes (abbreviated as **POE classes**) instead. This has a large impact on the performance since a POE class may represent thousands of actual traces, e.g. all the 58,786 traces in the base model mentioned above belong to the same POE class. The maximum number of POE classes for an interaction is equal to the total number of `\text{alt}` operands. For each POE class, we only represent the event orders per lifeline, so that each event occurs only once. An interaction may be defined as a collection of POE classes. The next lemma states that a
lifeline-based matching wrt. to each POE class is sufficient to identify all the possible matches:

**Lemma 1. (Lifeline-based matching)** For a pointcut without any arbitrary events symbols and a base trace \( \text{bTrace} \): There exists a match in one of its POE traces \( \text{(POE(\text{bTrace}}) \) if and only if

1. \( \forall l \in \text{Lifelines}: \) the event order on \( l \) of \( \text{POE(\text{bTrace}} \) contains the event order on \( l \) of the pointcut (where each event in the pointcut is mapped by \( \phi \)) AND
2. there are no messages in \( \text{POE(\text{bTrace}} \) having the reception event before the contained pointcut on one lifeline and the transmission event after the contained pointcut on another lifeline (match blocking messages).

A proof of Lemma 1 is given in [4]. Lemma 1 needs to exclude match blocking messages. Otherwise the if-direction of the lemma does not hold as we can see from Figure 6. The pointcut has a single trace: \( <!a, ?a, !b, ?b> \). None of the six shown base traces have a contained pointcut trace, and thus there are no matches (Def. 2). This is because the match blocking \( \text{c} \) message will always get its two events between the first and last events of the matched pointcut trace.

The decomposed lifelines and diagrams at different levels, as we had within the position user service (Figure 2), do not need any special treatment when we work with traces. This is because the trace events represent a flattened structure where the decomposed lifelines are not present, only lifelines with transmission and reception events. For the \( !\text{posRequest} \) event in Figure 2, the event is shown at both level 1 and level 2, but there will only be a single event represented in the trace, i.e. \( !(\text{posRequest, Response, PATS}) \).

### 4.1 Matching with the Arbitrary Events Symbol

As opposed to decompositioning of base models and aspect models, the arbitrary events symbol needs special treatment in the matching process. Since the arbitrary events symbol is a lifeline-based mechanism, it fits nicely with the lifeline-based matching. The event order per lifeline of the pointcut can simply be extended to include the arbitrary events symbol. Then the arbitrary events symbol represents a wildcard of an event list with zero or more arbitrary events, and the lifeline-based matching will work fine also for arbitrary events symbols.

Figure 7 shows a pointcut expressing that we are looking for an \( a \) message followed by a \( b \) message, where there may be an arbitrary number of events in between the \( !a \) and \( !b \) events and between the \( ?a \) and \( ?b \) events. Base model 1 will have a match, where the arbitrary events symbols have the matches: \( \phi^{L1} = <!d> \) and \( \phi^{L2} = <?c, !d> \).
The base model 2 in Figure 7 shows a base model where we have two overlapping matches, one with both a messages (reducible match) and the other one with only a single a message (irreducible match). In the reducible match, the second a message matches arbitrary events symbols. It seems more appropriate and intuitive to choose the match with only a single a message.

**Definition 3.** A match \( m_{\text{min}} = (m_1, \ldots, m_n) \) is an **irreducible match** if there is no subsequence of \( m_{\text{min}} \) which is a match.

The irreducible match definition can be easily translated also to the lifeline-based matching by saying that there shall not exist a subsequence on any of the lifelines, which also constitutes a match in combination with the other lifelines. Reducible matches will only occur in cases where the arbitrary events symbol is used. In our matching we require that the matches are irreducible.

## 5 Weaving

The previous section showed that a lifeline-based matching of the POE classes is equivalent to a semantics-based matching on the traces. This section continues by defining a lifeline-based weaving.

We calculate the POE classes of the base model, the single POE class of the pointcut and the POE classes of the advice. Since the pointcut is restricted to use only seq and events, it has always only one POE class. The POE classes are derived from an interaction by configuring all combinations of a1 t operands into (potentially) different POE classes.

The weave algorithm repeats the following three steps as long as there are unhandled matches in the base POE classes: 1) Identify a match in a base POE class (lifeline-based matching), 2) Perform lifeline-based weaving, according to Def. 4 below, for each of the advice POE classes. Add the results, a new POE class for each advice POE class, to the set of base POE classes, and 3) Remove the matched base POE class and repeat the three steps if there are more matches.

**Definition 4.** **Lifeline-based weaving** for a matched base POE class \( \text{baseP} \) with match \( m \) and an advice POE class \( \text{advP} \). The resulting POE class, res, gets the initial value: \( \text{res} = \text{baseP} \). Then the lifelines of res are updated according to three rules:

1. \( \forall l \in \text{baseP}.LLs : m(l) \neq \emptyset \Rightarrow \text{res.replaceEvts}(l, m(l), \text{advP.evts}(l)) \)
2. \( \forall l \in \text{advP}.LLs \setminus \text{baseP}.LLs : \text{res.addLL}(l, \text{advP.evts}(l)) \)
3. \( \forall l \in \text{baseP}.LLs : (m(l) = \emptyset \land \text{advP.evts}(l) \neq \emptyset) \Rightarrow \text{res.ins}(l, \text{advP.evts}(l)) \)
Prerequisites of Def. 4: A POE class contains the following methods; \( \text{LLs}() \) retrieves the set of (non-empty) lifelines; \( \text{replaceE} \text{mts}(l, \text{m}(l), \text{advP.e} \text{vts}(l)) \) replaces the match events by the advice events on lifeline \( l \); \( \text{advP.e} \text{vts}(l) \) retrieves the list of events of the advice on lifeline \( l \); \( \text{addL} \text{L}(l, \text{advP.e} \text{vts}(l)) \) adds \( l \) as a new lifeline with the advice events on \( l \) as the content; \( \text{ins}(l, \text{advP.e} \text{vts}(l)) \) inserts the advice event list on lifeline \( l \) into an appropriate position on lifeline \( l \) (the details are given below). \( \text{m}(l) \) retrieves the event list of the match on the 1 lifeline, and \( \emptyset \) denotes an empty event list.

Explanation of Def. 4: Each lifeline can be woven separately, as defined by three mutually exclusive rules. When a lifeline has matched events, rule (1), then the matched events on this lifeline are simply replaced by the corresponding advice events (in some cases an empty list). When a lifeline has events in the advice and not in the base, rule (2), then all of this advice lifeline is inserted as a new base lifeline.

The most difficult rule, rule (3), is when a lifeline has no matched events, but have events in both the base and advice, e.g. the ?adv event in the advice of Figure 8 occurs on lifeline L3 with no events in the pointcut (the match part), and there is a !b event on the L3 lifeline in the base model. Should the new event ?adv be placed before or after the !b event?. Choosing to place ?adv before !b will produce the undesired woven diagram (Figure 8) which has no possible traces because there is a deadlock.

In many cases, a proper placement can be found by exploring the partial order relationships. Let \( \text{po}(e_1, e_2) \) denote a partial order where the event \( e_1 \) must happen before the event \( e_2 \). We will produce the union of the partial orders of the advice POE class and the matched base POE class:

\[
\{ \text{po}(!a, ?a), \text{po}(!\text{adv, } ?\text{adv}), \text{po}(!b, ?b), \text{po}(?a, !\text{adv}), \text{po}(?b, ?a) \}
\]

Since partial order is a transitive relation, we may calculate the transitive closure, which will produce the pair \( \text{po}(!b, ?\text{adv}) \). This defines a unique and proper position for ?adv on the base L3 lifeline in Figure 8. There are however cases, where there may be several position choices fulfilling the partial order requirements, e.g. add another event !c (L3, L4) after !b on L3. In such cases we choose an arbitrary position among the choices except that we will avoid or minimize the number of crossing messages, and provide a warning message to the modeler.

In our advice at level 2 in Figure 3 there are four events on the DB and Response lifelines. The pointcut has no events on these lifelines, but the base model does. However, all these four events get unique positions when calculating the transitive closure of the partial order relation (due to limited space the base model details to show this is not included in the paper).

![Fig. 8. Placement of a new event on a lifeline with no events in the pointcut](image-url)
When there are no more unhandled matches, the woven result is a set of POE classes. Finally, we need to go from POE classes to a woven interaction. Each POE class is represented by a single seq operator with the lifeline events as operands in one of the legal orders (the choice is insignificant). Then all these seq operators are used as operands inside an outermost alt operator to represent the woven interaction.

Figure 9 shows the weaving of an aspect and a base model. The aspect defines a replacement of all occurrences of two consecutive messages \(m_1\) and \(m_2\), by an advice which adds the message \(m_3\) and an alt operator. Each POE class is represented by its event order on each of the two lifelines. In addition to the single pointcut POE class, there are two advice POE classes and four base POE classes. The only base POE class with a match (marked by rectangles) is woven for each of the two advice POE classes, resulting in two new POE classes which adds up to a total of five POE classes. The weaving terminates since there are no more matches. The final woven interaction is shown in the bottom right part of the figure.

5.1 Weaving of the Arbitrary Events Symbol and the insertrest Operator

To handle the arbitrary events symbols in the weaving, we need to bind these symbols to actual events relative to the match. \(L_i\) will then hold an event sequence for the \(i\)’th arbitrary events symbol on the \(L\) lifeline, and these event sequences will replace the corresponding arbitrary events symbols in the advice so that the advice will have ordinary event sequences for all its lifelines.

Each insertRest operator in the advice will be replaced by an event order per lifeline. For each lifeline the event order will be the remaining subsequence of the base model events after the match part. Alternatively we may describe this relative to the general match definition and for a given match \(\langle m_1, \ldots, m_n \rangle\) within a base trace, the insertRest events will be:
These insertRest events are distributed to event orders per lifeline for the lifeline-based implementation.

We have a tool implementation of the basic matching and weaving approach described in this paper. The tool uses an Eclipse-based SeDi sequence diagram editor v.1 [11] to define base, pointcut and advice diagrams. The weaving has been verified to behave correctly on our test examples, by manually investigating the woven textual interactions. We are currently implementing a translation from textual interactions to graphical diagrams for easier manual validation purposes. Future work is to extend the tool to also support the arbitrary events symbol, decomposition (introduced in SeDi v.2) and the insertRest operator.

5.2 Discussion

Consider the aspect and base model example in Figure 10. The aspect defines that two consecutive a messages should be replaced by a b message, and the base model contains four consecutive a messages. Without using ids and the injective mapping function in the match, we could mistakenly choose a match which does not pair the correct transmission and reception events (shown as black boxes in the figure). By matching the last two events on the L1 lifeline and the first two events on the L2 lifeline, we get a final woven result with a crossing b message (notice that the b message is a match blocking message for the two remaining a messages). Crossing messages are allowed in general, but it is unexpected and undesired in this case.

We have described the matching strategy as a random matching. Find any match, perform weaving and repeat the process. If our weaving terminates, then we are guaranteed that there will not exist any matches in the woven model. With the aspect and base models in Figure 10, a random matching strategy gives one of the following three alternative derivations with two different end results: 1) a, a, a ⇒ a, b, a, 2) a, a, a, a ⇒ a, a, b ⇒ b, b, and 3) a, a, a ⇒ b, a, a ⇒ b, b. Klein et al. [9] suggest a left-most matching strategy leading to the unique derivation alternative 3. Our weaving also supports the left-most matching by ensuring that we always choose the top-most matches of each lifeline.

We define a plain additive aspect to be an aspect that does not delete events. For such aspects we will mark all the events in a treated match and exclude them from

\[
\text{baseTrace} = \langle \ldots, m_1, \ldots, m_n, \quad \ldots \rangle \\
\text{insertRest}
\]

![Fig. 10. Incorrect match leads to undesired weaving / Alternative woven results](image-url)
possible future matches. This ensures a terminating weaving process for a lot of aspects
that would otherwise never terminate, e.g. a,a → a,a,b (shorthand notation for an
aspect: pointcut → advice).

6 Related Work

In this paper we have restricted the base model to use only the seq and alt operators.
However, the results are directly applicable to other operators that can be defined with
seq and alt, e.g. opt (optional), par (parallel), and the loop operator for loops with
an upper bound. The strict operator is not supported. It represents a strict sequence
of events also across lifelines, which is in strong contrast to our approach.

We have reported complementary work in [4], where we perform a static weaving
on a finite structure even for many typical loops without upper bounds. This paper goes
beyond [4] by providing details of the aspect language including property matching,
identifiers, negative pointcuts, decomposition, the insertRest operator and the arbit-
rary events symbol.

The pointcut model in AspectJ [7] cannot express matching based on a sequence
of events, which is necessary to encounter the problem of syntactic-based matching
described in this paper. QVT [13] is a model-to-model transformation language which
supports general source and target MOF-languages. Since we address transformations
where the source and target is the same language, UML sequence diagrams, we benefit
from making tailored constructs and enabling the user to work on the more intuitive
concrete syntax.

The identifiers, property name matching, and the arbitrary events symbol are in-
spired by Join Point Designation Diagrams (JPDD) [18] proposed by Stein et al. We
have modified the notation slightly and introduced advice diagrams since JPDD only
covers pointcut diagrams. JPDD is intended for mapping to aspect-oriented program-
ing languages such as AspectJ, as opposed to our model matching and weaving.

Deubler et al. [2], Solberg et al. [17], and Jayaraman et al. [6] all define syntactic-
based approaches for sequence diagrams. Deubler et al. match single events only and
provide no model weaving or mapping to a concrete aspect language. Solberg et al.
rely on binding models instead of a generic matching pattern (as in our approach), to
identify the base model elements to be affected by the aspects.

Klein et al. [8,9] perform a semantics-based weaving of sequence diagrams by using
automata representations. They present four different matching choices which is defined
to be a tool-specific configuration, and where a single matching strategy applies to the
entire pointcut diagram. We support the general part (by using the arbitrary events
symbol) and enclosed part.

Klein et al. [8] support wildcard matching of events. The important difference be-
tween their approach and our proposed arbitrary events symbol, is that they have no
explicit graphical element for this, but define a matching strategy outside of the aspect
diagram. Their approach have two drawbacks compared to our arbitrary events symbol:
1) their wildcard matching applies in all or no positions within the entire diagram, 2) it
is a tool choice (or a global choice) how to merge the wildcard events with the aspect
added events.
Decomposed lifelines and aspect diagram definition over multiple levels is something we have not seen in any related work. Decomposition for aspect diagrams allows the modeler to work with smaller units in isolation. Furthermore, decomposition does not introduce any added complexity since this syntactic arrangement is not visible at the trace level where our matching and weaving is performed.

Avgustinov et al. [1] have a trace-based run-time matching of events to execute some extra code when a match occurs. Since this happens during run-time and not statically as in our approach, the aspects are restricted to additive parts that are inserted entirely after the already executed match part. While performance is a major issue in run-time weaving, our weaving is static and termination within reasonable time is sufficient.

7 Conclusions

We have proposed an aspect language for UML 2.0 sequence diagrams. Aspect diagrams in terms of pointcut and advice diagrams use the same graphical elements as sequence diagrams, with only a few extensions, thus providing a familiar notation to sequence diagram users. The advice diagram replaces matches of the pointcut diagram, which can simulate traditional aspect-oriented mechanisms like before, after, replace and around.

Syntactic-based matching will fail to match all the intended base model joinpoints even for simple pointcuts as two consecutive messages. Our aspect language is therefore based upon a formal trace model (STAIRS) for sequence diagrams, and thereby on semantics-based matching. Matching is defined on traces of the base model traces. For performance reasons we have established a semantically equivalent implementation to plain trace-based matching, which works on partial order equivalent classes representing sets of traces.

The arbitrary events symbol is a powerful extension that allows to match an arbitrary number of events on a lifeline. This symbol allows to define flexible and robust pointcut definitions, and also to define how the additive parts of an aspect shall be positioned in relation to these arbitrary events. This mechanism may be useful in the context of multiple aspects or base model evolution which could otherwise unintentionally prevent matches of a pointcut.

As future work we plan to investigate more real-life scenarios to see if the expressiveness of the proposed language is enough, or if additional constructs are needed. We are also investigating confluence and termination properties of the proposed aspect language.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project).

References

14. OMG. UML 2.0 Superstructure Spec., OMG Adopted Spec. ptc/03-08-02 (August 2003)
Appendix D

Paper 3: Semantics-Based Weaving of UML Sequence Diagrams
Abstract. In this paper we briefly introduce an aspect language that can define cross-cutting effects on a set of UML 2.0 sequence diagrams. Our main contribution is to weave aspects and sequence diagrams at the model level. By basing the weaving upon a formal trace model for sequence diagrams, we ensure that the weaving is semantics-based. To avoid the intractability of working on complete trace sets, we define a lifeline-based weaving upon trace-based equivalence classes. A major challenge is to handle unbounded loops which produce infinite trace sets. We establish a systematic way to rewrite the original loop definition so that the weaving can be performed on a finite structure. A weaving tool has been implemented to validate the approach.

1 Introduction

Aspect-orientation for programming has emerged as a promising way to separately define cross-cutting parts of programs, in order to achieve separation of concerns. We believe that the same potential is there also for modeling. This paper explores aspect-oriented modeling for UML 2 sequence diagrams [13].

In aspect-oriented programming the base program is the main program upon which one or more aspects may define some cross-cutting code as additions or changes. An aspect is defined by a pair (pointcut and advice), where the pointcut defines where to affect the base program and the corresponding advice defines what to do in the places identified by the pointcut. Analogously we term our set of sequence diagrams as the base model, and we define an aspect diagram to consist of a pointcut diagram and an advice diagram, both based upon the concrete syntax of sequence diagrams.

In this paper we assume that the sequence diagrams are used as input for tools that automatically produce executable test code, e.g. to test if a sequence diagram is a correct refinement of another sequence diagram [12], or to test if a system specified by UML statecharts, class diagrams and object diagrams is consistent with sequence diagram specifications [14]. The test tools expect complete sequence diagrams. Therefore we need to weave the aspect and the base model, into a woven model, before generation of test code.

The woven model is not intended to be viewed (except for debugging) or further updated by the modeler. This means that the structure of the result is not a primary focus. It suffices that the woven model is semantically correct with respect to our formal model.

Many aspect-oriented approaches suffer because they rely on a pure syntactic pointcut matching and weaving. This paper gives part of an answer to the following question:
How can we use the trace-based formal model of STAIRS [15] to achieve a semantics-based matching and weaving algorithm for sequence diagrams?

The paper is organized as follows; Section 2 introduces the STAIRS formal model for sequence diagrams; Section 3 presents the matching and weaving approach; Section 4 shows how we may statically weave unbounded loops; Section 5 describes our tool implementation; Section 6 presents related work; and finally Section 7 provides the conclusions.

2 STAIRS

STAIRS gives the semantics of a sequence diagram by a set of traces that represents the set of possible execution runs. The trace set may be infinite while each individual trace is finite.

The syntax of a UML sequence diagram, called interaction, follows the EBNF of Figure 1 [15]. We focus on the operators seq, alt and loop. The first two operators are chosen because they are the basic operators from which we also may define several other operators. The loop is included since it provides some challenges in the context of semantics-based weaving (section 4).

Each message is represented by two events, a transmission event (!) and a reception event (?) (the transmitter and receiver lifelines are omitted for readability in the paper examples when this information is unambiguously defined by associated diagrams). An event takes place on a lifeline L1 if it is a transmission event on L1, e.g. !(signal, L1, L2), or a reception event on L1, e.g. ?(signal, L2, L1). We assume that the messages are complete (i.e. contain both events ! and ?) within each alt operand, loop operand, pointcut, advice and each sequence diagram in the base model.

The weak sequence operator, seq, of sequence diagrams imposes a partial order of events given by: 1) the transmission event must come before the reception event of the same message, and 2) all events are ordered for each lifeline in isolation. An intuitive idea behind this partial order is that messages are sent asynchronously and that they may happen in any order on different lifelines, but sequentially on the same lifeline. Figure 2 shows an interaction example and the corresponding arrows of its four partial order requirements.

Sequence diagrams allow crossing messages, i.e. two messages a and b are crossing only when they have events on the same two lifelines, and a has an event before b on one lifeline, while b has an event before a on the other lifeline, e.g. seq [(a, L1, L2), !(b, L1, L2), ?(b, L1, L2), ?(a, L1, L2)].

The alt operator defines alternative interactions for each operand, and the loop operator represents alternative interactions for each allowed repetition of the loop body.

Fig. 1. Syntax of interactions
Notice that the loop operator has an optional parameter, Set, to define the possible iterations of the loop. This can be expressed in many different ways such as \{1, 3, 5\} or 2..\,* (0..\,* is the default).

We briefly explain the semantics operator, \(\llbracket \rrbracket\), while a precise definition is given in [15]. The semantics of an interaction \(i\) is \(\llbracket i \rrbracket = (p, n)\), where \(p\) is the set of positive traces and \(n\) is the set of negative traces. Positive traces define valid behavior and negative traces define invalid behavior, while all other traces are defined as inconclusive. In this paper we concentrate on aspects that only affect positive traces, and we therefore use a simplified model without negative traces: that is \(\llbracket i \rrbracket = p\). A trace is a sequence of events which we display as \(\langle e_1, \ldots, e_n \rangle\), where \(e_i\) are events for all \(i \in 1..n\).

The \(\llbracket \rrbracket\) operator produces one trace for each valid permutation of events that satisfy the two partial order requirements as explained for the seq operator above. The \(\llbracket \rrbracket\) operator produces the union of the traces for each operand of the alt operator and the union of traces for each possible number of iterations of a loop. \(n\) iterations of the loop are replaced by a weak sequence of \(n\) occurrences of the loop body before the semantics operator is applied to it. If the loop has no upper bound, called unbounded loop, then we will have infinitely many traces. Each message in the trace is dynamically given a unique identifier, which is shared between the transmission and reception events of the message. We say that two interactions \(i_1\) and \(i_2\) are semantically equivalent if they represent the same trace set (except for the dynamic identifiers), i.e. if \(\llbracket i_1 \rrbracket = \llbracket i_2 \rrbracket\).

We define one trace to be partial order equivalent (POE) to another trace if they are both permutations of the same set of events with the same order on each lifeline. The \(\llbracket \rrbracket\) operator is used to define POE since it is defined to produce all such permutations:

**Definition 1.** We say that two traces \(t^A = \langle t_1^A, \ldots, t_n^A \rangle\) and \(t^B = \langle t_1^B, \ldots, t_n^B \rangle\) are partial order equivalent (POE) if and only if:

\[
\llbracket \text{seq} [t_1^A, \ldots, t_n^A] \rrbracket = \llbracket \text{seq} [t_1^B, \ldots, t_n^B] \rrbracket
\]

We let the function, POE: Trace \(\rightarrow\) TraceSet, calculate all the POE traces of a given trace, \(\text{POE}(\langle t_1^A, \ldots, t_n^A \rangle) = \llbracket \text{seq} [t_1^A, \ldots, t_n^A] \rrbracket\). The diagram in Figure 2 has two traces which are POE: \(\langle lm1, \, lm1, \, lm2, \, ?m2 \rangle\) and \(\langle ?m1, \, lm2, \, ?m2, \, ?m2 \rangle\). The function POE() of either of these two traces returns the set of both traces.

### 3 The Approach

This section explains how we perform the semantics-based weaving for finite traces where there are no unbounded loops, while section 4 covers unbounded loops. Finite
loops can be seen as a set of alternatives, so we only have to cover the seq and alt operators.

The aspect diagrams are inspired by graph transformation [4] where the left part, the pointcut diagram, defines a pattern for which we are looking for matches or morphisms in the base model. The right part, the advice diagram, defines a replacement of the matches within the base model. This implies that messages present only in the pointcut and not in the advice, will be deleted, while messages present only in the advice and not in the pointcut, will be added. Both the pointcut and advice diagrams are based upon the graphical elements of sequence diagrams so that the modeler can work with an already familiar notation.

3.1 Syntactic-Based Matching Does Not Work

If the pointcut identifies only a single message to be matched, then there is no difference between syntactic-based and semantics-based matching. However, even for only two consecutive messages, as in the example in Figure 3, syntactic-based and semantics-based matching is different.

The pointcut of Figure 3 expresses that the message m1 from the lifeline L1 to the lifeline L2 is followed by the message m2 from L1 to L2. A weaving must replace all pointcut matches by an advice which adds the message new and an alt operator. The base model has two consecutive alt operators. An alt operator defines a choice of different alternatives, where the alternatives are given as operands separated by a dashed line. If we try to find matches of the pointcut within the base model with pure syntactic matching, then we do not find any matches. However, one possible execution trace chooses the second operands of the two alt operators, which then should result in a match of the specified pointcut.

![Diagram](image)

Fig. 3. Example: Aspect model, base model, and expected woven model(extract)

3.2 Lifeline-Based Matching

In order to make the matching semantics-based, we define matches directly on the base model traces. We need an injective mapping function, \( \phi: \text{Event} \rightarrow \text{Event} \), which maps from pointcut events to base events. For each event, \( \phi \) only maps the identifier, while it preserves all the other event properties (kind, signal, transmitter, receiver).

**Definition 2.** We have a match if and only if a base trace contains a pointcut trace (where each event in the pointcut trace is mapped by \( \phi \)).
In theory we may calculate all the pointcut and base traces to find matches. In practice this is an intractable problem since the number of traces may have an exponential growth relative to the number of events in the diagram. In our first test implementation we were not able to handle a relatively small base model, consisting of eleven consecutive messages in the same direction between the same two lifelines, since there are as much as 58,786 traces.

In an optimized weave algorithm we avoid calculating all the traces by instead working on the POE equivalence classes (abbreviated as POE classes) instead. This has a large impact on the performance since a POE class may represent thousands of actual traces, e.g. all the 58,786 traces in the base model mentioned above belong to the same POE class. The set of POE classes, representing an interaction, is derived from an interaction by a tree-like traversal of the alt operands. Each valid combination of the alt operands represents a POE class. For each POE class, we only represent the event orders per lifeline, so that each event occurs only once. Hence, the optimized algorithm scales well and performs linearly wrt. to the number of alt operands and the number of events. The next lemma states that a lifeline-based matching wrt. to each POE class is sufficient to identify all the possible matches:

**Lemma 1. (Lifeline-based matching)** For a base trace, $bTrace$, there exists a match in one of its POE traces ($POE(bTrace)$) if and only if

1. $\forall l \in \text{Lifelines}: \text{the event order on } l \text{ of } POE(bTrace) \text{ contains the event order on } l \text{ of the pointcut (where each event in the pointcut is mapped by } \phi) \text{ AND}$
2. there are no messages in $POE(bTrace)$ having the reception event before the contained pointcut on one lifeline and the transmission event after the contained pointcut on another lifeline (match blocking messages).

**Proof:** *If-direction:* Assume all the pointcut event orders per lifeline is contained within base event orders per lifeline, and that there are no match blocking messages in the base trace. No match blocking messages ensure that we may construct a matching base trace within $POE(bTrace)$ as follows: select all the events on each lifeline prior to each lifeline match in one of the valid orders, then select all the events from the $pTrace$ and persist their order (this will be the contained match), then proceed with any valid selection of the remaining events. *Only-if-direction:* Assume there is a pointcut trace contained in a base trace. No match blocking messages follow directly since such messages otherwise would split the match events. To get a contradiction, assume there is a lifeline on which the pointcut event order is not contained within the base event order of the same lifeline. Then there must be an intermediate event, not part of the match, between two of the matching base trace events on the lifeline. But then it would also be part of the match trace due the relationship between lifeline event orders and traces. □

Lemma 1 needs to exclude match blocking messages. Otherwise the if-direction of the lemma does not hold as we can see from Figure 4. The pointcut has a single trace: $(!a, ?a, !b, ?b)$. None of the six shown base traces have a contained pointcut trace, and thus there are no matches (Def. 2). This is because the match blocking c message will always get its two events between the first and last events of the matched pointcut trace.
3.3 Lifeline-Based Weaving

The previous section showed that a lifeline-based matching of the POE classes is equivalent to a semantics-based matching on the traces. This section continues by defining a lifeline-based weaving. We calculate the POE classes of the base model, the single POE class of the pointcut and the POE classes of the advice. Since the pointcut is restricted to use only seq and events, it has always only one POE class.

The weave algorithm repeats the following three steps as long as there are unhandled matches in the base POE classes: 1) Identify a match in a base POE class (lifeline-based matching), 2) Perform lifeline-based weaving, according to Def. 3 below, for each of the advice POE classes. Add the results, a new POE class for each advice POE class, to the set of base POE classes, and 3) Remove the matched base POE class and repeat the three steps if there are more matches.

Definition 3. Lifeline-based weaving for a matched base POE class (baseP) with match m and an advice POE class (advP). The resulting POE class, res, gets the initial value: res = baseP. Then the lifelines of res are updated according to three rules:

1. \( \forall l \in \text{baseP.LLs} : m(l) \neq \langle \rangle \Rightarrow \text{res.replaceEvts}(l, m(l), \text{advP.evts}(l)) \)
2. \( \forall l \in \text{advP.LLs} \setminus \text{baseP.LLs} : \text{res.addLL}(l, \text{advP.evts}(l)) \)
3. \( \forall l \in \text{baseP.LLs} : (m(l) = \langle \rangle \land \text{advP.evts}(l) \neq \langle \rangle) \Rightarrow \text{res.ins}(l, \text{advP.evts}(l)) \)

Prerequisites of Def. 3: A POE class contains the following methods; LLs() retrieves the set of (non-empty) lifelines; replaceEvts(l, m(l), advP.evts(l)) replaces the match events by the advice events on lifeline l; advP.evts(l) retrieves the list of events of the advice on lifeline l; addLL(1, advP.evts(1)) adds 1 as a new lifeline with the advice events on 1 as the content; ins(1, advP.evts(1)) inserts the advice event list on lifeline 1 into an appropriate position on lifeline 1 (the details are given below). m(1) retrieves the event list of the match on the 1 lifeline, and \( \langle \rangle \) denotes an empty event list.

Explanation of Def. 3: Each lifeline can be woven separately as defined by the three mutually exclusive rules. When a lifeline has matched events, rule (1), then the matched events on this lifeline are simply replaced by the corresponding advice events (in some cases an empty list). When a lifeline has events in the advice and not in the base, rule (2), then all of this advice lifeline is inserted as a new base lifeline. Figure 5 shows how the lifeline-based weaving works for rules 1 and 2.

The most difficult rule, rule (3), is when a lifeline has no matched events, but have events in both the base and advice, e.g. the ?adv event in the advice of Figure 6 occurs on lifeline L3 with no events in the pointcut (the match part), and there is a !b event on
the L3 lifeline in the base model. *Should the new event, ?adv, be placed before or after the !b event?*. Choosing to place ?adv before !b will produce the undesired woven diagram (Figure 6) which has no possible traces because there is a deadlock.

In many cases, a proper placement can be found by exploring the partial order relationships. Let \( \text{po}(e_1, e_2) \) denote a partial order where the event \( e_1 \) must happen before the event \( e_2 \). We will produce the union of the partial orders of the advice POE class and the matched base POE class:

\[
\{ \text{po}(!a, ?a), \text{po}(!adv, ?adv), \text{po}(!b, ?b), \text{po}(?a, !adv), \text{po}(?b, ?a) \}
\]

Since partial order is a transitive relation, we may calculate the transitive closure, which will produce the pair \( \text{po}(!b, ?adv) \). This defines a unique and proper position for ?adv on the base L3 lifeline in Figure 6. There are however cases, where there may be several position choices fulfilling the partial order requirements, e.g. add another event ! (c, L3, L4) after !b on L3. In such cases we choose an arbitrary position among the choices except that we will avoid or minimize the number of crossing messages, and provide a warning message to the modeler.

We know that the transitive closure of the partial orders will not produce conflicting position instructions. Otherwise we get a contradiction: Assume there exists two events, \( e_1 \) and \( e_2 \), on the lifeline where the new advice event, new, shall be inserted, such that \( \text{po}(e_1, e_2) \). The only way to get a conflict is if both \( \text{po} (\text{new}, e_1) \) and \( \text{po} (e_2, \text{new}) \) are part of the transitive closure. But then also the pairs \( \text{po}(e_1, \text{new}) \) and \( \text{po}(\text{new}, e_2) \) must belong to the transitive closure. This is a contradiction, and we may conclude that we will not encounter conflicting partial order requirements for new advice events.

When there are no more unhandled matches, the woven result is a set of POE classes. Finally, we need to go from POE classes to a woven interaction. Each POE class is represented by a single seq operator with the lifeline events as operands in one of the legal orders (the choice is insignificant). Then all these seq operators are used as operands inside an outermost alt operator to represent the woven interaction.
Figure 7 shows our proposed weaving on the example in Figure 3. Each POE class is represented by its event order on each of the two lifelines L1 and L2. The only base POE class with a match is woven for each of the two advice POE classes, resulting in five woven POE classes. The final woven interaction (Figure 7) is semantically equivalent to the expected woven result (Figure 3).

3.4 Discussion

We have described the matching strategy as a random matching. Find any match, perform weaving and repeat the process. If our weaving terminates, then we are guaranteed that there will not exist any matches in the woven model. Consider an example of an aspect \( a, a \rightarrow b \) (shorthand notation for an aspect: pointcut \( \rightarrow \) advice), and a base model \( a, a, a, a \), where the a’s and b’s are messages in the same direction between the same two lifelines. A random matching strategy gives one of the following three alternative derivations with two different end results: 1) \( a, a, a, a \Rightarrow a, b, a, a \Rightarrow a, a, b \Rightarrow b, b \), and 3) \( a, a, a, a \Rightarrow b, a, a \Rightarrow b, b \). Klein et al. [10] suggest a left-most matching strategy leading to the unique derivation alternative 3. Our weaving supports the left-most matching by choosing the top-most matches of each lifeline.

We define a plain additive aspect to be an aspect that does not delete events. For such aspects we will mark all the events in a treated match and exclude them from possible future matches. This ensures a terminating weaving process for a lot of aspects that would otherwise never terminate, e.g. \( a, a \rightarrow a, a, b \).

Our weaving algorithm uses the lifeline-based matching for performance reasons. The remainder of this paper will, however, refer to matches according to the equivalent trace-based match definition (Def. 2) to ease the presentation.

4 Weaving Unbounded Loops

This section describes how, and under which conditions, we can do the weaving also for unbounded loops. Loops without an upper bound are troublesome because they produce an infinite trace set.

We classify unbounded loops as two types relative to a pointcut. The two loop types need different kinds of treatment. Figure 8 shows a pointcut to the left and two unbounded loops to the right:
– *non-matchRepetitive*. A loop which cannot produce any matches on its own. Such a loop may however be part of a match in combination with trace events outside of the loop. The loop in Figure 8A is non-matchRepetitive since the c message prevents the loop to produce matches only from loop events no matter how many iterations we use.

– *matchRepetitive*. A loop which produces matches on its own after some number of iterations. Such a loop has infinitely many matches since the number of iterations is infinite. The loop in Figure 8B is matchRepetitive since we get the first match by three iterations. Thus, there will be \( n \) matches after \( n \times 3 \) iterations for all \( n > 0 \).

We will treat the loops by rewriting them into an expanded, but semantically equivalent (except possibly some weaving) interaction structure. After the rewrite we have isolated or woven a remaining unbounded loop such that we are guaranteed it will not take part in further matches. The treatment of loops happens before the rest of the weaving process, and each loop is treated individually and in isolation from the rest of the model. This means that we cannot ensure a left-most matching for unbounded loops, but restrict ourselves to cases where it is acceptable with a random matching strategy. When all the unbounded loops are treated, they will be ignored while we perform ordinary weaving for the surrounding finite parts of the interaction.

We now introduce three conditions under which we are able to present (terminating) algorithms to statically weave unbounded loops: 1) The unbounded loop bodies contain no \texttt{alt} or \texttt{loop} operators, 2) the aspect is plain additive, and 3) the pointcut is connected.

A diagram is connected if and only if every involved lifeline has a path to any other involved lifeline in the diagram. There is a path between two lifelines if there is a message from one of them to the other. Furthermore this path relation is both reflexive, symmetric and transitive. In Figure 9A both the pointcut and base models are connected, while both the pointcut and base models are disconnected in Figure 9B since neither of L1 and L2 has no path to neither of L3 and L4.
We need a way to differentiate the two loop types. The following lemma proves it is sufficient to consider the loop with an upper bound equal to the number of messages in the pointcut. If that bounded loop has a match, then it is matchRepetitive, otherwise it is non-matchRepetitive (numP returns the number of messages within the pointcut diagram):

**Lemma 2.** An unbounded loop \( lp = \text{loop} [\text{body}] \) is **matchRepetitive** for a pointcut pd if and only if there exist at least one match in the bounded loop \( lp^b = \text{loop} \{\text{numP}\} [\text{body}] \).

**Proof:** *If-direction:* Since \( lp \) has a match after \( b \) iterations, it is by definition matchRepetitive. *Only-if-direction:* Let \( n \) be the fewest number of iterations for \( lp \) which gives a match. Such a matching trace must involve at least one event from all iterations, otherwise we could exclude iterations not contributing to the match and get a match within fewer iterations than \( n \). The pointcut trace which equals the match involves a number of messages which all have two events. Since the match need to involve both the transmission and reception events of a message, we know that each iteration contributes with at least two events in the matching trace. The length of the pointcut trace is twice the number of messages, which means that \( n \) cannot be larger than the number of messages within the pointcut. \( \square \)

If we apply this lemma to the example pointcut that contains four messages in Figure 8, then it is sufficient to consider the bounded loop of four iterations for any loop to determine if it is matchRepetitive or not. Now that we have a systematic way to determine the loop type relative to a pointcut, the next two sections show how to perform a static weaving for non-matchRepetitive and matchRepetitive loops.

### 4.1 Non-matchRepetitive Loops

From the definition of a non-matchRepetitive loop, we know that possible matches include preceding or succeeding (of the loop) trace events or both. Matches starting in the preceding trace events may be ended by loop trace events, and matches starting in the loop trace events may be ended by succeeding trace events. By looking at the proof of lemma 2 we deduce that the maximum number of loop iterations involved in the match, is \( \text{numP} - 1 \).

We translate the syntactic representation of the loop into a semantically equivalent form (Def. 4):

**Definition 4.** An unbounded, non-matchRepetitive loop, \( \text{loop} [\text{body}] \), has the following rewrite expression:

\[
\text{alt}\{ \text{loop} [0..((\text{numP} - 1) \ast 2 - 1)] [\text{body}] , \text{seq}\{ \text{loop}[\text{numP} - 1] [\text{body}] , \text{loop'}[\text{body}] , \text{loop}[\text{numP} - 1] [\text{body}] \} \}
\]

The remaining unbounded loop, \( \text{loop'} \), can no longer be part of a match, if all the surrounding finite parts are woven. \( \text{loop'} \) is always preceded by \( \text{numP} - 1 \) loop iterations, which prevents any matches to end in the unbounded loop. Similarly the unbounded loop is always followed by \( \text{numP} - 1 \) loop iterations, which prevents that matches start...
in the unbounded loop. This claim that the remaining unbounded loop cannot contain matches rely on the two conditions of connected pointcut (connected blocking match) and that the aspect is plain additive (the blocking match is maintained by the weaving).

Figure 9A illustrates why the rewrite expression works. We have a pointcut with two consecutive messages \(x, y\), an original base model (not shown) \(x, x, \text{loop}[y]\), and a simplified extract of the base model showing the second alt operand result of the rewrite. After the rewrite, the remaining unbounded loop is prefixed by \(\text{loop}\{1\}[y] = y\). It is easy to see that preceding \(x\) messages are matched in the preceding bounded loop, and that this match effectively blocks possible matches ending in the remaining unbounded loop.

With a slight modification to the example in Figure 9A, we get the disconnected pointcut (and disconnected base model) in Figure 9B. Now we may (depending on the advice) get a match starting with one of the \(x\) messages and ending with the \(y\) message in the unbounded loop which illustrates that the rewrite expression does not always ensure a proper result for disconnected pointcuts.

### 4.2 MatchRepetitive Loops

The base model example and associated aspect in Figure 10 is adopted from Klein et al. [9]. The base model starts with a login attempt from Customer to Server. At the end the Server finally answers with an ok message to indicate successful login. In between these two events there may be zero or more iterations of a loop. The loops first message, \(\text{tryAgain}\), informs of login failure, while the second message, \(\text{newAttempt}\), is a new customer login attempt.

The aspect in Figure 10 expresses that whenever the message \(\text{newAttempt}\) is followed by \(\text{tryAgain}\), then add another message \(\text{saveAttempt}\), in between the two messages matched by the pointcut, to log the failed attempt. Since we only want to log bad attempts, we need to ensure that the the message \(\text{newAttempt}\) is followed by \(\text{tryAgain}\). A syntactic-based pointcut matching fails to find matches within our base model, since the two messages come in a different order syntactically. However we easily observe that they will occur in an execution involving two or more iterations.

We now use lemma 2 to check if our base model loop example is matchRepetitive \((tA=\text{tryAgain} \text{ and } nA=\text{NewAttempt})\):

\[
\begin{align*}
numP &= 2 \\
 admitting{p} = \{!nA, ?nA, !tA, ?tA\} \\
\end{align*}
\]

Fig. 10. Base model: login w/ loop, Aspect: logging
This bounded loop has a single trace with a match, meaning that our base model loop is matchRepetitive. This means that there are matches for every even iteration. In a first weaving attempt we make a new loop with a woven body for all even iterations. Such a loop is produced by expanding the loop body to two iterations and weaving the loop body ($sA = \text{saveAttempt}$):


This weaving attempt is not good enough, since this loop still is matchRepetitive and has match(es) for all iterations greater than or equal to 2. This happens because the end part of the loop body together with the beginning part of the loop body makes a match, which we miss when only weaving the loop body without taking into account that it may have repetitions. Since the pointcut is connected and there is a marked match within the loop, we know that additional matches are restricted to be only within the part after the match combined with the part before the match.

To fix the problem of missed matches, we permute the loop so that a match part makes up the beginning of the loop body. Then we also need to insert a prefix and a postfix so that the semantics of the loop is not changed, resulting in the following rewrite expression (before/after is the sequence of events relative to the match in the loop body):

$$\text{seq} [\text{before, loop} [\text{seq} [\text{match, after, before}], \text{match, after}]]$$

For the example in Figure 10 we get the following woven rewritten structure, where the permutation resulted in one additional match/weaving in the loop body:


This structure is now fine for all the original even iterations. However, we have lost all the original iterations of odd numbers. By appending the $\text{loop} \{0..1\} [\text{body}]$ to the end, we also get the odd iterations, and this is a bounded loop for which we can generate all traces. In general, the appended loop should be $\text{loop} \{0..(numP - 1)\} [\text{body}]$.

We have only considered unbounded loops with cardinality 0..*, where * is unbounded. Other unbounded loops may easily be translated into finite loops combined with 0..* loops by rewrites from $\text{loop} \{n..*\} [\text{body}]$ to $\text{seq} [\text{loop} \{n\} [\text{body}], \text{loop} [\text{body}]]$, and using the alt operator to split loops of the form $\text{loop} \{2, 10, 20..*\}$.

## 5 Implementation

We have a tool implementation of the full approach described in this paper. The tool uses the Eclipse-based SeDi sequence diagram editor v.1 [11] to define base, pointcut and advice diagrams. The weaving has been verified to behave correctly on the paper examples, by manually investigating the woven textual interactions. We are currently implementing a translation from textual interactions to graphical diagrams for easier manual validation purposes.
In [5] we have described how to define a single aspect to weave cross-cutting behavior into 40 sequence diagrams of an SMS-based buddy positioning service. The aspect definition uses some additional constructs, outside the scope of this paper, such as decomposition, negative application conditions, wildcards and an `insertRest` operator.

Automata-based weaving attempts (Grosu and Smolka [6], Klein et al. [10,9]) achieve semantics-based weaving of UML 2 sequence diagrams. In contrast to our approach they cannot handle loops leading to irregular trace expressions. Example (from [10]): Figure 11 shows an aspect that matches an m1 message followed by an m2 message. The base represents an unbounded sequence diagram since it has a loop that leads to an irregular expression. n loop iterations means that there are also n a messages and n b messages, and these messages may have an arbitrary order. There is an m1 message before the base loop and an m2 message after the base loop, and independent of the number of loop iterations there will always be a match in the base.

The woven interaction of Figure 11 shows that our weaving tool produces the expected result. The outermost `alt` operator contains four alternatives representing the POE classes of 0, 1, 2, and 2+ loop iterations, and all of these POE classes have the `advice` message added exactly once in an appropriate position. We need to emphasize that the `?advice` message is defined to be inserted directly after the `!m1` message on the L2 lifeline, which means that `?advice` must come before all the `!a` messages. Although it looks odd that the `?advice` event is placed before the corresponding `!advice` in the latter POE class, this is allowed for textual interactions, and will be sorted out by the `[]` operator when making traces. Notice also that we indicate the messages that have been matched and the loops that are treated, by the prime (`'`).

6 Related Work

In this paper we have restricted the base model to use only the `seq`, `alt` and `loop` operators. However, the results are directly applicable to other operators that can be defined with `seq` and `alt`, e.g. `opt` (optional) and `par` (parallel). The `strict` operator is not supported. It represents a strict sequence of events also across lifelines, which is in strong contrast to our approach.

The pointcut model in AspectJ [8] cannot express matching based on a sequence of events, which is necessary to encounter the problem of syntactic-based matching described in this paper.

Klein et al. [9], Stein et al. [17] allow the match to contain additional events in between the explicit pointcut events, called `general part` matching in [9]. Our matching
definition in this paper corresponds to the enclosed part matching, while our full aspect language includes the arbitrary events symbol [5] to support general part matching.

Clarke and Walker [2] model aspects using sequence diagrams. Their intention is to map the aspects onto an aspect-oriented programming language, such as AspectJ, and not to produce woven sequence diagrams as in our approach.

Deubler et al. [3], Solberg et al. [16] and Jayaraman et al. [7] all define syntactic-based approaches for sequence diagrams. Deubler et al. can only match single events. The approach taken by Jayaraman et al. is similar to ours in that they define aspects similar to graph transformation rules directly upon sequence diagram syntax.

Avgustinov et al. [1] have a trace-based run-time matching of events to execute some extra code when a match occurs. Since this happens during run-time and not statically as in our approach, the aspects are restricted to additive parts that are inserted entirely after the already executed match part (excluding aspects like in Figure 10). While performance is a major issue in run-time weaving, our weaving is static and termination within reasonable time is sufficient.

7 Conclusions

We have demonstrated that it is possible to do semantics-based aspect weaving for UML 2.0 sequence diagrams based upon a formal trace model for these. In our semantics-based weaving the matching is defined at the trace level (‘what’ the sequence diagrams really describe), and not at the syntactic level (‘how’ the sequence diagrams are described), but still with the convenience for the developer that the advice specifications can be done by means of syntactic elements of sequence diagrams.

We have proven that the semantics-based matching is equivalent to a lifeline-based matching upon trace-based equivalence classes, and our lifeline-based weaving algorithm will thus produce the same result as a pure trace-based implementation. While a pure trace-based implementation performs in exponential time wrt. to the number of events, the lifeline-based implementation performs in linear time wrt. the number of events and the number of alt operands.

Klein et al. [10] have an automata-based (and semantics-based) weaving that fails to handle cases of infinite loops leading to non-regular trace expressions. Such loops impose no problem in our solution.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project). We thank Mass Soldal Lund for valuable discussions and implementation help.

References


Appendix E

Paper 4: Comparison of Three Model Transformation Languages
Comparison of Three Model Transformation Languages

Roy Grønmo\textsuperscript{1,2}, Birger Møller-Pedersen\textsuperscript{1}, and Gøran K. Olsen\textsuperscript{2}

\textsuperscript{1} Department of Informatics, University of Oslo, Norway
\textsuperscript{2} SINTEF Information and Communication Technology, Oslo, Norway
{roygr, birger}@ifi.uio.no, goran.k.olsen@sintef.no

Abstract. In this paper we compare three model transformation languages: 1) Concrete syntax-based graph transformation (CGT) which is our emerging model transformation language, 2) Attributed Graph Grammar (AGG) representing traditional graph transformation, and 3) Atlas Transformation Language (ATL) representing model transformation. Our case study is a fairly complicated refactoring of UML activity models. The case study shows that CGT rules are more concise and requires considerably less effort from the modeler, than with AGG and ATL. With AGG and ATL, the transformation modeler needs access to and knowledge of the metamodel and the representation in the abstract syntax. In CGT rules on the other hand, the transformation modeler can concentrate on the familiar concrete syntax of the source and target languages.

1 Introduction

In model-driven engineering, the graphical models are the primary assets, and model-to-model transformations define mappings between models. The leading model-to-model transformation languages, such as Atlas Transformation Language (ATL)\textsuperscript{[6]}, are textual-based programming languages even though the source and target models are graphical models. It is a paradox that the model-driven community promotes the usage of models instead of textual code, while the same community dominantly uses textual code to define the model transformations. Model-to-model transformation languages can handle arbitrary source and target modeling languages, as long as they can be defined in a highly generic metamodeling language, such as Meta-Object Facility\textsuperscript{[9]}.

As opposed to ATL, graph transformations (such as Attributed Graph Grammar (AGG)\textsuperscript{[16]}) provide graphical languages to define model-to-model transformations. Graph transformations are defined upon metamodel elements and visualized with a generic layout, called abstract syntax, where nodes are visualized as rectangles and edges as directed arrows.

The concrete syntax of a modeling language uses a tailored visualization with icons and rendering rules depending on the element types. Concrete syntax-based graph transformation (CGT) is our own emerging model-to-model transformation language that uses graph transformation principles, and where the rules
are defined with concrete syntax. In this paper we investigate one configuration of CGT, with UML 2 activity models [11] as the source and target language (described previously in [3]). In general CGT can be configured with a different source and target language, and is thus comparable with the general model-to-model transformation languages ATL and AGG.

We illustrate the benefits of CGT by investigating a complicated model refactoring problem, which we will refer to as removeGoto. The refactoring task is to translate an activity model with arbitrary goto-like control flow into an activity model with structured loops (the refactoring task is taken from Koehler et al. [7]). We have implemented three alternative solutions for the removeGoto task using: 1) CGT, 2) AGG, and 3) ATL. Our findings can be summarized as follows:

- CGT can handle complex model transformations such as the removeGoto problem.
- The CGT solution to the removeGoto problem is more concise and was made with considerably less effort than the corresponding solutions in AGG and ATL.

2 Remove Unstructured Cycles

Our running example is a business process model of a Web-based shopping application also taken from Koehler et al. [7]. The example is modeled with UML 2 activity models and we only show a submodel of the full business process model. The first model in Figure 1 (labeled 1) shows our source model of the transformation, and represents a business process model with four activities. A Web shopper is allowed to select items (Select activity), configure the chosen items (Configure activity), put chosen items into the shopping cart (Put activity), and to finalize the shopping by leaving with an empty cart or with items to buy (Finish activity). Each control flow between two activities has a two letter guard that reflects the user choice to move from one activity to the next. The two letters are the first letters in the involved activity names.

The transformation task explained below requires that there is no explicit parallelism (no forks), and no implicit parallelism resulting from multiple outgoing control flow edges from the same activity. With no parallelism we can simplify the model, as we have in the figure, by not using explicit decision nodes and interpret multiple outgoing control flow as XOR-behavior. This interpretation is different from the activity model semantics, but is unproblematic since a complete transformation would isolate this simplification to the intermediate models. All the three transformation languages benefit from the simplification.

There are several approaches where the business process models are used to automatically generate BPEL code [13,12] that can be used to execute the business process in a BPEL engine. However, BPEL does not support unstructured cycles, meaning that we need to remove all the unstructured cycles of the activity model before generating BPEL.

In an unstructured cycle there is more than one entry or exit point into or out of the cycle. For instance, model 1 in Figure 1 contains the cycle
Select-Configure-Put-Select which can be exited to the Finish activity from all three activities in the cycle. Another example is the cycle Select-Put-Select which can be entered from both the initial node and the Configure activity, and exited from both activities in the cycle.

Forcing the business process designer to not use unstructured cycles, but structured loops instead, is a heavy burden to put on the business process designer. Avoiding unstructured cycles, as in our starting model, is often a non-trivial and complex task.

Fortunately, an automatic transformation of graphs, like the shopping business process model (Figure 1), into a graph with structured loops (and no unstructured cycles), is well-known from compiler theory. Two tasks, called T1 and T2, can be applied non-deterministically until neither is applicable.

The tasks T1 and T2 will reduce the number of activities and control flow, while expanding the activity nodes from plain activities to become structured activities. We will use the name property of the activities to represent structured activities with arbitrarily many repeat-while and if expressions. At the end we have a single structured activity with no explicit control flow, only hidden...
control flow in the name attribute value (the model with label 5 in Figure 1). It is straightforward to translate from the hidden control flow of \texttt{repeat-while} and \texttt{if} expressions in the name attribute of an activity, into plain activities with explicit control flow, by introducing decision and merge nodes. The end result is then guaranteed to be unstructured cycle free.

In order to apply a transformation based on the tasks T1 and T2, the source model must have the following three characteristics: 1) the model contains at least one unstructured cycle, 2) there is no parallelism, and 3) the model represents a \textit{two-terminal region}. A source model is called a \textit{two-terminal region} if it has a single initial node and a single final node. Our source model in Figure 1 satisfies all the three requirements. According to experience at IBM Zurich, subgraphs with all the three characteristics above, occur frequently in business process designs [7].

One possible transformation process with the tasks T1 and T2, over four steps, is shown in Figure 1. A model is displayed with dashed marking for elements that are replaced in the next transformation step, and $\xrightarrow{T_x}$ denotes the application of task $T_x$, where $x \in \{1, 2\}$.

The task T1 replaces cyclic control flow by \texttt{repeat-while} statements. The task T2 removes an activity with a single predecessor, moves its outgoing control flows to the predecessor activity, adds an \texttt{if} statement to the predecessor activity, and introduces a cyclic control flow to the predecessor activity if there is a "reverse" control flow from the successor to the predecessor. In the application of task T2 from model 1 to model 2, the \texttt{Configure} activity plays the role of a successor node with \texttt{Select} as the single predecessor activity. To ensure that there is at most one control flow in one direction between two activities, we make a combined control flow with \texttt{or} operators between the guards of the control flows.

In three following sections we describe an implementation of the tasks T1 and T2 by CGT rules (section 4), AGG rules (section 5), and ATL transformation modules (section 6). For all the three transformation languages our chosen strategy is to allow multiple control flows in the same direction between two activities in the intermediate models, while we define separate rules to combine multiple control flow edges into one. We do not consider nested activities due to limited space. The following section covers preliminary information about the three model transformation languages.

3 Preliminary

AGG uses graph transformation rules consisting of exactly one \textit{left hand side} (LHS), a (possibly empty) set of \textit{negative application conditions} (NACs), and exactly one \textit{right hand side} (RHS). The LHS defines a subgraph for which we are looking for matches within the graph to be transformed. A NAC prevents application of a rule if the LHS combined with the NAC has a match. None of
the NACs can have a match in order to apply a rule. When a rule is matched by a LHS, then the matched LHS within the source graph is replaced by the RHS of the matched rule.

The dangling condition [4] ensures that a rule, involving node deletion, is only applied when there will be no dangling edges in the resulting graph. Identifiers, displayed with a number followed by a colon, e.g. 1:Activity, are shown in the rule when there are shared elements between the LHS and the RHS/NACs. Elements that are shared between the LHS and the RHS are preserved by the rule. Elements where we have not displayed an identifier, occur either only in the LHS and will be deleted, or they occur only in the RHS and will be added.

CGT is basically the same as graph transformation, except that rules use concrete syntax instead of the abstract syntax. In addition CGT defines a collection operator which is explained below. In CGT, identifiers are displayed next to the elements (e.g. id=1), and attribute variables are prefixed with a question mark (e.g. ?guard1) and displayed in the same position as the attribute within the concrete syntax.

A collection operator can be used in a graph transformation rule to match a set of similar subgraphs. In the right part of Figure 2, we see how an activity model with a redundant decision node can be refactored by combining two guard expressions with an and operator [5,3]. With plain graph transformation it is not possible to express the removal of redundant decision nodes with a single rule. In the left part of Figure 2, a single CGT rule with the collection operator (dashed line frame) is sufficient to do the refactoring. The collection operator matches an arbitrary number of subgraphs (the doA and doB branches), removes a decision node and a merge node, and combines two guards for each subgraph match.

In the ATL code, transformation modules (hereafter called modules for short) declare the imported metamodel of the source and target models to be used in a transformation. Rules are used to implicitly match source elements and produce target elements, lazy rules are called explicitly to produce target elements based on source elements, and helpers represent user-defined functions with return values that does not produce target elements. ATL is built around the Object Constraint Language (OCL) [10] with some additional predefined functions.

![Fig. 2. Removing redundant decision nodes](image-url)
4 RemoveGOTO by CGT

In Figure 3 we have defined five rules to simulate the tasks T1 and T2. Since we use the concrete syntax to define the rules, they resemble the transformation steps from Figure 1.

A single rule is sufficient to simulate the task T1. The LHS expresses that we are looking for matches of arbitrary activities with a cyclic control flow. \(\text{id} = 1\) is an identifier of the matched activity, and \(?\text{guard}\) is an identifier of the guard value of the cyclic control flow. The NAC ensures that the matched activity has exactly one cyclic control flow. The RHS expresses the replacement of the LHS match, implicitly meaning that the cyclic control flow is removed, and the activity name is extended with a \texttt{repeat-while} expression.

To simulate the task T2, we define two rules depending on the node type(s) following the successor activity. Either the next node(s) is the final node or activity node(s). In both cases a "reverse" control flow going from the successor activity back to the predecessor activity shall result in a cyclic control flow of the predecessor activity, where the guards are combined with an \texttt{and} operator. A collection operator with cardinality 0..1 expresses that such a "reverse" control flow is either present or not.

For both T2 rules the predecessor activity name is extended by the same \texttt{if}-expression. For the \texttt{T2-NextIsFinal} rule the predecessor activity gets an

![Fig. 3. RemoveGoto: Rules using concrete syntax (CGT)](image_url)
outgoing control flow to the final node. The T2-NextIsActivity is a bit more complicated. Here we need to move each outgoing control flow of the successor activity over to the predecessor activity, and the guard of the new control flow is extended with an and operator between two successive guards. A collection operator with cardinality 1..* expresses that there are arbitrarily many such outgoing control flows from the successor activity.

Note that we have not defined a NAC to ensure that the successor activity has exactly one predecessor activity, since this is ensured by the dangling condition. Otherwise an additional incoming control flow to the to-be-deleted successor activity would become a dangling edge.

Finally, we need two simple rules to define: 1) the merging of two cyclic control flows into one control flow (RemMultiCircEdge), and 2) the merging of two control flows in the same direction between two distinct activities (RemMultiEdge). The merged control flow uses an or operator to combine the guards of the joined control flows.

The two rules to merge multiple control flow rules should always be applied after each application of a T1 or a T2 rule. However, no specific control flow ordering of the rules is necessary due to 1) the dangling condition for the two T2 rules, and 2) the NAC of the T1 rule. The NAC of the T1 rule implies that the RemMultiCircEdge rule must be applied as long as possible first on the relevant activity, while the dangling condition on the to-be-deleted successor activity ensures that the RemMultiEdge rule is applied before the T2 rules.

5 RemoveGOTO by AGG

While the CGT transformations uses the concrete syntax, graph transformations (such as AGG) uses the abstract syntax. In addition, AGG does not have a collection operator. Thus, we get several rules for a single CGT rule using the collection operator. An automated mapping from a CGT rule to AGG rules is described in our earlier work [3].

We will use the AGG rules that result directly from our mapping of CGT rules into AGG rules. To be fair to AGG we manually investigated the generated rules to see if they could be optimized or further improved. No such improvements were found, and in fact the generated rules were fewer than a previous attempt where we coded the rules manually in AGG.

We get eight AGG rules (Figure 4 and Figure 5) corresponding to the five CGT rules. The three CGT rules without collection operators (Figure 5) are simply translated from concrete to abstract syntax.

The CGT rule, named T2-NextIsActivity, has two collection operators and is mapped to three transactional rules in AGG. The Iter-1 rule represents the 0..1 collection with the "reverse" control flow. The Iter-2 rule represents the 1..* collection with the arbitrary number of outgoing control flow from the successor activity. The Final rule deletes the successor activity. For both the iteration rules we get autogenerated NACs to exclude matches when there are multiple predecessors for the successor activity.
The CGT rule, named T2-NextIsFinal, has one collection operator and is mapped to two transactional rules in AGG. The T2-NextIsFinal-Iter rule will replace a "reverse" control flow by a cyclic control flow of the predecessor activity, and it gets a NAC to prevent multiple predecessors. The T2-NextIsFinal-Final rule deletes the successor activity.

We need additional Java code to control the rule application order. The set of rules (e.g. T2-NextIsFinal-Iter and T2-NextIsFinal-Final) corresponding to a single rule with collection operators shall be applied as one transactional group. The Iter rules are applied first and at least the minimum cardinality
number of times, and as long as possible (or up to the maximum cardinality if it is different from \(\ast\)). The Final rule must then be applied.

6 RemoveGOTO by ATL

We use three ATL modules to accomplish the removeGOTO task: T1, T2 and RemMultiEdges. The T1/T2 module performs the task T1/T2 with the simplification also used by CGT and AGG to allow creation of multiple control flow in the same direction between two activities, and to allow creation of multiple circular control flows for the same activity. The RemMultiEdges module removes all such multiple control flow edges. Since the removeGOTO transformation is a refactoring task, the target model should keep all the unchanged parts from the source model. This is achieved by using the publicly available UML2Copy.atl module [15] that has rules to copy all UML source elements into the target model. Our three modules are then superimposed on the UML2Copy in order to override only the rules where things are changed.

Due to limited space we have made some abbreviations in the ATL code listings: UML2! is skipped as prefix on metamodel types, Action instead of UML2!CallOperationAction, CFlow instead of UML2!ControlFlow, LString instead of UML2!LiteralString, Node instead of UML2!ControlNode, and thisModule is skipped as prefix when calling helper functions. We have also left out a few rules, all the helpers implementation code in the RemMultiEdges and T2 modules, and left out a large number of attributes that should be copied from the source to the target. We discuss some of these details after the code extracts of the modules are presented.

The T1 module consists of one main rule, Action, where we produce a repeat-while statement in the Action name, for each circular control flow. The T1 module will remove all circular control flow at once, and thus corresponds to
multiple applications of the T1 rule of CGT/AGG. A precondition for calling T1 is that there is at most one circular control flow of each Action element. This precondition is ensured by calling the RemMultiEdges module after each application of a T1 or T2 module.

Listing 1.1. ATL code extract from the T1 module

```atl
rule Action { from s: Action to t: Action ( name <- if s.incoming->excludesAll(s.outgoing) then s.name else 'REPEAT ' + s.name + ' WHILE ' + loopGuard(s) + ';' endif) }

helper def: loopGuard(act: Action): String = act.incoming->asSet()->intersection(act.outgoing)->asSequence() ->first().guard.value;
```

The RemMultiEdges module (not listed due to limited space) removes all multiple control flow occurrences, both circular and non-circular. In the main rule, Activity, we calculate all combinations of two activity nodes (n1,n2), where n1 and n2 may be the same node. If there exists a control flow edge between these two nodes, then a single, possibly combined control flow is produced from n1 to n2 by a call to the rule makeOneEdge. The combined control flow uses an or operator between the guards from the replaced multiple control flows.

The T2 module defines the Action rule to ignore successor nodes from the target model (guarded by not isSucc(s)). For the remaining Action elements, the name is kept unchanged if the Action source object is different from the predecessor node, and for predecessors the original name is replaced by an if-statement. If the "reverse" control flow exists from successor to predecessor, then an explicit call to the circEdge rule produces a circular control flow for the predecessor activity. All other outgoing control flows of the successor activity are moved over to the predecessor node, with extended guard values, by calls to the newNextEdge rule. Both the circEdge and newNextEdge lazy rules call on associated rules (circGuard and nextGuard) for combining guards by and operators.

The T2 module uses the helper functions for which the signatures are shown in the listing below. It is important to stress that we cannot safely apply the task T2 on multiple matches at the same time. We need to ensure that at most one node is treated as predecessor and at most one node is treated as successor. We use the predefined ATL method indexOf to get a unique index of each activity, and we choose only the successor with the lowest index. There is no rationale behind this choice except to ensure that we get only one successor activity, and then the rules also ensure that we get a single predecessor activity.

Since all the three modules deletes some of the source control flow, we need CFlow rules to override the default behavior of copying all the the CFlow. In the
Listing 1.2. ATL code extract from the T2 module

```
rule Activity { from s: Activity to t: Activity ( name <- s.name, node <- s.node, edge <- s.edge,  -- circular pre edge if edge from succ to pre edge <- s.edge->select(cf|isSucc(cf.source) and role(cf.target) = 'pre') ->collect(cf| circEdge(cf)),  -- outgoing edges of succ is moved to pre edge <- s.edge->select(cf|isSucc(cf.source) and role(cf.target) <> 'pre') ->collect(cf| newNextEdge(cf)))}

rule Action { from s:Action(not isSucc(s)) to t:Action ( name <- if role(s) = 'pre' then s.name+ IF ' +guardPreToSucc(s)+ ' '+succName(s)+ ' ENDIF ; ' else s.name endif )}

lazy rule circEdge { from revCF: CFlow to t: CFlow ( source <- revCF.target, target <- revCF.target, guard <- circGuard(revCF.target) ) }

lazy rule circGuard { from pre:Action to guard:LString ( value <- '(' + guardPreToSucc(pre)+ ' AND ' + guardSuccToPre(pre) + ')')

lazy rule newNextEdge { -- move edge from succ to pre from toNext : CFlow to t: CFlow ( source<-toNext.source.incoming->first().source, — pre target <-toNext.target, — next guard <- if toNext.guard.oclIsUndefined () then OclUndefined else nextGuard(toNext) endif)

lazy rule nextGuard { from next:CFlow to guard:LString ( value<- '(' + next.source.incoming->first().guard.value + ' AND ' + next.guard.value + ')') }
```

listing below we show how the T1 module adds a rule guard to ignore circular control flow. Similarly for the string value (called LString) of the corresponding control flow guard, we need to avoid producing target elements.

As a shortcut in our code we have assumed that all guard values are registered as literal strings, while in general there are a number of possibilities. While these are treated generically in CGT and AGG, we need additional rules (similar to LString) to handle all these other value types (e.g. integer, boolean, time).
Listing 1.3. Helpers from the T2 module

```plaintext
helper def: role(a: Action): String = ...
−− The role of a node is 'pre', 'succ' or 'other'.

helper def: isSucc(a: Action): Boolean = ...
−− match only the lowest indexed succ node

helper def: guardPreToSucc(pre: Action): String = ...
−− returns true for all candidate succ nodes

helper def: isWeakSucc(a: Action): Boolean = ...

helper def: guardSuccToPre(pre: Action): String = ...
−− each action element has a unique index

helper def: nodeIndex(a: Action): Integer = ...

helper def: noLowerIndexSucc(a: Action): Boolean = ...
```

For the module RemMultiEdges to combine multiple control flow edges into one, the guard of CFlow and associated LString is set to false. This is because no source control flow is preserved and all the resulting control flow is explicitly produced. The guard of the rule CFlow in the module T2 ensures that control flow connected to the to-be-deleted successor activity is omitted in the target model.

Listing 1.4. CFlow and LString rules in the T1 module

```plaintext
rule CFlow{from s:CFlow (s.source <> s.target) to t:CFlow
 (−− copy all attrs ) }

rule LString { from inGuard: LString
 (not (inGuard->parent()).oclIsTypeOf(CFlow)) or
 (inGuard->refImmediateComposite()().source <>
 inGuard->refImmediateComposite()().target )
to outGuard: LString (−− copy all attrs ) }
```

The ATL code listings are simplified because we need to copy all the source element properties (e.g. Activity has 46 properties and ControlFlow has 14 properties), just like UML2Copy already does. This could be avoided if there was a way to execute the existing body of the superimposed rule, but this is not possible. Inheritance can only be achieved on rules within the same transformation module. The same problem does not occur with CGT/AGG since unmatched elements and properties are preserved by default.

The three modules must be invoked in a controlled manner. The modules T1 and T2 must be applied as long as possible, but after each application of T1 or T2, we have to call the module RemMultiEdges before we continue. This module application order is shown in Figure 6. The ATL transformation modules can be invoked from Java code, and it is trivial to translate the control flow in
Figure 6 into Java code by using if statements for the decision nodes, and a do-while statement for the outer loop. The data flow is omitted from the figure, but basically each transformation module takes an activity model as input and returns an activity model as output. In addition there will be some data objects involved to test if a transformation module has been applied or not. This can be achieved in many ways, for instance checking if the number of control flow edges are reduced by $T_1$, or if the number of activities are reduced by $T_2$.

7 Discussion

Our paper example can be compared with respect to the amount of code or models that a programmer or modeler needs to specify in order to make an executable solution. While the CGT solution is expressed with half a paper page of graphical models, the AGG solution needs one page of graphical models. In addition AGG needs rule application order code to ensure the transactional behavior of the iteration/final rules. The ATL solution (excluding the UML2Copy.atl) would use about 4.5 pages (160 code lines without empty lines and comments) of textual code without the rule application order code, and without the copying of unchanged attributes which would make the complete ATL solution several pages longer. So, CGT is much better than AGG, and AGG is much better than ATL with respect to the solution size.

The usage of the collection operator in the concrete syntax of CGT raises the level of abstraction compared to the collection free AGG rules. This is why we get fewer rules in CGT than in AGG. The ATL solution gets very large compared to CGT/AGG since we need: 1) to copy all the unchanged properties of elements that are modified, 2) several helper methods to ensure that the $T_2$ module matches only one activity as a predecessor and only one activity as the successor for each application (otherwise the result may be corrupted).

CGT appears to be more intuitive since it uses the concrete syntax of activity models which is already familiar to the modeler. Even though AGG is graphical, it takes more time to understand what the diagrams express since they are defined on the abstract syntax. It is clearly a benefit for a modeler to define an activity model by using a concrete syntax-based editor. The same benefit applies when rules are defined as model extracts in LHS/RHS.
The transformation modeler does not need to know anything about the representation of the UML activity metamodel when defining CGT rules, while access to and knowledge of the metamodel and the abstract syntax is essential when using AGG or ATL. This largely increases the overall complexity of using AGG and ATL compared to CGT. While a control flow is simply drawn in CGT as within activity models, the modeler need to know how a control flow is represented in AGG and ATL. In AGG we represented a control flow with a node of type cFlow, and two outgoing edges labeled src and trg going to the source and target nodes of the control flow. In the UML 2 metamodel we used in ATL, the incoming and outgoing properties of the activity node provides the set of incoming and outgoing control flow references.

ATL has additional weaknesses compared to CGT and AGG for refactoring transformations like the paper example. While CGT and AGG defines only the changes to the source model, and preserves all the unchanged parts by default, this is very cumbersome in ATL. In the ATL 2006 version, a refining mode was introduced, with the intention to preserve unchanged parts. But it currently has limitations and does not behave as expected. We still have to define a copy module with default copying of all elements. With UML, one was publicly available (UML2Copy.atl), which saved a lot of effort, but this will generally not be the case. Upon this copy module, we need to superimpose our actual module. In ATL there is no way to change only a few properties of an element, and to keep the other attributes unchanged. All the properties must be explicitly listed. This is because we cannot use inheritance combined with superimposition.

With refactoring it is useful to apply a set of rules for as long as possible. This is directly supported by CGT and AGG, but not by ATL.

CGT is one configuration of concrete syntax-based graph transformation, and its implementation has been hard coded. Full support for concrete syntax-based graph transformation to support general model transformations requires more tool implementation and also some initial configuration where the user defines the relation between the concrete and abstract syntax of the source/target language. Thus, AGG and ATL have a benefit, compared to concrete syntax-based graph transformation, by having existing tools that support general model transformations.

We have developed a proof-of-concept Eclipse GMF-based [2] rule editor for the configuration of CGT where activity models is the source and target language [3]. The transformation from CGT to AGG rules has been implemented using the MOFScript language [8]. We have not implemented the transactional support needed to generally ensure a correct simulation of the collection operator. In the removeGOTO solution it was sufficient to add a few NACs, and to apply all the rules non-deterministically.

For the ATL code implementation of the removeGOTO solution, we did not implement the Java code to control rule application order, but tested several orders manually. The generated AGG rules, and the ATL rules were tested successfully on four different models including the source model of Figure 1.
8 Related Work

Strommer et al. [14] also aim at using the concrete syntax of the source and target languages to define model transformations. Our transformation definitions are completely defined upon the concrete syntax, while they generate a starting point that must be further modified and extended within the ATL language and the abstract syntax of the source and target.

The removeGOTO problem used in this paper is taken from Koehler et al. [7]. They have an OCL-based solution to the problem, which has many of the same drawbacks that ATL has. In addition pure OCL tends to be more complicated than ATL since the rule construct of ATL and additional functions is an improvement over pure OCL.

The TIGER tool [1] and the MATA tool [17] use concrete syntax in their transformation rules. These rules are then mapped to abstract syntax rules that are executed in a traditional graph transformation tool. As opposed to our approach, the focus with TIGER and MATA seems so far to be restricted to model transformations where the source and target languages are the same. Neither TIGER nor MATA support a similar concept as the collection operator.

9 Conclusions

In concrete syntax-based graph transformation, the transformation modeler can concentrate on rules directly within the familiar concrete syntaxes of the source and target modeling languages. With AGG and ATL, on the other hand, the transformation modeler must master several related languages: 1) the concrete syntax of source and target, 2) the metamodel of source and target, and 3) the abstract syntax of source and target.

This paper presents three solutions (in CGT, AGG, and ATL) to a fairly complicated refactoring example of activity models. The conclusion is that CGT in this case requires less effort and is the most concise solution of the three. In addition to the usage of concrete syntax, the usage of a collection operator is the reason why CGT outcompetes the other two. The collection operator can also be used on the abstract syntax of traditional graph transformation, which would reduce the disadvantage of using traditional graph transformation like AGG.

Concrete syntax-based graph transformation (with the collection operator) is at a higher level of abstraction than traditional graph transformation and model transformation. While this is a benefit for the users, it also requires more tools and infrastructure than is available today. As a simplified view, the missing infrastructure can be seen as a compiler that translates concrete syntax rules into abstract syntax rules. For each new combination of source and target, the user also needs to define the relationship between abstract and concrete syntax which is not needed in the traditional approaches. It is a future goal to implement full tool support for CGT.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project grant no. 215412 (EU FP7 STREP).
References

Appendix F

Paper 5: A Collection Operator for Graph Transformation
A Collection Operator for Graph Transformation

Roy Grønmo¹,², Stein Krogdahl¹, and Birger Møller-Pedersen¹

¹ Department of Informatics, University of Oslo, Norway
² SINTEF Information and Communication Technology, Oslo, Norway
{roygr,birger,steinkr}@ifi.uio.no

Abstract. Graph transformation has a well-established theory and associated tools that can be used to perform model transformations. However, the lack of a construct to match and transform collections of similar subgraphs makes graph transformation complex or even impractical to use in a number of transformation cases. This is addressed in this paper, by defining a collection operator which is powerful, yet fairly simple to model and understand. We present model transformation examples from different modeling domains to illustrate the benefit of the approach.

1 Introduction

Graph transformations have been proposed by several authors as a means to perform model transformations [4,7]. The graphical way to define graph transformations, the available tool support [10,23,25], and the well-established theory including termination and confluence analysis [16,21] makes graph transformation appealing.

The graph concept is based on nodes and directed edges from which we can define models. Many model transformations can then be defined by a set of graph transformation rules, where each rule consists of a left hand side (LHS) graph, a right hand side (RHS) graph, and an interface (I) graph. The elements in the interface graph are to be preserved, the elements in LHS \ I are to be deleted, and the elements in RHS \ I are to be added.

The minimalistic nature of graph transformation is probably a key factor to its success, since it makes it easier to implement tools and to establish theory on its concepts. For the graph transformation designer, on the other hand, the lack of higher level constructs reduces the usability of graph transformation. This is why some authors have proposed to raise the level of abstraction by introducing new and powerful graph transformation mechanisms, e.g. the star operator [17] and recursion [12]. Our experience on a number of graph transformation examples reveals an often occurring need to match collections of similar subgraphs, which is addressed by our collection operator. The collection operator allows us to express a fairly powerful model transformation using a single rule.

Fujaba [10] and PROGRES [23] have support for matching collections of single nodes only. In many cases this is too restrictive and a lot of recent approaches [1,5,9,14,18,22] address this by allowing to match collections of subgraphs. Our collection operator aims to be concise and easy to use for the rule designer,
and at the same time expressive enough for many typical model transformation scenarios.

The paper is structured as follows. Section 2 provides the formal foundation of graph transformation; Section 3 presents the collection operator; Section 4 shows how complicated it is to simulate a rule with collection operators by multiple collection free rules in the AGG graph transformation tool; Section 5 shows three example rules with collection operators; Section 6 covers related work; and Section 7 concludes.

2 Graph Transformation

Below we provide the known formal foundation of graph transformation [15].

Definition 1 (Graph and graph morphism). A graph \( G = (G_N, G_E, src, trg) \) consists of a set \( G_N \) of nodes, a set \( G_E \) of edges, two mappings \( src, trg : G_E \rightarrow G_N \), assigning to each edge \( e \in G_E \) its source node \( src(e) \in G_N \) and target node \( trg(e) \in G_N \). A graph morphism \( f : G_1 \rightarrow G_2 \) from one graph to another, with \( G_i = (G_{E,i}, G_{N,i}, src_i, trg_i), (i = 1, 2) \), is a pair \( f = (f_E : G_{E,1} \rightarrow G_{E,2}, f_N : G_{N,1} \rightarrow G_{N,2}) \) of mappings, such that \( f_N \circ src_1 = src_2 \circ f_E \) and \( f_N \circ trg_1 = trg_2 \circ f_E \) (preserve source and target).

A graph morphism \( f : G_1 \rightarrow G_2 \) is injective if \( f_N \) and \( f_E \) are injective mappings. Only injective graph morphisms will be relevant in this paper.

Definition 2 (Rule). A graph transformation rule \( p : L \xleftarrow{\ell} I \xrightarrow{r} R \) consists of three graphs \( L \) (LHS), \( I \) (Interface) and \( R \) (RHS) and a pair of injective graph morphisms \( l : I \rightarrow L \) and \( r : I \rightarrow R \).

Definition 3 (Match). Given a rule \( p : L \xleftarrow{\ell} I \xrightarrow{r} R \) and a graph \( G \). Then an occurrence of \( L \) in \( G \), i.e. an injective graph morphism \( m : L \rightarrow G \), is called match. The function \( isMatch : L, G, (L \rightarrow G) \rightarrow \text{Bool} \) returns true if and only if \( L \rightarrow G \) is a match from \( L \) to \( G \). A match \( m \) for rule \( p \) satisfies the dangling condition if no node in \( m(L \setminus l(I)) \) is incident to an edge in \( G \setminus m(L \setminus l(I)) \).

Definition 4 (Derivation Step). Given a graph \( G \), a graph transformation rule \( p : L \xleftarrow{\ell} I \xrightarrow{r} R \), and a match \( m : L \rightarrow G \), then there exists a derivation step from the graph \( G \) to the graph \( H \) if and only if the dangling condition is satisfied. \( H \) is constructed as follows:

1. Remove the image of the non-interface elements of \( L \) in \( G \), i.e. \( H' = G \setminus m(L \setminus l(I)) \).
2. Add the non-interface elements of \( R \) into \( H \), i.e. \( H = H' \cup (R \setminus r(I)) \).

A negative application condition [15] is an extension of the LHS which prevents matches from being applied in a derivation step.

Definition 5 (Negative application condition (NAC)). A NAC for a graph transformation rule \( L \xleftarrow{\ell} I \xrightarrow{r} R \), is defined by a pair of injective graph morphisms:
$L \leftarrow^s NI \rightarrow^t N$, where $N$ is the negative graph, and $NI$ defines the interface graph between $L$ and $N$. A match $m : L \rightarrow G$ satisfies the NAC if and only if there does not exist an injective graph morphism $n : N \rightarrow G$ which preserves the $NI$ interface mappings, i.e. for all nodes $v$ in $NI$ we have $n_N(t_N(v)) = m_N(s_N(v))$ and for all edges $e$ in $NI$ we have $n_E(t_E(v)) = m_E(s_E(e))$. A rule can have an arbitrary number of NACs, and a derivation step can only be applied if a match satisfies all the NACs of the matched rule.

In addition to the above, we adopt the theory of typed attributed graphs [13], where graphs are extended by assigning types to nodes and edges, and by assigning a set of named attributes to each node type. A graph morphism must now also preserve the node and edge types, and the attribute values.

In the graph transformation rules throughout this paper we only explicitly display the LHS and the RHS graphs, while the interface graph is given by shared identifiers of elements in the LHS and the RHS. Such identifiers are displayed next to its element.

2.1 Concrete and Abstract Syntaxes

Typed attributed graphs are rich enough to represent most of today’s modeling languages in a natural way. These graphs use a generic graphical layout, called abstract syntax, where nodes are visualized as rectangles containing the type name and a list of attribute values, and edges are visualized as directed arrows with the type name. The concrete syntax of a modeling language uses a tailored visualization with icons and rendering rules depending on the element types. To improve the usability for the graph transformation designer, we define the transformation rules upon concrete syntax. The transformation designer can think entirely in the concrete syntax, while the matching and derivation steps are still carried out in the abstract syntax.

In a natural translation of UML activity models [19] to typed attributed graphs, an activity in the concrete syntax corresponds to a node of type Activity in the abstract syntax. A control flow in the concrete syntax corresponds to a node of type CFlow and two edges of types src and trg. The figure below shows an activity model concrete syntax on the left and the corresponding abstract syntax on the right.

<table>
<thead>
<tr>
<th>models: concrete syntax</th>
<th>graphs: abstract syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^C \rightarrow^C T^C$</td>
<td>$S^A \rightarrow^A T^A$</td>
</tr>
<tr>
<td>S=source, T=target, r=rule</td>
<td></td>
</tr>
<tr>
<td>conceptual transformation</td>
<td></td>
</tr>
<tr>
<td>actual transformation</td>
<td></td>
</tr>
<tr>
<td>+ translations</td>
<td></td>
</tr>
</tbody>
</table>

Our examples use the concrete syntax, while the formalization is defined on the abstract syntax. We assume that the translation from concrete to abstract syntax (c2a), and the opposite direction (a2c), is already defined for the relevant modeling languages. Then we can link concrete syntax-based graph transformation to abstract (and traditional) syntax-based graph transformation in a systematic...
way: 1) translate the concrete syntaxes of source model and rules (consisting of \(L, I, R, NI\), and \(N\) models) into abstract syntax graphs by \(c2a\), 2) apply the abstract syntax graph transformation rules on the source graph, and 3) translate the resulting abstract syntax graph to a concrete model by \(a2c\).

Linking concrete syntax-based graph transformation to the traditional graph transformation has been successfully applied in our previous work [11] and by other authors [3,26]. With a large number of modeling languages, including those illustrated in this paper, the same translation (\(c2a\)) is reasonable to use for both the rules and the source model. When the same translation is used for both the rules and the source model, the principles of graph transformation can be directly applied at the concrete syntax level, and the transformation designer does not have to care about the underlying translations to graphs at the lower abstraction level.

3 The Collection Operator

We propose a collection operator that can be used in a graph transformation rule to match and transform a set of similar subgraphs in one step. Figure 1 illustrates the collection operator in a workflow refactoring example [6,11]. The source model (labeled 1) is an activity model with two consecutive decision nodes (displayed as diamond symbols), and two inner paths leading to the activities named doA and doB. The refactored model (labeled 2) shows that the two decision nodes can be combined into one.

Since there can be an arbitrary number of inner paths, plain graph transformation as defined above cannot express the removal of a redundant decision node with

![Fig. 1. Activity model refactoring: Removing redundant decision node](image)

**Fig. 1. Activity model refactoring: Removing redundant decision node**

![Fig. 2. Semantics of the collection operator](image)

**Fig. 2. Semantics of the collection operator**
a single rule. In the right part of Figure 1 a single rule with the collection operator (dashed line frame) is sufficient to do the refactoring. The collection operator matches an arbitrary number of subgraphs, which all have an inner path leading to a single activity node between the inner decision and merge nodes. The outer guard (?guard1) is combined with each inner guard (?guard2) by using and operators. Notice that a matched subgraph at the abstract syntax level contains three nodes (one node of type Activity and two nodes of type CFlow) and associated edges.

The size of the collection match must be greater or equal to the lower bound cardinality (1 in the example) in order to apply a rule. A collection match size is non-deterministically increased until we reach the upper bound (no limit in the example) or there are no more possible subgraph matches. The parts outside of the collection operator must occur only once in a rule match.

Identifiers (e.g. id=1) are associated to the main elements such as activities, control nodes and control flow. Attribute variables (e.g. ?guard2) are associated with the values of attributes such as activity name and guard. An identifier/variable inside a collection represents a set of identifiers/variables.

In Figure 2 we use a simple concrete syntax of named circles connected by arrows to show the relationship between the collection operator in the LHS and possible matches. In case a) only the arrow is inside the collection while the source and target circles are outside the collection. This means that possible matches have a set of arrows between the same two circles. In case b) the circle named A is also inside the collection which means that a match contains a set of distinct A circles with arrows leading to the same target circle named B. In case c) both the circles are inside the collection which means that a match contains a set of distinct circles each having their own arrow.

A NAC and the RHS can only use collection operators that are introduced in the LHS. The RHS indicates the changes to each subgraph match in the collection, and the cardinality of the same collection operator must be the same in the LHS and the RHS/NACs. The actual matching collection size of the LHS leads to the same collection instantiation size within the RHS/NACs. If the collection operator is absent in the RHS, then it implies a deletion of all the matched collection subgraphs.

A collection operator has an identifier which is visualized next to the collection frame. No collection identifier visualization is needed in cases where the collection operator is uniquely identified by its cardinality, or when the rule has only one collection operator (e.g. Figure 1). To avoid complexity we disallow collection operators to be overlapping or nested in the abstract syntax, which also implies that two collection operators cannot be adjacent in the concrete syntax. Otherwise the two collection operators would include at least one common edge in the abstract syntax. This fact becomes clearer in the following subsection.

3.1 Mapping Collection Operator from Concrete to Abstract Syntax

In the translation from concrete syntax rules to abstract syntax rules, we must determine which abstract syntax elements belong to the collection. This is
illustrated in the figure below with a collection operator in concrete syntax to the left, and corresponding abstract syntax to the right:

If an element is inside a collection in the concrete syntax, then a corresponding node goes inside the collection in the abstract syntax (e.g. leftmost Activity node and CFlow node). An edge connecting two nodes that are both inside a collection, belongs to the collection (e.g. src edge).

An edge in the abstract syntax connecting a collection node to a non-collection node must also be included in the collection (e.g. trg edge). This is because all edges shall have exactly one source and one target node (note: source and target should not be confused with the example edges of type src and trg). Otherwise, with a non-collection edge incident to a collection node, the only possible collection cardinality is $1..1$, which implies that the collection is redundant.

3.2 Collection Operator Formalized

A collection operator can be represented in graphs as a node of type coll, with min and max as cardinality attributes, and with a set of edges targeting all the collection subgraph nodes. The set of all collection operators in a rule $p$ is referred to as $Coll_p$. We use $\psi$ to denote a function that maps each collection operator in a rule $p$, to a number within its cardinality range, i.e. $\psi : Coll_p \rightarrow (\mathbb{N} = \{0, 1, 2, \ldots\})$, where $\forall c \in Coll_p : \psi(c) \in [c.min, c.max]$.

For a rule $p : L \leftarrow I \rightarrow R$ with at least one collection operator, we let $p^\psi : L^\psi \leftarrow I^\psi \rightarrow R^\psi$ denote the collection free rule where all collection operators in $p$ are replaced by the $\psi$ mapped number of collection content copies. In these copies all the copied elements/attributes get fresh identifiers/variables respectively, while the interface elements between the LHS and the RHS are maintained. Similarly, $L^\psi \leftarrow NI^\psi \rightarrow N^\psi$ denotes a collection free NAC.

Figure 3 shows $p^\psi$, where $c_1$ is the collection operator in the transformation rule $p$ representing the redundant decision node example from Figure 1, and $\psi(c_1) = 2$.

```
Fig. 3. The rule for activity model refactoring with 2 as the collection size
```
Definition 6 (Extensions). Given a rule \( p : L \leftarrow I \rightarrow R \) with at least one collection and a graph \( G \). A collection cardinality mapping \( \psi^+ \) extends the cardinality mapping \( \psi \) (denoted \( \psi^+ \succ_p \psi \)) if and only if there is at least one greater collection cardinality and none of the collection cardinalities are smaller:

\[
\exists c \in \text{Coll}_p : \psi^+(c) > \psi(c) \land \forall c \in \text{Coll}_p : \psi^+(c) \geq \psi(c)
\]

A rule \( p^\psi^+ \) extends the rule \( p^\psi \) (denoted \( \psi^+ \supset \psi \)) if and only if the following holds: \( \psi^+ \succ_p \psi \) and the three graphs \( L^\psi^+, I^\psi^+, \) and \( R^\psi^+ \) contains respectively \( L^\psi, I^\psi, \) and \( R^\psi \) as subgraphs.

An injective morphism \( m^\psi^+ : L^\psi^+ \rightarrow G \) extends the injective morphism \( m^\psi : L^\psi \rightarrow G \) (denoted \( m^\psi^+ \supset m^\psi \)) if and only if \( p^\psi^+ \supset p^\psi \) and \( m^\psi^+ (L^\psi) = m^\psi (L^\psi) \).

Definition 7 (Match for a rule with collections (cMatch)). Given a rule \( p : L \leftarrow I \rightarrow R \) with at least one collection, a graph \( G \), and a collection cardinality mapping \( \psi \). An injective morphism \( m^\psi : L^\psi \rightarrow G \) is a cMatch of rule \( p \) in \( G \) if and only if \( m^\psi \) is a non-extendable injective morphism in \( G \). Formally,

\[
isCMatch(L, G, \psi, m^\psi, L^\psi) \overset{\text{def}}{=} isMatch(L^\psi, G, m^\psi) \land \forall m^{\psi^+} \in (L^\psi^+ \rightarrow G) : (m^{\psi^+} \supset m^\psi) \land isMatch(L^\psi^+, G, m^{\psi^+})
\]

When we have a cMatch \( m^\psi : L^\psi \rightarrow G \) for a rule \( p \) with collections, then \( m^\psi \) is also a match in the collection free rule \( p^\psi : L^\psi \leftarrow I^\psi \rightarrow R^\psi \) where Def. 4 for derivation steps is still valid. We also get collection free NAC definitions as \( L^\psi \leftarrow N^\psi \rightarrow N^\psi \), where Def. 5 applies.

3.3 Inherent Tool Support for Rules with Collection Operators

This section describes how we can provide tool support for rules with collection operators. The minimal configuration of \( \psi \) for which we can find a cMatch for a rule \( p \) with collection operators, is when \( \forall c \in \text{Coll}_p : \psi(c) = c.\min \). We refer to this minimal configuration of \( \psi \) as \( \psi^- \). Given a rule \( p \) with collection operators and a graph \( G \), the following steps can be used to find a cMatch in \( p \) and try to apply a derivation step for that cMatch:

1. Look for an injective morphism \( m^{\psi^-} : L^{\psi^-} \rightarrow G \) in the collection free rule \( p^{\psi^-} \).
2. Extend (if possible) the injective morphism \( m^{\psi^-} \) until it is a non-extendable injective morphism \( m^\psi : L^\psi \rightarrow G \), i.e. a cMatch for \( p \). The extension process can be achieved by iterating over each collection operator \( c \in \text{Coll}_p \) and increasing \( \psi(c) \) as much as possible. \( \psi(c) \) can only be increased by 1, if the injective morphism can be extended with an additional subgraph match of the collection content in \( c \).
3. Apply a derivation step with the collection free rule $p^\psi$ and the match $m^\psi$ if $m^\psi$ satisfies all the NACs and the dangling condition.

We use a transformation task of state machine refactoring [24] to illustrate the proposed matching process above. The refactoring applies to cases where all the inner states of a composite state have outgoing transitions to the same state, and all these outgoing transitions share the same trigger and effect, while the guards must all be undefined or equivalent. In such cases we can replace all these outgoing transitions by a single transition from the composite state to the external state.

The top left part of Figure 4 shows an example state machine modeling the behavior of a smartphone (based on [4]). The state called Idle represents a waiting state of a smartphone. The signal phoneMode triggers a composite state named Active in which we can make phone calls. All the inner states have a trigger with the same outgoing trigger hangUp targeting the outer Idle state.

Fig. 4. State machine refactoring: phone example (top), transformation rule (middle), matching process (bottom)
The middle part of Figure 4 shows a transformation rule, named \( p \), that defines the refactoring. The rule uses a collection operator with the id \( c_1 \), where a transition is inside the collection and its three attributes are outside of the collection. Recall that the parts outside of the collections occur once, and must therefore have the same value for all the transitions. When the variables or values must be shared by collection nodes, we call them shared variables (e.g. \(?\text{trigger}\) and \(?\text{effect}\)) or shared values (e.g. \(#\text{null}\)). We have introduced a keyword \(#\text{null}\) to indicate that all the guard values shall be undefined (the same interpretation as a true value).

We have included a NAC to ensure that all the substates within the composite state have the requested transitions to the external state. The matching is injective, which means that the NAC prohibits the existence of other substates than those already matched by the LHS and repeated with \( \text{id}=1 \) in the NAC. The new transition in the RHS gets the same trigger and effect values as those shared by all the replaced transitions, and it gets an undefined guard value.

The bottom part of Figure 4 illustrates how the matching algorithm works. First we non-deterministically find a match \( m^{\psi^-} \) of the rule \( m^{\psi^-} \), which is an injective morphism for the rule \( p \) (shown in the bottom left part). The injective morphism \( m^{\psi^-} \) is extended by three subgraph matches of the collection content until we reach the cMatch \( m^{\psi} \). The top right part of the Figure 4 shows the refactored model after applying the match \( m^{\psi} \) and the rule \( p^{\psi} \) on the source model.

4 Simulating a Collection Rule by a Transactional Sequence of Collection Free Rules

As an alternative to implementing the new matching algorithm from Section 3.3, this section shows how a rule with collection operators can be simulated in an existing graph transformation tool such as AGG [25]. Since the collection operator is not available, we use a transactional sequence of multiple collection free rules to simulate the intended effects of a single rule with collection operators. An early prototype of the approach was described in [11]. We only consider NAC free rules in this section.

The complicated apparatus and the set of less intuitive collection free rules show the large benefits for the transformation designer to have direct support for the collection operator. The alternative is to manually define and ensure a correct execution strategy of collection free rules, which is time consuming and error prone.

A rule \( r \) with collection operators can be represented by zero or one Init rule, one or more Iter rules and zero or one Final rule. These rules are ordered and executed as a transaction.

**Init rule.** The rule shall be applied only once as the first rule in the transaction. This rule has \( LHS = L^{\psi^-} \), which ensures that there is a match of the original rule \( r \). The RHS copies the LHS and adds all (if any) of the to-be-added non-collection
elements. It must be the first applied rule, since the other rules may connect to the added elements from this rule. **Iter rules.** Each collection operator is mapped to an Iter rule. The Iter rule shall be applied to each subgraph match of a collection. **Final rule.** The Final rule deletes all non-collection elements.

The transformation of a rule \( r \) with collection operators is defined by the pseudocode of algorithm 1. The `numCollections` method returns the number of collection operators. The `remColls` method removes all collection operators including their content. The `collContent(i)` method retrieves the content inside collection operator number \( i \). `Iter_i` is the rule for collection \( i \). `Iter_i` only applies the changes relevant to collection \( i \). By not changing any other parts, all the individual matches within collection \( i \) as well as the other collections get an equal chance to be matched.

**Algorithm 1. ToCollFree(r : CollRule)**

\[
\begin{align*}
\text{nonCollAdd} &= r.R.\text{remColls} \setminus r.I.\text{remColls} \\
\text{Init} &= \text{new Rule};\quad \text{Init}.L = r.L^{\psi^{-}};\quad \text{Init}.R = \text{Init}.L \cup \text{nonCollAdd} \\
\text{for } i &\leftarrow 1 \text{ to } r.\text{numCollections} \\
&\quad \text{do } \begin{cases} 
\quad \text{Iter}_i = \text{new Rule} \\
\quad \text{Iter}_i.L = r.L.\text{remColls} \cup \text{nonCollAdd} \cup r.L.\text{collContent}(i) \\
\quad \text{Iter}_i.R = r.L.\text{remColls} \cup \text{nonCollAdd} \cup r.R.\text{collContent}(i) \\
\quad \text{if } (r.L.\text{remColls} \setminus r.I.\text{remColls}) \not\equiv \emptyset \\
\quad \text{then } \begin{cases} 
\quad \text{Final} = \text{new Rule}; \quad \text{Final}.L = r.L.\text{remColls} \\
\quad \text{Final}.R = r.R.\text{remColls} \setminus \text{nonCollAdd}
\end{cases}
\end{cases}
\end{align*}
\]

Figure 1 showed an example of a rule with collection operators. By following algorithm 1 we get a set of rules, \{`Init`, `Iter`, `Final`\}, as shown in Figure 5. The Final rule is produced since there are non-collection elements to-be-deleted. The Iter rule replaces a path of control flows going to the innermost decision and merge nodes, with a new path of control flows only going to the outermost decision and merge nodes with a combined guard. The Iter rule replaces one path each time the rule is applied. The Final rule is applied when the Iter rule is no longer applicable.

We need to ensure that all the rules in a single transaction involve the same context regarding the non-collection elements, which we achieve by introducing an additional `id` attribute for all the elements. All the non-collection elements in the Iter and Final rules gets `id` values corresponding to the elements matched by the Init rule.

Conceptually, the collection rule LHS builds an entire match, and then applies the effect defined by the RHS. When simulating such a behavior with multiple rules in AGG, we need to be careful about possible dependencies and interactions between the Iter rules. One Iter rule may for instance add elements leading to yet another individual matching of another Iter rule, which is incorrect behavior. To avoid this problem we extend all the model elements by a boolean helper attribute named `exclude`. All `exclude` attributes are set to `false` at the start.
of the transaction, while all the collection content `exclude` attributes are set to `true` in the RHS of the Iter rules. Furthermore, the LHS of the Iter rules are extended so that they only match elements with the `exclude` attribute set to `false`. By doing so, the Iter rules can be applied in an arbitrary order.

In other tools with more control flow and transaction support like in Fujaba [10] and PROGRES [23], it may be simpler than with AGG to simulate a collection rule by collection free rules. Still, the introduction of a collection operator will greatly reduce the effort needed by the transformation designer when designing rules.

5 Examples

In this section we show three examples where the collection operator is helpful.

**Fire transition in petri nets.** A petri net model consists of places, transitions and directed arrows. The directed arrows goes from a place to a transition or from a transition to a place. A transition $T_1$ has a preset of places which is the places that have a directed edge to $T_1$, and $T_1$ has a postset of places which is the places that have a directed edge from $T_1$. At any moment a number of tokens are assigned to each place, and each token is assigned to exactly one place.

In our concrete syntax, the tokens are drawn as small, filled circles, places are drawn as larger, unfilled circles, and transitions are drawn as rectangles. An example is shown at the top left of Figure 6 with label 1, where we have a single transition consisting of two places in the preset and three places in the postset. The places in the preset have one and two tokens respectively. The places in the postset have zero, zero and one token respectively.

A transition is enabled when all the places in the preset of a transition have at least one token. The transition, within the model labeled 1 in Figure 6, is thus enabled and we can fire a transition. When firing a transition we shall remove
one token from each place in the preset and add one token to each place in the postset. The resulting model after firing the transition is shown with label 2.

With two collection operators (identified as $c_1$ and $c_2$) we can define the firing of a transition by a single rule. Collection $c_1$ expresses that we remove one token from each place in the preset, while collection $c_2$ expresses that we add one token to each place in the postset. The NAC ensures that there are no preset places without a token.

**Activity model refactoring: add fork.** UML activity models allow an activity to have multiple outgoing control flows, which are interpreted as an implicit fork. It is normally encouraged to use an explicit fork node instead, which we can automatically introduce by the leftmost rule in Figure 7. We assume that the rule editor is more flexible than typical activity model editors, by allowing a control flow without a target. The missing target allows any kind of target node type in the model match. If a target node is required in the editor, then we can use an abstract supertype from the UML metamodel representing the possible target nodes. This type will be displayed with the abstract syntax as the target node in the rule. The lower cardinality of the collection operator is 2, so that the fork node is only introduced when there is more than one outgoing control flow.
From feature models to BPMN. This example, given by the rightmost rule in Figure 7, shows a need for two collection operators in the same rule. The rule MandatoryAndOptional is one of many rules we have defined to transform from feature models [2] to Business Process Modeling Notation (BPMN) [20] (BPMN models are very close to UML 2 activity models). For this example transformation, the sibling features are assumed to represent independent tasks. The rule is simplified compared to the complete rule that works recursively when the child features themselves also are parent features.

Features are mapped to BPMN activities. Activities of child features are placed inside independent control flow branches of an activity. We use two collection operators, one for optional tasks and the other for mandatory tasks.

A parent feature with the arbitrary name ?F is mapped to an activity node with the same name. We get an internal fork-join branch (fork and join are displayed with a diamond symbol with a plus sign inside) to represent all the mandatory tasks, and an internal inclusive decision-merge branch (decision and merge are displayed with a diamond symbol with a circle inside) to represent all the optional tasks.

6 Related Work

In this section we describe related approaches, and these can be distinguished as two groups: 1) collection matching and transformation that is restricted to single nodes only, and 2) collection matching and transformation of subgraphs.

Fujaba [10] and PROGRES [23] have support for matching collections of single nodes only (set nodes in PROGRES, multi objects in Fujaba), which is a limited expressiveness compared to the collection operator that allows for collections of a fixed but arbitrarily large subgraph. Furthermore, the single node approaches are only defined for abstract syntax. To determine if single node collections are expressive enough for a particular transformation task may depend on the choice of abstract syntax representation of the involved source and target languages.

As an example, we now consider if we can use single node collections to express a rule for firing of petri nets (Figure 6). If the abstract syntax of petri net graph representation uses two different node types to represent tokens and places, then a rule to perform transition firing with single node collections will fail. This is because all tokens of the places in the transition preset will be consumed, and not only one token per place as required. This problem can be avoided by choosing a different abstract syntax where a place has an integer attribute to keep track of the number of tokens instead of having a separate node type for a token. In general it is undesirable to adjust the abstract syntax due to limitations in the rule language. By using E-graphs [8] where edges can have attributes we can get away with using single node collections for some of the paper examples, depending on the choice of abstract syntax.

The remaining approaches in this section are all capable of handling subgraph collection matching and transformation. A group operator, introduced by Balasubramanian et al. [1] and implemented in the GReaT tool, enables arbi-
Arbitrarily large subgraph matches that can be copied, moved or deleted. However, the subgraph matches can not be modified as with the collection operator.

Amalgamated rules [5] by Jaramillo et al. can simulate the collection operator. Our collection operator is more concise since we can use a single rule, while they need one subrule to capture the rule part outside of all collections, and one elementary rule for each collection operator.

Nested quantification is proposed by Rensink [22] as an extension to the GROOVE tool, which is similarly concise as our collection operator by allowing a single rule to express subgraph matches. His notation is a bit different from ours since they use exists ($\exists$) and for all ($\forall$) quantifiers to express the parts outside of a collection, and those inside a collection respectively.

Fuss and Tuttles [9] propose an extension to PROGRES called set-regions, which is quite similar to our collection operator. However the concrete notation of such set-regions within the rules is not shown. A strength compared to many other approaches is that they allow for nested set-regions. In this paper we have not allowed the collection operators to be nested, but this seems to be an appropriate extension which we plan to describe in future work.

Minas and Hoffmann [14,18] define a cloning operator which is an alternative to our collection operator. Cloned nodes and incident edges correspond to elements inside a collection operator. They support multiple elements inside the same collection operator by assigning the same cloning identifier to several cloned nodes (the incident edges of the cloned nodes implicitly belongs to the same collection).

To our best knowledge none of the other subgraph collection matching approaches have support for shared variables nor collection cardinalities beyond 0..* and 1..*. Furthermore, the other approaches focus only on applying their collection operators for the abstract syntax. The notations by Rensink [22] and as sketched by Fuss and Tuttles [9] have a nature which makes them appropriate to be introduced on the concrete syntax, which is not the case for Minas and Hoffmann [14,18]. We extend our earlier work [11] where the collection operator was restricted to activity model transformations. The improvements in this paper includes support for multiple collection operators of arbitrary cardinalities in the same rule.

7 Conclusions

In this paper we have introduced the collection operator, which makes graph transformation suitable to use on a number of model transformation cases where it would be cumbersome or impractical without. The collection operator raises the level of abstraction, which is a benefit to the transformation designer. For model transformations where the collection operator naturally applies, Section 4 shows that it is a complicated and time consuming task to manually define transformations without the collection operator.

The collection operator can be used both on the concrete syntax of the modeling language and at the abstract syntax of graphs. A straightforward implementation
strategy, described in Section 3.3, shows how we can identify matches and apply derivation steps by reusing much of the existing graph transformation apparatus.

We leave it as future work to investigate how the use of collection operators affect the theory of termination and confluence. It is also future work to decide the conditions under which graph transformation is naturally applicable at the concrete syntax level.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project grant no. 215412 (EU FP7 STREP).

References

Appendix G

Paper 6: Concrete Syntax-based Graph Transformation
Concrete Syntax-based Graph Transformation

Roy Grønmo and Birger Møller-Pedersen

October 2009

Abstract
In model-driven engineering model-to-model transformations are crucial. Thus, the widespread adoption of model-driven engineering relies heavily on the usability of model-to-model transformation languages. The leading transformation languages come from two related fields: graph transformation and model transformation. The drawback in many of these approaches is that the user needs access to and knowledge of the often complex metamodels. In this paper we present the framework of a general purpose model-to-model transformation language based upon algebraic graph transformation. The language provides rules that allow the transformation modeler to concentrate on the intuitive concrete syntaxes of the source and target modeling languages. We introduce the approach by an example transformation from feature models to business process models. The approach has been tested on a number of modeling languages and we report the major findings of these case studies.

1 Introduction
In model-driven engineering model-to-model transformations are crucial in order to maintain relationships between models and to increase the efficiency by automating as much as possible in the software engineering process. Relevant transformations include horizontal transformations between models from different viewpoints (e.g. UML sequence diagrams to UML state machines [34]), vertical transformations from an abstract model to a more detailed model (e.g. UML class model to WSDL [17]), and model refactoring (e.g. transformation of workflow graphs [10]).

There is a multitude of modeling languages, diagram types and domain-specific languages which are tailored for different purposes and for the businesses involved. This leads to an unanticipated number of combinations of modeling languages on which model-to-model transformations are needed.

The leading model-to-model transformation languages are from the fields of graph transformation (e.g. AGG [35], Fujaba [13], PROGRES [32]) and model transformation (e.g. ATL [21], Epsilon [23], QVT [28]). These fields address the generic problem by generic metamodel languages that are capable of defining virtually any modeling language. The transformation developer defines the
model-to-model transformation relative to the metamodel concepts of the source and target languages. This makes the current approaches applicable to arbitrary model-to-model transformation needs.

Although the current model-to-model transformation approaches are general, they are often cumbersome to use for the transformation developer. This is because the transformation developer in most of the existing approaches must investigate the details of the often complex metamodels in order to define a transformation [12], and because rules are not defined by using the concrete syntax.

Model instances can be represented by graphs consisting of nodes and directed edges that connect the nodes. Such graphs have a predefined generic layout, called the abstract syntax. The abstract syntax visualizes all element types with a limited set of visual symbols, such as rectangles for node instances and arrows for the edges, where the types are distinguished by a type name. In order to reduce the gap between abstract and concrete syntax, some tools (e.g., AGG [35]) allow to use special icons and tailored symbols to render elements depending on their type. This is however still quite limited compared to the full concrete syntax.

In this paper we present the framework of a general purpose model-to-model transformation approach, which we call concrete syntax-based graph transformation (CGT). CGT has the following properties:

- **Concrete syntax.** The transformation modeler can think entirely in the concrete, graphical syntax of the source and target modeling languages, i.e. no knowledge of the source or target metamodels is needed.

- **Graph transformation.** CGT rules are compiled into algebraic graph transformation rules in abstract syntax, which means that the well-established algebraic graph transformation theory [11, 29, 24, 25] and tools [35, 5, 3] can be directly applied.

- **Generic.** CGT is applicable to arbitrary source and target languages. CGT shall strive to make concrete syntax-based rules appropriate for the most common modeling languages. Abstract syntax can still be used for non-graphical modeling languages or if there are problematic concrete syntax constructs.

Our approach requires that the concrete syntax is precisely defined and not just a collection of graphical symbols. Fortunately this is the case for all the syntaxes we have considered. As an example, the syntax of UML state machines is not formally defined, but the abstract syntax (and its well-formed rules) together with the symbols for each meta class provides a definition that is precise enough for our purpose.

AToM3 [8] is an existing tool in which you can define the rules by using concrete syntax-based rules that are compiled into python code with their own matching and transformation engine. This makes it best suited for transformations where both the source and target languages have a concrete, graphical
Concrete Syntax-based Graph Transformation

syntax. For other languages, the transformation code needs to be written in python code. Our approach of using concrete syntax-based rules that are compiled into abstract syntax-based rules means that we can reuse existing matching and transformation engines, as well as switching to graphical, abstract syntax when this is better suited. Furthermore, confluence and termination analysis can be directly applied if our rules are compiled into AGG rules.

The contribution of this paper is to describe the overall architecture of our fairly comprehensive but coherent approach, so that tool developers are able to implement the approach. We only provide a brief overview, and not a new contribution, of the existing algebraic graph transformation, how to define the concrete and the abstract syntax, and how to define transformations between the concrete and the abstract syntax. All these parts are already covered in several publications and tools. Based on our case studies, we devote much space in the discussion section to cover some of the more difficult parts of using concrete syntax-based rules.

The remainder of this paper is structured as follows: Section 2 presents the main principles of traditional abstract syntax-based graph transformation; Section 3 provides an overview of our CGT approach; Section 4 applies the CGT approach on an example where we transform from feature models [4] to Business Process Modeling Notation BPMN [27]; Section 5 explains how CGT rules are compiled into traditional graph transformation rules for execution; Section 6 reports our experience with CGT based on our case studies; Section 7 presents related work; and finally Section 8 concludes our work.

2 Traditional abstract syntax-based graph transformation

In algebraic graph transformation, typed attributed graphs [20] (hereafter called graphs) are used to define metamodels. The two main buildings blocks are nodes and edges. Both nodes and edges have a predefined type attribute and contain an arbitrary number of attributes. An edge has exactly one source and one target node.

A BPMN graph model in abstract syntax is shown in the middle part of Figure 1. The graph edges are displayed as directed arrows labeled by the edge type. The graph nodes are displayed by rectangles with two compartments: the node type in the first compartment, and the attributes in the second compartment (similar to a UML class layout). The abstract syntax is automatically derived from the metamodel as opposed to a graphical concrete syntax, where the different types have special icons and visual representations that are not defined in the metamodel. BPMN graph models in concrete syntax are shown in the right part of Figure 1.

A transformation rule uses abstract syntax with a left hand side graph (LHS), a right hand side graph (RHS), and an arbitrary number of negative application condition graphs (NACs). The LHS defines a subgraph to be matched within
the graph to be transformed (hereafter called source graph). A matched LHS within the source graph is replaced by the corresponding RHS. The so-called dangling condition ensures that a rule can be applied only when there will be no dangling edges in the resulting graph. A NAC prevents application of a rule if the LHS combined with the NAC has a match. None of the NACs can have a match in order to apply a rule.

Figure 1 shows a graph transformation that introduces a fork node (displayed as a diamond with a plus-sign) when there are two sequential flows leaving the same activity in a BPMN model. The rule is defined on abstract syntax to the left, the graph in the middle is the source graph in abstract syntax, and the concrete syntax of the source model (top) and target model (bottom) are shown to the right. Identifiers, displayed with a number followed by a colon, e.g. 1:Activity, are shown in the rule when there are shared elements between the LHS and the RHS/NACs. Elements that are shared between the LHS and the RHS are preserved in the rule application. Elements where we have not displayed an identifier, occur either only in the LHS and will be deleted, or they occur only in the RHS and will be added.

3 Concrete syntax-based graph transformation

Our approach is to provide a framework similar to graph transformation, with the major improvement that rules can be defined with the concrete syntax.

Figure 2 explains the process that a transformation modeler performs in order to configure and automatically generate a concrete syntax-based rule editor. The transformation modeler provides the definitions of the source and target metamodels, i.e. the abstract syntax. For commonly used modeling languages, such metamodel definitions may be publicly available. Assuming our framework supports the available format, it is sufficient to import the definition. Other-
Concrete Syntax-based Graph Transformation

The next step is to associate a graphical representation with the abstract syntax, i.e., to define the concrete syntax. The Graphical Modeling Framework (GMF) [9] is one example of a tool where a transformation modeler can define the concrete syntax and link it to the abstract syntax. A concrete syntax defines the rendering of nodes and edges, such as a closed arrow with thin line to display a BPMN sequence which has one node and two edges in the abstract syntax. Again, for commonly used modeling languages, it may be sufficient to import the concrete syntax definition from a publicly available registry. The steps of defining source or target concrete syntaxes can be skipped, but then only abstract syntax will be available in the rule editor.

When the source and target abstract and concrete syntaxes are defined, a fully automated tool can generate a rule editor. In the rule editor, the transformation modeler can concentrate on the concrete syntax, while the actual implementation behind the scenes uses the abstract syntax and a traditional graph transformation tool to do the transformation. A proof-of-concept tool, specialized for activity models [15], demonstrates this principle.

The steps of linking abstract syntax to concrete syntax will be additional manual work compared to using an abstract syntax based rule editor like in AGG. This additional work is only needed once each time we need to use a new modeling language as either the source or target.

We illustrate the approach by investigating an example transformation from feature models (FM) [4] to Business Process Modeling Notation (BPMN) [27].

4 Example: from FM to BPMN

Feature models are commonly used in software product line engineering [30] to capture the variations of a product. Some features are optional, others are mandatory and the features are organized in a tree structure with parent-child relationships: optional, mandatory, or, and alternative.

Montero et al. [26] describe a transformation from feature models into the basic structure of a business process model represented in BPMN. It is assumed that the sibling features represent independent tasks, although this is not the...
An example transformation is given in Figure 3. The source model is a feature model of an airline travel agency booking process. A booking process (Booking) will either cancel the booking (Cancel) or proceed with booking (Book). The Cancel feature has no child feature, while the Book feature has two mandatory child features: Book Hotel and Book Flight.

The figure shows snapshots of the current model at four different stages in the transformation, where \( \Rightarrow \) denotes the application of rule \( r \) from one stage to the next. The source model (labeled 1) is a pure feature model. The intermediate models (labeled 2 and 3) are mixtures of feature model, BPMN model, and the ready edge (explained in section 4.3). The transformation has ended in the fourth stage, and we have reached the target model which is a pure BPMN model.

In the BPMN model, the activity Booking is the overall process, which starts with its inner start node (a thin lined circle). Directed edges represent sequential control flow. Gateways are displayed as diamonds with predefined symbols inside. A gateway with a + sign represents a fork if it has multiple outgoing edges, and a join if it has multiple incoming edges. A gateway with an o/x sign (o=or, x=xor) represents a decision/merge node if it has multiple outgoing/incoming edges. An activity ends its control flow in an end node (a thick lined circle).

The following subsections go through the steps needed to configure and automatically generate a concrete syntax-based rule editor.
4.1 Import/define abstract and concrete syntax

We need to define the abstract and concrete syntaxes for both the source and target modeling languages. The transformation modeler first checks with publicly available registries, and we assume that the modeler finds both an abstract and a concrete syntax definition (and in formats accepted by the tool) of the target BPMN modeling language (wiki.eclipse.org/index.php/GMF_Tutorial_BPMN). Furthermore it is assumed that an abstract syntax definition of the source FM modeling language is found (gsd.uwaterloo.ca/category/projects/ecorefmp/), while a concrete syntax definition for FM is not found in existing registries. This means that three out of four definitions are simply imported from existing repositories. Only a concrete syntax definition for the FM language needs to be specified manually by the transformation modeler.

The table in Figure 4 provides an overview of how the different parent-child types from the feature model are mapped from concrete to abstract syntax.

<table>
<thead>
<tr>
<th>FM construct</th>
<th>Concrete syntax</th>
<th>Abstract syntax</th>
<th>Metamodel</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternative</td>
<td><img src="image" alt="alternative" /></td>
<td><img src="image" alt="alternative" /></td>
<td>feature parent 1 child 2..* xor</td>
</tr>
<tr>
<td>or</td>
<td><img src="image" alt="or" /></td>
<td><img src="image" alt="or" /></td>
<td>feature parent 1 child 2..* or</td>
</tr>
<tr>
<td>mandatory</td>
<td><img src="image" alt="mandatory" /></td>
<td><img src="image" alt="mandatory" /></td>
<td>feature parent 1 child 1 mand</td>
</tr>
<tr>
<td>optional</td>
<td><img src="image" alt="optional" /></td>
<td><img src="image" alt="optional" /></td>
<td>feature parent 1 child 1 opt</td>
</tr>
</tbody>
</table>

Figure 4: Mapping Concrete Syntax to Abstract Syntax for Feature Modeling

4.2 Generate the CGT rule language and the CGT rule editor

Now that the abstract and concrete syntaxes are defined, the CGT rule language and associated rule editor can be automatically generated.

The metamodel of the rule language connects the abstract syntax elements to concrete syntax elements, and the rule language is the union of the source and target language with a few extension, plus three constructs described below. The LHS and the NACs use the same rule language, while the RHS rule language only have a small difference for the attribute expressions. A LHS attribute can have a match expression, while a RHS attribute has an assignment expression.
Figure 5 shows an extract of the LHS rule metamodel for BPMN. The white rectangles are abstract syntax types that are instances of the graph metamodel, and the black rectangles are the concrete syntax types. We have an automatic procedure to extend the concrete syntax elements with rule specific types (shown as grey rectangles): 1) The concrete syntax elements that have been assigned to at least one node or edge, will get an IDTextEventField where the rule designer can attach identifiers, and 2) The concrete syntax text fields of attributes are extended to AttrMatchExprTextField for the LHS, and to AttrAssignExprField for the RHS.

The extension for attributes works only for attributes that uses text fields to display their value, which applies to many of the typical attributes in modeling languages. For other attribute types we do not see a need to extend the concrete syntax in the rule language, e.g. a UML class name is displayed in italics if and only if the isAbstract attribute of the class is true.

The AttrMatchExprTextField and AttrAssignExprField may be ordinary attribute values or include expressions with variables, similar to the attribute values in a graph transformation tool like AGG [35].

A CGT rule editor can have a layout similar to traditional graph transformation tools. The editor allows to define and execute CGT rules. All the concrete syntax elements of the source and target languages are available for the user to select and place within the LHS/RHS/NACs of a rule in a similar manner as in a traditional modeling tool. As opposed to a traditional modeling tool, the CGT rule editor must allow the LHS/RHS/NACs to be model extracts, e.g. elements with missing mandatory relations and properties. Furthermore, the CGT rule editor must allow the transformation modeler to define shared identifiers of elements between the LHS and the RHS/NACs of a rule.

Figure 6 shows a proposed layout for the rule editor (currently only illustrated and not implemented). The leftmost panel (called Rules) is a tree view of the current project and its set of rules. The middle panel shows the currently viewed rule with its LHS and RHS. Possible NACs will also be shown in this middle panel. To the right we have the Palette panel which is tailored specifically for the source and target modeling languages. It has all the source
constructs available on the left side in blue, and all the target constructs available on the right side in red. In the special case where the source and target modeling languages are the same, the left and right parts of the Palette are merged and all the constructs are visualized with the same color.

The bottom part of the Palette panel contains some generic constructs in black that are independent of the source and target languages. These constructs are part of any rule editor. The collection operator \([14]\) is a high-level operator to denote collections of similar subgraphs (more details are provided with the rules below). We also have the generic node and edge which are identical to the constructs available when defining the rules on abstract syntax in traditional graph transformation. The generic nodes and edges can be used for metalelements and helpers that take part in a transformation. Generic edges can have any generic node or any concrete syntax element as its source or target.

The bottom part of the rule editor allows to switch between concrete and abstract syntax. If the rule designer has not defined the concrete syntax for the source or the target language, then only abstract syntax will be available. The Palette panel will then show all the abstract syntax node and edge types for the source or target. When both the source and target language uses the abstract syntax we are reduced to abstract syntax-based graph transformation.

All the Palette constructs can be used in the LHS, the RHS and the NACs. It is not the case that only source constructs go in the LHS and target constructs go in the RHS. This becomes clear as we investigate the actual rules below.

With the generated rule editor, the transformation modeler proceeds with defining the necessary rules.
4.3 Define the concrete syntax-based rules

We have defined eight rules to do the desired transformation from feature models to BPMN. Figure 7 shows the four rules that are needed to do the example transformation that was previously shown in Figure 3. The shared identifiers between common elements of the LHS and the RHS/NACs are displayed next to the elements with an underlined text field (e.g. \texttt{id=1}). When it is hard to see which element an identifier belongs to, the modeler can open a property view of an element to see its identifier.

Our rules allow the user to define variables in the attribute fields of an element. The variables are prefixed by a question mark, e.g. \texttt{?F} in the feature name fields. The \texttt{?F} matches any feature name and can be repeated in the RHS part of a rule to indicate an assignment of the matched variable from the LHS part of the rule. An example of such usage is in the \texttt{MarkRoot} rule where we add a new activity which is assigned the same name (\texttt{?F}) as the LHS matched feature name (\texttt{?F}).

The transformation is performed recursively from the root feature to its children. The \texttt{MarkRoot} rule finds the single root node of the feature model. The NACs \texttt{NAC-2}, \texttt{NAC-3}, \texttt{NAC-4} and \texttt{NAC-5} ensure that the root feature is not a child in any parent-child relationship. The root is attached to a newly created target activity by a \texttt{ready} edge (a generic edge) which indicates that this feature is ready to be transformed. The NAC \texttt{NAC-1} ensures that the \texttt{MarkRoot} rule is only applied once, and that the root features are not multiply marked as ready. The idea is to handle the root first, then all the roots children will act as roots and these children all get \texttt{ready} relations, since all parents are handled.

The \texttt{Empty} rule ends the recursive process by deleting the feature and the \texttt{ready} edge. It can only be applied to features with a single \texttt{ready} edge attached and having no other relations. The dangling condition ensures that the feature

![Figure 7: Transformation from feature model to BPMN using concrete syntax](image)
Concrete Syntax-based Graph Transformation

has no other relations.

The remaining rules apply to a root feature with a ready edge where the feature is a parent in at least one parent-child construct. There is one rule for each combination of parent-child relations this parent feature can have. The child relations all lead to a structure inside the activity corresponding to the parent feature. All these remaining rules delete a root feature and a ready edge. The dangling condition is helpful once again to prevent the root feature from having other relations.

Five of the rules (including Mandatory and Alternative shown in Figure 7) take advantage of the collection operator, which is visualized with a dashed frame. A collection operator contains a subgraph of the rule that can occur multiple times in a single match, but restricted by its cardinality. Potential matches shall be extended to the largest number of subgraph matches (limited by the upper cardinality) for each collection operator.

The Mandatory rule applies in cases where the ready marked feature ?F is a parent in two or more mandatory constructs (ensured by having 2 as the lower cardinality). Then we produce a fork/join construct in the BPMN model. The Alternative rule is mapped to an exclusive decision/merge in BPMN, which means that exactly one branch will be fired.

5 Mapping from concrete to abstract syntax

To improve the usability for the graph transformation designer, we have defined the transformation rules upon concrete syntax. The transformation designer can think entirely in the concrete syntax, while the matching and transformation is carried out in the abstract syntax. This is illustrated in the figure below:

```
models: concrete syntax  |  graphs: abstract syntax
SC  \rightarrow  TC  |  SC  \rightarrow  TA
\downarrow  c2a  |  \downarrow  a2c
S  \rightarrow  T
```

The mapping from concrete to abstract syntax (c2a), and the opposite direction (a2c), is defined by the transformation modeler as described in Section 4.1. We can link concrete syntax-based graph transformation to abstract (and traditional) syntax-based graph transformation in a systematic way: 1) map the concrete syntaxes of source model and rules into abstract syntax graphs by c2a, 2) apply the abstract syntax graph transformation rules on the source graph, and 3) map the resulting abstract syntax graph to a concrete model by a2c.

Since the rule language extends the modeling languages, the mapping from concrete to abstract syntax of the rules (c2a⁺) extends the mapping from concrete to abstract syntax for the models (a2c). The extensions of the rule language contain id’s, match expressions and assignment expressions as illustrated by the grey rectangles in Figure 5. These are chosen to be very similar to what you can express on abstract syntax rules in the AGG tool, so the mapping is
easy to define. The extended part of the mapping, i.e. $c2a$, is defined once for all source and target languages.

With a large number of modeling languages, including feature models and BPMN illustrated in this paper, the same mapping ($c2a$) is reasonable to use for both the rules and the source model. For some languages (e.g. UML sequence diagrams) we need to deviate a bit from using the same mapping.

Figure 8 shows how our previous CGT rule Mandatory is mapped to a traditional abstract syntax-based rule. When the mapping between concrete and abstract syntax is given, as illustrated in section 4.1, the mapping from a CGT rule to a GT rule is straightforward.

The different models of the rule (LHS/NACs/RHS) are mapped individually by applying the c2a mapping rules. The identifiers and attributes are preserved, although they are presented differently. A collection operator shall include all incident edges to nodes inside the collection (e.g. the parent edge in the LHS) as defined by [14].

Just as a Java programmer should not have to worry about the compiled code, the transformation modeler should not have to worry about the compiled abstract syntax-based rules. This means that all the feedback to the transformation modeler should relate to the concrete syntax-based rules, including error reports, debugging, critical pair, consistency and termination analysis.

Pragmatically, however, we need to accept that early implementations of our approach only reports errors, critical pairs and termination problems in the abstract syntax. It is still better to have for instance a critical pair analysis available on the abstract syntax-level than no such analysis available.

6 Experiences and Discussion

We have experimented with our proposed approach in a number of case studies: UML activity diagram aspects [15], UML activity diagram refactoring [16, 15, 14], UML state machine refactoring [14], UML component diagram aspects, UML sequence diagram aspects [19, 18], transformation from sequence diagrams to state machines, and petri nets [14].
Our basic principle of using the same mapping from concrete to abstract syntax for the models and the rules is sufficient for all the modeling languages in our case studies, except sequence diagrams. The ordering of events on a lifeline needs special treatment.

We also propose a special treatment for languages with containment relations (e.g., UML activity diagrams/state machines), to make the default CGT rule language more user-friendly. Both ordered relations and containment relations can be described in a sufficiently precise metamodel, which means that our adjustments can be incorporated into the automatic rule generator.

### 6.1 Ordered relation

The problem with ordered relations occur when an element has a significant order among its connected elements, e.g., the order of the events on a sequence diagram lifeline:

We illustrate the problem by investigating transformations where the source and target models are both sequence diagrams [18, 19], and where the same mapping from concrete to abstract syntax is used for both the rules and the source model. Figure 9 shows a simple example where we have the concrete syntax of one rule and one source model at the top, and our choice of corresponding abstract syntax below. The transformation rule defines an intended replacement of $x$ messages from lifeline $L_1$ to lifeline $L_2$, by $y$ messages in the same direction between the same two lifelines.

The intention is that the $x$ message in the base model should be replaced by the $y$ message. However, with the default configuration of CGT there are no matches in the source model, since the events of the $x$ message, $send_x$ and $receive_x$, are not in the correct order.
Grønmo and Møller-Pedersen

receive x, must be the first events on their respective lifelines in the source model to be a match of the LHS. The intention of getting matches, even though the send and receive events are not necessarily the first events on the respective lifelines, is in accordance with all the proposals we have seen of so-called sequence diagram aspects (e.g. [37, 22]).

CGT can be extended by special treatment of all the ordered relations in the metamodel. The abstract syntax for the models needs to be designed in a specific way, and we need to use a different mapping for the rules, than for the models, to ensure that the order of the elements have relative positions. We leave the details of this to future work.

The sequential control flow constituting a path in a BPMN model is also a kind of order, but such an ordering does not impose a problem for CGT. This is because from the viewpoint of all single elements, such as the activity element, there is no order on the incident edges.

6.2 Containment relation

For both activity diagrams and state machines, there is an outermost model element that contains the other model elements. Model elements may in turn contain other model elements.

The container relation is important when designing the rules, as explained by Biermann et al. [6]. When elements are to be deleted by a rule, the match must include the container relation. Otherwise, application of the rule is prevented by the dangling condition. This is illustrated in Figure 10. The source model is shown at the top leftmost part as concrete syntax, and an extract of the source model as a graph/abstract syntax is shown as the top rightmost part. A rule shall express that source models having equivalent activities in two paths can be combined into a single path. Hence, one activity will be deleted by the rule. The rule in the middle fails since a container relation would become a dangling edge. A fixed version of the rule is shown in the bottom where the container is explicitly matched as part of the rule.

We also need to be careful when adding elements. When elements are to be added by a rule, the match must include the container relation of these new elements. This means that we include a surrounding container just as in the bottom rule of Figure 10. How often we need to include the surrounding container depends also on the choice of abstract syntax. A control flow between two activities may implicitly get the same container as its source and target activities. Similarly a transition between two state machines may implicitly get the same container as the closest container to the root container among its source and target states.

As a convention, our rules automatically inserts a surrounding container which is shared between the LHS and the NACs/RHS. This makes the rules less overloaded and they can be designed with less effort. This is not dramatic for a single rule, but it seems to be a frequent simplification that applies to a majority of the rules. Our implicit container convention only applies to mandatory containers, i.e. when all the elements implicitly or explicitly must have a
Concrete Syntax-based Graph Transformation

Figure 10: Containment relation. Top: Source model, Middle: Rule failing, Bottom: Rule succeeding

In some rules it can be useful to match the root container or to match multiple independent structures at different nested levels. For these rules the tool must allow to turn off the automatic surrounding container.

6.3 Discussion of concrete vs. abstract syntax-based rules

Traditional abstract syntax-based rules have the advantages that they more easily apply to a larger class of modeling languages, and that the initial configuration requires less effort compared to CGT. This is because traditional graph transformation rules can simply ignore the concrete syntax.

On the other hand, CGT allows the transformation modeler to work with rule editors using the more intuitive, familiar and optimized syntax for user comprehension. Furthermore, in CGT the transformation modeler does not need any knowledge about how the metamodels are defined.

In order for the CGT approach to succeed, the following two requirements must be fulfilled: 1) a concrete syntax model instance can only be mapped to one abstract syntax graph instance, and 2) all the metamodel properties of the source and target models must be editable in the CGT rule editor, directly as graphical elements or within additional property views. The first requirement makes a CGT rule unambiguous, and this must be ensured in the mapping from concrete to abstract syntax (Baar [1] formalizes how to check this requirement). The second requirement makes a CGT rule as expressive as a traditional graph
transformation rule, and this is ensured if the mapping from abstract to concrete syntax is sufficiently comprehensive and when the generation of the CGT rule editor provides property views for non-visualized elements.

When multiple abstract syntax elements are represented by concrete syntax elements that are overlapping, very close to each other or even merged into combined concrete syntax elements, then there may be difficulties to use our approach without special treatment. This challenge occurs for an alt operator and its operands, and for a UML state machine and its regions. Such constructs can be difficult to match and transform, and in future work we plan to introduce special treatment for such constructs.

In traditional graph transformation rules on abstract syntax, the LHS, the NACs and the RHS must be proper graphs, where all edges must include both the source and target. This is quite similar to UML editors where a control flow arrow in activity models needs both a source and target in order to be drawn. It is preferable that the CGT rule editor allows more flexibility. This can only be achieved by representing a control flow as a node (and two edges indicating source and target), instead of an edge in the abstract syntax. This is illustrated by the rule below:

The rule above introduces an explicit fork node when there is more than one outgoing control flow from the same activity. Notice that in the concrete syntax, the outgoing control flow edges have a missing target. The missing target acts as a wildcard for all the possible kinds of target node (activity, fork, choice etc.). Otherwise we would have to specify a number of rules for each possible target node. To achieve such flexibility in general, we advice to represent every concrete syntax element as at least one node element in the abstract syntax. In the example, an alternative could be to use node type inheritance [7] and an abstract supertype node in the rule. However, we discourage this, since it may confuse the user if the rules mix concrete and abstract syntax to represent elements for the same modeling language.

7 Related Work

Tools like GenGed [3] and Tiger [5] have used rules defined completely on the concrete syntax. The scope of these approaches has been limited to transformations where the source and target languages are the same, and mostly to generate modeling language editors. Our approach generalizes the principles described in GenGed and Tiger to be applicable also in the context of arbitrary source and target languages. Like GenGed and Tiger, our approach is based on
algebraic graph transformation rules. We rely on the usage of a modeling lan-
guage editor like GenGed, Tiger and GEF/GMF to realize part of the necessary
infrastructure.

Other work on using concrete syntax in model transformation specifications
is by using concrete transformation examples to generate an initial model trans-
formation, e.g. Varró and Balogh in [36] and Strommer and Wimmer in [33].
Their initial generated model transformation needs to be manually refined and
generalized in the generated transformation code/rules, while our transforma-
tion rules on concrete syntax constitute complete and generalized transforma-
tions.

Schmidt [31] allow to define transformation rules involving any UML dia-
gram as the source or target language. The rules use a single combined diagram
to represent the LHS and RHS in our approach. They lack some of the ex-
pressiveness of traditional model and graph transformation languages, that we
inherit from our reuse of the existing graph transformation apparatus. Proper-
ties that are not visualized cannot be a part of their transformation rules, while
we include property views that are not part of the diagram in the complete
transformation specification.

The matching in our concrete syntax-based graph transformation (and in
traditional graph transformation) is syntactics-based as opposed to semantics-
based matching techniques [19]. A semantics-based matching takes the seman-
tics of the source language into account so that syntactically different models,
but semantically equivalent, are matched. While we have only sketched how to
associate concrete syntax to the abstract syntax elements, this is detailed and
formalized by Baar [1].

The MATA tool [37] use concrete syntax in their transformation rules. The
rules use a single combined diagram to represent the LHS, NAGs and RHS
in our approach. These rules are then mapped to abstract syntax rules in
AGG, as within our approach. As opposed to our approach, MATA is limited
to model transformations where the source and target languages are the same.
Our approach prescribes a generic way to generate the rule language, while their
rules need tailoring for each particular modeling language. MATA so far covers
UML class diagrams, sequence diagrams and state machines.

Baar and Whittle [2] show how to express concrete syntax-based rules that
are equally expressive as QVT graphical rules [28]. A rule has a LHS and a
RHS, but uses OCL expressions in a when-clause instead of graphical NACs. It
is not trivial to see how they could express our example transformation from
feature models to BMPN without support for our generic nodes/edges and the
collection operator. Their refactoring rules for UML class diagrams seem to be
expressible in our approach where most of the complexity lies in defining NACs
that correspond to their OCL when-clauses.

In previous work we have investigated several case studies which all can be
seen as hardcoded configurations of our approach [14, 15, 16]. These case studies
show that the approach is feasible in many contexts. CGT is a generalization
of the previous work that shows how to configure and automatically generate a
model-to-model transformation tool.
8 Conclusions and Future Work

The transformation modeler can largely benefit from defining transformation rules upon the concrete syntax. As opposed to traditional model and graph transformation, the transformation modeler does not need any knowledge about the often complex metamodels of the source and target languages.

While the transformations are defined at the modeling level, we reuse the well-established theory as well as the matching and transformation apparatus from graph transformation. Our approach has been manually tested, with a successful result, for a number of commonly used modeling languages. The next step is to implement full tool support for the approach.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project grant no. 215412 (EU FP7 STREP).

References

Concrete Syntax-based Graph Transformation


Appendix H

Paper 7: Confluence of Aspects for Sequence Diagrams
Confluence of Aspects for Sequence Diagrams

Roy Grønmo, Ragnhild Kobro Runde and Birger Møller-Pedersen

October 2009

Abstract

The last decade has seen several aspect language proposals for UML sequence diagrams. Aspects allow the modeler to define crosscutting concerns/aspects of sequence diagrams and have these woven with the sequence diagrams of a base model. In a real-world scenario there may be multiple aspects applicable to the same base model. This raises the need to analyse the set of aspects in order to identify possible dependencies and conflicts between applications of aspects. In this paper we establish a confluence theory for sequence diagram-based aspects with respect to the expressiveness of the language in which aspects are specified. We show that high expressiveness leads to undecidability of confluence, while less expressiveness gives decidability by an extended version of a traditional critical pair analysis from term rewriting and graph transformation.

1 Introduction

There have been a number of aspect language proposals for UML 2 sequence diagrams [2, 3, 5, 8, 18, 20, 14]. In all these aspect languages, an aspect is expressed based on the same concrete syntax as sequence diagrams. In this paper, we use the term aspect diagram to denote a sequence diagram-based aspect. Aspect diagrams at the model level define cross-cutting effects on the base model.

Some of the proposals pursue a model weaving approach [5, 8, 18, 20], while others intend to postpone the weaving to the program level [2, 3, 14]. When dealing with multiple aspects, there is in both weave alternatives a need to analyse if there are dependencies and conflicts between the aspects.

A set of terminating aspects that are confluent will always yield the same result when applied non-deterministically on the same initial model, i.e. a confluent set of aspects have no dependencies or conflicts between the aspects. Non-confluent aspects often means that it is necessary to specify an explicit weave order, redesign some of the aspects, or exclude one or more aspects.

There already exists a well-established theoretical foundation on confluence for graph transformation systems (GTS) [6, 12, 16], and confluence for term rewrite systems [11, 7]. The concrete syntax of sequence diagrams and aspects defined upon these are, however, quite different from graphs and GTS rules. For a node in a graph, there is no order among its incoming and outgoing edges. In sequence diagrams, on the other hand, the events are partially ordered. The partial order also makes sequence diagrams
different from term rewrite systems, where the elements in a term are totally ordered. This means that the GTS and term rewrite systems confluence theory cannot be directly applied. Aspect diagram extensions to sequence diagrams and new sorts of matching and weaving techniques, such as semantics-based weaving [5, 9], further complicates the relation to the existing confluence theory. In this paper we establish a specialized confluence theory for sequence diagram-based aspects.

Some of the proposed sequence diagram aspect languages have limitations. Firstly, in some proposals only single events can be used as a match condition, while we want to allow sequences of events to be a match condition. Secondly, some proposals provide no precise definition of a match. Thirdly, most of the proposals are syntactic-based. Syntactically different, but semantically equivalent structures do not result in a match for a syntactic-based aspect language. A semantics-based language defines the match in relation to the semantics of the language.

Fortunately, there are two sequence diagram aspect languages that overcome the limitations described above [5, 9]. We use one of these two aspect languages, which is our own tool-supported aspect language, previously described in [4, 5]. We investigate different levels of expressive power in our aspect language. For one level of high expressiveness, we prove that the confluence of aspects is undecidable. For another level of less expressiveness, we prove that an extended version of traditional critical pairs can be used to decide confluence for a set of aspects.

In addition to the confluence theory, this paper proves that our match and weave definitions guarantee valid woven sequence diagrams. To our best knowledge, this property has not been proven in any other work on sequence diagram aspects.

The paper is organized as follows; Section 2 provides the preliminaries regarding sequence diagrams; Section 3 contains mathematical preliminaries; Section 4 presents our aspect diagrams with the match and weave definitions; Section 5 defines independence for aspect derivations; Section 6 proves that confluence is undecidable for one class of aspect diagrams; Section 7 proves that confluence is decidable for another class of aspect diagrams; Section 8 presents related work; Section 9 briefly describe some of the potential future work; and finally section 10 provides the conclusions.

2 Sequence diagrams

Figure 1a shows a sequence diagram with two lifelines L1 and L2, and two messages with the signals a and b. A lifeline, visualized with a rectangle and a dashed line below, represents an interacting entity on which events take place in an order from top to bottom on the dashed line.

Each message is represented by two events, a send event (!) and a receive event (?). A unique identifier (not shown in the diagram) is assigned to each message, and the identifier is shared between the send and receive events of the message. Our example diagram has four events, !a and !b on lifeline L1, and ?a and ?b on lifeline L2.

Sequence diagrams impose a partial order of events given by: 1) the send event must come before the receive event of the same message (this is referred to as the message invariant), and 2) all events are ordered from top to bottom on each lifeline. An intuitive idea behind this partial order is that messages are sent asynchronously and
that they may happen in any order on different lifelines, but sequentially on the same lifeline. Figure 1b shows the four partial order requirements of the sequence diagram.

UML [15] defines the semantics of a sequence diagram by using traces that represent possible execution runs, where a trace is ‘a sequence of event occurrences’. More precisely, the semantics of a sequence diagram can be described as a set of positive traces and a set of negative traces. Positive traces define valid behavior and negative traces define invalid behavior, while all other traces are defined as inconclusive. In this paper we concentrate only on positive traces, and let \[ d \] denote the positive traces, i.e. the semantics, of the sequence diagram \( d \). The set of (positive) traces of a sequence diagram corresponds to each valid permutation of events that satisfy the partial order requirements. As shown in Figure 1d, we get two traces in our example.

The partial order relation and its transitive closure is important for some of the proofs in this paper. We let \( _{dpo} \in \text{Diagram} \rightarrow \text{Set}(\text{Event, Event}) \) denote an asymmetric relation for a given diagram, where each element pair in the relation has a direct partial order from the first event to the second event. There is a direct partial order relation between two immediate neighbor events on a lifeline (where the topmost event is the first event in the relation), and between the send and receive events of the same message (where the send event is the first event in the relation). The partial order relation, \( _{po} \in \text{Diagram} \rightarrow \text{Set}(\text{Event, Event}) \), is the transitive closure of the \( dpo \) relation. The arrows in Figure 1b shows the \( dpo \) relations, while the arrows in Figure 1c shows the \( po \) relations. If it is not clear from the context, the \( dpo \) and \( po \) relations may be prefixed with the name of the sequence diagram that they relate to.

If the partial order relation contains a cycle, then there exists no traces, and according to the UML specification [15] this means that the original sequence diagram is an invalid sequence diagram. All messages in a sequence diagram must be drawn horizontally or downwards, which prevents drawing a diagram without any traces. We only consider valid sequence diagrams.

In this paper we only cover basic sequence diagrams such as the one shown in Figure 1a. With UML 2, sequence diagrams were enhanced with a set of control flow-based operators, such as \( alt \) for alternatives, \( loop \) for loops, and \( par \) for parallel behavior. In previous work [5] we have explained how a sequence diagram using such operators can be broken down into a set of sequence diagrams without control flow-based operators. This enables us to ignore the control flow-based operators in this paper, while the results still apply to more general sequence diagrams.
3 Mathematical preliminaries

This section defines some helper functions and useful notations that are used throughout the paper:

- **Event**
  - denotes the set of all finite event sequences.

- A substring is a continuous subsequence of events, formally defined by:
  
  \( \text{substr}(t_1, t_2) \overset{\text{def}}{=} \exists h_1, h_2 \in \text{Event}^*: t_2 = h_1 \sim t_1 \sim h_2 \)

  where \( \sim \) is an operator concatenating two (finite) event sequences.

- \( \text{ev}_- \in \text{Diagram} \rightarrow \text{Set(Event)} \) returns the set of events in a given diagram

- \( \text{msg}_- \in \text{Diagram} \rightarrow \text{Set(Message)} \) returns the set of messages in a given diagram

- \( t \upharpoonright l \) is the trace \( t \) projected onto the lifeline \( l \), i.e. we remove all trace events that does not occur on the lifeline \( l \)

- \( d[l] \) is the top-down sequence of events on the lifeline \( l \) in the sequence diagram \( d \).

- \( \text{first}(d[l]) / \text{last}(d[l]) \) is the first and last event on the lifeline \( l \) in diagram \( d \) (undefined if \( l \) does not have any events in \( d \))

- \( \text{before}(e, d) / \text{after}(e, d) \) are functions returning the sequence of events before and after the event \( e \) (on its lifeline) in the diagram \( d \).

- \( \mathcal{L} \) is the set of all lifelines

4 Aspect diagrams

In our approach, the base model is a set of initial sequence diagrams. An aspect consists of exactly one pointcut diagram, exactly one advice diagram, and a (possibly empty) set of negative pointcut diagrams. These diagrams are all basic and valid sequence diagrams extended with symbolic messages and an arbitrary events symbol. An aspect is applied to one diagram within the base model at a time. We refer to this as the base diagram. For an aspect \( A \), we refer to its pointcut by \( A.pc \) and its advice by \( A.a \).

An aspect is similar to a GTS rule, where the pointcut diagram (corresponds to LHS in GTS) defines a pattern for which we are looking for matches in the base diagram. The advice diagram (corresponds to RHS in GTS) defines a replacement of the matches within the base diagram. This implies that messages present only in the pointcut and not in the advice, will be deleted, while messages present only in the advice and not in the pointcut, will be added. If any of the negative pointcuts (correspond to negative application conditions in GTS) have a match, then a potential match of the pointcut is prevented. Figure 2 shows an aspect which matches the message \( a \) directly followed by
message b. The message a is preserved, message b is deleted and message c is added. The right part of the figure shows the woven diagram when the aspect is applied to the base diagram in the middle part of the figure.

An Aspect that preserves all the pointcut messages, is called a \textit{plain additive aspect}. A non-plain additive aspect is called \textit{deletion aspect}. Figure 3 shows two plain additive aspects with the same pointcut, but different advice. We allow plain additive aspects, but we assume that care is taken to ensure that the weaving process terminates. The first aspect (labeled 1) guarantees termination if this is the only aspect to be applied. The second aspect (labeled 2) includes a match of the pointcut in the advice. To obtain termination we need to define that this aspect is applied only once to the same match. Woven matches are marked (here displayed by a superscript number) to exclude these messages from future matches of the same aspect as the number indicates. The match marking will not be part of the woven diagram.

In general our pointcut diagrams can use symbolic message symbols, such as a mix of hardcoded letters and wildcards (e.g. * to denote an arbitrary sequence of letters). For simplicity, our formalism only covers fixed message symbols in this paper. The negative pointcuts of the undecidability proof, is the only place in this paper where we use symbolic message symbols.

In sections 4.1 and 4.2, we consider first matching and then weaving of aspects consisting of one pointcut and one advice diagram only. Negative pointcut diagrams are introduced in Section 4.3, while the arbitrary events symbol is explained in Section 4.4.
4.1 Matching

We now define precisely how the matching works. The definitions use an injective mapping function, \( \phi : \text{Message} \to \text{Message} \), which maps each pointcut message to a base message with the same signal and the same lifelines as sender and receiver. Implicitly, \( \phi \) also defines a mapping from pointcut events to base events where the signal kind (send or receive) is preserved, and from pointcut traces to base traces. We will therefore overload \( \phi \) to take both messages and events as parameter. Similarly, \( \phi \) is overloaded to take a set of messages or events as parameter.

Figure 4 shows one correct and two incorrect \( \phi \) mappings for the given pointcut and base diagrams. Ids are shown explicitly, by the id number followed by a colon, as a prefix to the message signal. The second and third \( \phi \) mappings are incorrect since there is a mismatch between the sender and receiver lifelines.

\[
\begin{align*}
\phi &= \{4 \to 2\} \\
\phi &= \{4 \to 1\} - \text{wrong sender and receiver!} \\
\phi &= \{4 \to 3\} - \text{wrong sender and receiver!}
\end{align*}
\]

Figure 4: One correct and two incorrect \( \phi \) mappings

**Definition 1** (Trace-based match) The mapping \( \phi \) defines a trace-based match between a pointcut trace \( t_{pc} \) and a base trace \( t_b \) if and only if the pointcut trace is a continuous subtrace of the base trace (where the message of each event in the pointcut trace is mapped by \( \phi \)). Formally:

\[
t_{\text{match}}(\phi(t_{pc}), t_b) \overset{\text{def}}{=} \text{substr}(\phi(t_{pc}), t_b)
\]

**Definition 2** (Semantics-based match) The mapping \( \phi \) defines a semantics-based match between a pointcut diagram \( d_{pc} \) and a base diagram \( d_b \) if and only if there exists a trace-based match between one of the pointcut traces and one of the base traces. Formally:

\[
\exists t_{pc} \in \parallel d_{pc} \parallel, t_b \in \parallel d_b \parallel : t_{\text{match}}(\phi(t_{pc}), t_b)
\]

The choice of match definition is further backed by the following lemma.

**Lemma 1** (All pointcut traces are matched) If the mapping \( \phi \) defines a semantics-based match between a pointcut diagram \( d_{pc} \) and a base diagram \( d_b \), then all traces in the pointcut diagram have a matching trace in the base diagram. Formally:

\[
\forall t'_{pc} \in \parallel d_{pc} \parallel : \exists t'_b \in \parallel d_b \parallel : t_{\text{match}}(\phi(t'_{pc}), t'_b)
\]
Proof: For a simpler presentation of this proof, we assume that \( \phi \) is the identity mapping and we do not refer to \( \phi \) in the proof. However, the proof is easily generalized to other \( \phi \)s.

According to definitions 1 and 2, \( \text{smatch}_\phi(d_{pc}, d_b) \) (the left hand-side of the implication) means that there exists a base trace \( t_b \) with a pointcut trace \( t_{pc} \) as a continuous subtrace, i.e. \( t_b = h_1 \sim t_{pc} \sim h_2 \) for some event sequences \( h_1 \) and \( h_2 \).

The pointcut diagram does not contain any control flow-based operators, and any pointcut trace \( t'_{pc} \) will be a permutation of the events in \( t_{pc} \). The trace \( t'_b = h_1 \sim t'_{pc} \sim h_2 \) will be a valid base trace having the pointcut trace as a continuous subtrace. The only difference between \( t'_b \) and \( t_b \) is the relative ordering of the pointcut events, meaning that \( t'_b \) obeys all base partial orders between two events where one or both is not in the set of pointcut events.

It remains to prove that for any two pointcut events, there is a partial order between them in the base diagram only if the same partial order exists also in the pointcut diagram. For direct partial orders, this follows from lemma 12 in appendix. Lemma 12 is sufficient, as all other partial orders in the transitive closure will be implied by the direct partial orders.

In theory we may calculate all the pointcut and base traces to find matches. In practice this is an intractable problem since the number of traces may have an exponential growth relative to the number of events in the diagram. Instead we use a lifeline-based matching which is equivalent to the semantics-based matching.

A necessary, but not sufficient, criterion for lifeline-based matching is that for all lifelines, the pointcut events must occur as a continuous subsequence in the base diagram. If this is the case, there is a candidate match between the two diagrams.

**Definition 3 (Candidate match)** The mapping \( \phi \) defines a candidate match between a pointcut diagram \( d_{pc} \) and a base diagram \( d_b \) if and only if for all lifelines, the events in the pointcut diagram is a (possibly empty) continuous subsequence of the events in the base diagram (where the message of each event in the pointcut diagram is mapped by \( \phi \)). Formally:

\[
cmatch_\phi(d_{pc}, d_b) \overset{\text{def}}{=} \forall l \in \mathcal{L} : \text{substr}(\phi(d_{pc}[l]), d_b[l])
\]

Informally, we will refer to the range of \( \phi \) in the base diagram, as the mapped events. For a candidate match to be a proper lifeline-based match, there must be no match-blocking partial orders that require some unmapped events to occur between two of the mapped events in the base diagram. If there is a match-blocking partial order, the candidate match cannot produce a contained pointcut trace within a base trace, since there will always be intermediate events in the base trace.

**Definition 4 (Match-blocking partial order)** A match-blocking partial order with respect to \( \phi \) and a pointcut diagram \( d_{pc} \), is a partial order between two of the unmapped base events, and such that the partial order of the base diagram \( d_b \) requires these two events to happen between two of the mapped events. Formally:

\[
\text{blocking}_{d_{pc}, \phi}(a, b, d_b) \overset{\text{def}}{=} \exists e_1, e_2 \in \phi(\text{ev}.d_{pc}) : ((e_1, a) \in d_b.\text{po} \land (e_2, b) \in d_b.\text{po})
\]
Definition 5 *(Lifeline-based match)* The mapping $\phi$ defines a lifeline-based match between a pointcut diagram $d_{pc}$ and a base diagram $d_b$ if and only if $\phi$ defines a candidate match between the two diagrams and there are no match-blocking partial orders. Formally:

$$lmatch_{\phi}(d_{pc}, d_b) \stackrel{\text{def}}{=} cmatch_{\phi}(d_{pc}, d_b) \wedge \forall a, b \in ev.d_b : \neg\text{blocking}_{(d_{pc}, \phi)}(a, b, d_b)$$

Figure 5 illustrates that the match definition needs to exclude base diagram matches with match blocking partial orders. All lifelines of the base diagram contain the pointcut events as a continuous subsequence (taking $\phi$ to be the identity mapping), which means that there is a candidate match between the pointcut and the base diagram. However, the $c$ message is a match blocking message, with $(!c, ?c)$ as the match blocking partial order, meaning that this is not a proper lifeline-based match. This is because the $!c$ event is after the candidate match $(!a)$ on lifeline $L1$, while the $?c$ event is before the candidate match $(?b)$ on lifeline $L3$. I.e., both $(!a, !c)$ and $(?c, ?b)$ are partial orders for the base diagram, requiring the unmapped events $!c$ and $?c$ to occur between the two mapped events $!a$ and $?b$.

When investigating the traces, we see that there is no semantics-based match either. The pointcut has a single trace: $\langle !a, ?a, !b, ?b \rangle$. None of the six shown base traces have a contained pointcut trace, and thus there are no semantics-based matches. This is because the match blocking $c$ message will always get its two events between the first and last events of the matched pointcut trace.

If we modify the base diagram of Figure 5 such that the $c$ message switches direction ($!c$ on $L3$ and $?c$ on $L1$), then we get a match since the $c$ message is no longer match blocking. From all the direct partial orders involving the $c$ message, $(!c, ?c), (!a, ?c), (!c, ?b) \in d_b.dpo$, it is clear that $!c$ may happen before and $?c$ after all of the mapped events. Similarly, we get a semantics-based match since the set of base traces now includes the trace $\langle !c, !a, ?a, !b, ?b, ?c \rangle$.

![Figure 5](image-url)
Lifeline-based matching considers not only match blocking messages, but match blocking partial orders in general. To illustrate why this is necessary, we keep the pointcut from Figure 5 and apply it to the base diagram in Figure 6. Now the partial order relation contains \((c_1, c_2)\) which is a match blocking partial order. Thus, there is no match for the base diagram.

![Diagram](image)

Figure 6: \((c_1, c_2) \in \text{base.po}\) is a match blocking partial order

With lifeline-based matching, the problem of finding matches is now reduced from calculating all possible traces to searching for candidate matches and checking for match-blocking partial orders. The following lemma ensures that lifeline-based matching may be performed instead of semantics-based matching as they both give the same result.

**Lemma 2 (Lifeline-based equals semantics-based match)** The mapping \(\phi\) defines a lifeline-based match between a pointcut diagram \(d_{pc}\) and a base diagram \(d_b\) if and only if \(\phi\) defines a semantics-based match. Formally:

\[
lmatch_\phi(d_{pc}, d_b) \iff smatch_\phi(d_{pc}, d_b)
\]

**Proof:** As for the proof of lemma 1, we use the identity mapping for \(\phi\) without loss of generality.

\(\Rightarrow\): By definitions 2 and 1 of \(smatch\), we need to prove that there exists a pointcut trace \(t_{pc}\) and a base trace \(t_b\), with \(t_{pc}\) as a continuous subsequence, i.e. there exists sequences \(h_1\) and \(h_2\) such that \(t_b = h_1 \bowtie t_{pc} \bowtie h_2\).

By definitions 5 and 3 of \(lmatch\) (the assumption), we know that for all lifelines, the base diagram contains the mapped pointcut events as a continuous subsequence. Together with the message invariant, this means that the pointcut events in \(d_{pc}\) and the mapped base events in \(d_b\) have the same direct partial orders.

From the assumption and definition of \(lmatch\) (definition 5), there are no match blocking partial orders. Thus, for any arbitrary unmapped event \(a\), and mapped events \(e_1\) and \(e_2\) such that \((e_1, e_2) \in d_b.po\), at most one of \((e_1, a) \in d_b.po\) and \((a, e_2) \in d_b.po\) may hold. Hence, we may construct a base trace \(t_b\) where any unmapped event \(a\) occurs either before (in \(h_1\)) or after (in \(h_2\)) the mapped events \(t_{pc}\).

\(\Leftarrow\): We first prove that the assumption, \(smatch\), implies that there is a \(cmatch\). By definitions 2 and 1 of \(smatch\), we know that there exists a pointcut trace \(t_{pc}\) and a base trace \(t_b\) such that \(t_{pc}\) is a continuous subtrace of \(t_b\). It follows that for each lifeline, the event sequence in \(t_{pc}\) is a continuous subtrace of the event sequence for the same
lifeline in \( t_b \). For any basic sequence diagram, the event sequence for one lifeline is the same for all traces of the diagram, and the same as the the top-down sequence of syntactic events on that lifeline. Hence, for all lifelines, the events in the pointcut diagram is a continuous subsequence of the events in the base diagram and we have a candidate match.

In order to have a lifeline-based match \( lmatch \), it remains to prove that there are no match blocking partial orders in the base diagram \( d_b \). Assume for contradiction that there exists a match blocking partial order, i.e. two unmapped events \( a \) and \( b \), and two mapped events \( e_1 \) and \( e_2 \) in the base diagram \( d_b \) such that \((a, b) \in d_b.po\), \((e_1, a) \in d_b.po\) and \((b, e_2) \in d_b.po\) hold. This means that any base trace must include the sequence \( \langle e_1, a, b, e_2 \rangle \) as a (possibly non-continuous) subsequence. Any pointcut trace is on the form

\[
h_1 \sim e_1 \sim h_2 \sim e_2 \sim h_3
\]

where \( h_2 \) cannot include \( a \) or \( b \), since \( a \) and \( b \) are not mapped by pointcut events. However, this means that no pointcut trace can be a continuous subsequence of a base trace, which contradicts the assumption that there exists a semantics-based match. \( \Box \)

Hereafter we use the short term \( match \) to mean a semantics-based / lifeline-based match, and we allow to use \( match \) in formulas when it is irrelevant to differentiate between the two (equivalent) match formulas. With the match definition formalized, we are in a position to define isomorphic diagrams.

**Definition 6 (Isomorphic diagrams)**

If there exists a \( \phi \) which is one-to-one (both injective and surjective) between two diagrams \( d_1 \) and \( d_2 \) and where \( \phi \) defines a match of \( d_1 \) in \( d_2 \) (\( \phi^{-1} \) defines a match of \( d_2 \) in \( d_1 \)), then \( d_1 \) and \( d_2 \) are isomorphic diagrams. Formally:

\[
isomorph(d_1, d_2) \overset{\text{def}}{=} \exists \phi \in \Phi : \text{match}_\phi(d_1, d_2) \land \text{match}_{\phi^{-1}}(d_2, d_1)
\]

where \( \Phi \) is the set of all \( \phi \)'s

Throughout this paper we will implicitly consider two diagrams to be the same if they are isomorphic.

### 4.2 Weaving

In this paper we consider weaving to be the non-deterministic application of a set of aspects on a base diagram. A **direct derivation** is an atomic step in the weaving, where a single aspect is applied to a single match in the base diagram, while a **derivation** consists of zero or more direct derivations.

Throughout this paper we will adapt important terms and notations from term rewriting systems [1] and graph transformation systems [16] to the context of aspect diagrams.

The notation \( B_1 \rightarrow B_2 \) or \( B_2 \leftarrow B_1 \) means a direct derivation from diagram \( B_1 \) to diagram \( B_2 \) by applying some (not specified) aspect. The notation can be extended to
Confluence of Aspects for Sequence Diagrams

provide additional information: $B_1 \xrightarrow{A_1} B_2$ means a direct derivation which has applied the aspect $A_1$, and $B_1 \xrightarrow{A_1, \phi_1} B_2$ means a direct derivation which has applied the aspect $A_1$ with the mapping $\phi_1$.

The notation $\rightarrow$ means a derivation (consisting of zero or more direct derivations) with unspecified aspects. A normal form is a diagram on which there are no possible direct derivations. The notation $\overline{B}$ means that the diagram $B$ is a normal form. We say that two diagrams $B_1$ and $B_2$ are joinable if there exists derivations leading to a common (up to isomorphism) diagram $B_{\text{join}}$, i.e.

$$B_1 \xrightarrow{*} B_{\text{join}} \xleftarrow{*} B_2$$

The notation $B_1 \downarrow B_2$ means that $B_1$ and $B_2$ are joinable, while $B_1 \nmid B_2$ means that $B_1$ and $B_2$ are not joinable.

A set of aspects is terminating if and only if there exists no infinite derivation sequence for any base diagram. A set of aspects is confluent (globally confluent) if and only if all derivations from the same diagram are joinable, i.e.

$$B_1 \xleftarrow{*} B \xrightarrow{*} B_2 \Rightarrow B_1 \downarrow B_2$$

A set of aspects is locally confluent if and only if all direct derivations from the same diagram are joinable, i.e.

$$B_1 \leftarrow B \rightarrow B_2 \Rightarrow B_1 \downarrow B_2$$

Newman’s Lemma [13] proves that local confluence and confluence are equivalent for terminating systems. For most practical purposes a system is terminating, so we assume that we have terminating sets of aspects in this paper. Termination theory is however outside the scope of this paper.

In general our aspect language supports advice events on lifelines with no pointcut events, and this can be meaningful with respect to a particular base diagram [4]. A confluence analysis, however, considers arbitrary base diagrams, where such aspects normally lead to non-confluence. Thus, we disregard such aspects in this paper.

We are not allowed to delete lifelines, since deleting a lifeline with events may produce invalid sequence diagrams where a message contains only one of its two events.

**Definition 7** (Direct derivation) $d_w$ is a direct derivation of $d_b$ with respect to the aspect $A$ (with pointcut $d_{pc}$ and advice $d_{a}$) and the mapping $\phi$, written $d_b \xrightarrow{A, \phi} d_w$, if and only if there is a match between $d_{pc}$ and $d_{b}$ and for each lifeline in $d_{b}$, the events matched by $\phi$ are replaced by the entire subsequence of events (possibly empty) of the advice for that lifeline (where the advice events are also mapped by $\phi$). Formally:

$$d_b \xrightarrow{A, \phi} d_w \overset{\text{def}}{=} \underbrace{\text{match}_A(d_{pc}, d_b)} \land \forall l \in L : \exists h_1, h_2 \in \text{Event}^*: d_b[l] = h_1 \sim \phi(d_{pc}[l]) \sim h_2 \land d_w[l] = h_1 \sim \phi(d_a[l]) \sim h_2$$
We have extended the $\phi$ mapping so that it also maps from advice messages/events to base messages/events. Shared ids between the pointcut and the advice denote messages that are preserved. As a general convention, we assume that messages with the same signal, sender and receiver in the pointcut and the advice have the same id, unless the ids are explicitly visible in the diagram. When the convention is ambiguous, any assignment of ids can be chosen as long as the ids are unique within each advice diagram. The advice ids are insignificant to the effects of the weaving, since the advice diagrams with different ids are still isomorphic. The trace set is independent of the ids. The $\phi$ mapping of advice messages not shared with the pointcut must be given fresh ids with respect to the base diagram.

Figure 7 shows how the lifeline-based weaving works in a special notation where we do not show the messages, only events as filled circles on each lifeline. The matched sequence of events on each lifeline is marked by a rectangle. For each lifeline the matched subsequence of events in the base is replaced by the entire advice event sequence (empty in the case of lifeline $\text{L2}$) on the same lifeline.

![Figure 7: Weaving](image)

The criterion of excluding candidate matches having match blocking partial orders complicates our confluence theory. The following lemma justifies that this criterion cannot be ignored if we want to ensure a sound weaving.

**Lemma 3** (*The woven result is a valid sequence diagram*) Given an aspect $A$ with pointcut diagram $d_{pc}$, advice diagram $d_a$, a base diagram $d_b$, and a direct derivation $d_b \xrightarrow{\Lambda, \phi} d_w$, then the woven result $d_w$ is a valid sequence diagram. Formally:

$$\text{match}(d_{pc}, d_b) \land \square d_b \not= \emptyset \land \square d_a \not= \emptyset \Rightarrow \square d_w \not= \emptyset$$
Proof: Remember that we only consider valid base, pointcut and advice diagrams. Assume for contradiction that the woven result is an invalid sequence diagram, i.e.
\[ [d_b] \neq \emptyset, [d_a] \neq \emptyset \text{ and } [d_w] = \emptyset. \]

For any basic sequence diagram, its trace-set is empty if and only if the partial order relation contains a cycle. By \([d_w] = \emptyset\), we may then choose two events \(e_1\) and \(e_2\) from \(d_w\) such that \((e_1, e_2) \in d_w.po\) and \((e_2, e_1) \in d_w.po\) both holds.

As neither the base nor the advice diagram contains a cycle, the cycle in \(d_w\) cannot be the result of a “backwards” message on a single lifeline (i.e. a message with the receive event occurring before the send event in the top-down order on the lifeline). Instead, the cycle must contain events on at least two different lifelines, meaning that in the partial order path from \(e_1\), via \(e_2\), and back to \(e_1\), there must exist at least two direct partial orders between events on different lifelines. Such direct partial orders can only be a consequence of the message invariant, i.e. a partial order between the send and the receive event of the same message.

As neither the base nor the advice diagram contains a cycle, one of the two messages in the cycle must be a message (mapped by \(\phi\)) from the advice diagram \(d_a\), and the other a message from the base diagram not included in the match.

Consequently, there must exist a message \(a\) in \(\phi(msg, d_a)\) and a message \(b\) in \(msg, d_b\setminus \phi(msg, d_p)\), such that \((?a, !b) \in d_w.po\) and \((?b, !a) \in d_w.po\) both holds, i.e. the advice message must be received before the base message may be sent, and vice versa. This is illustrated in Figure 8.

![Figure 8: Match blocking messages prevent matches leading to invalid woven sequence diagrams](image)

However, the advice events are only inserted as replacements for the pointcut events on each lifeline (since we do not allow introducing events on lifelines without events in the pointcut), meaning that there must have existed some mapped events \(e_x\) and \(e_y\) such that \((e_x, !b) \in d_b.po\) and \((?b, e_y) \in d_b.po\) both held in the original diagram. However, this makes \(b\) a match-blocking message (with \((?b, !b) \in d_b.po\) a match-blocking partial order) as its two events is required to happen between the mapped events \(e_x\) and \(e_y\).

By definitions 5 and 4 of \(\text{match}\), no such match-blocking message may exist, and we have a contradiction. Hence, \([d_w] \neq \emptyset\) and the woven result \(d_w\) is indeed a valid sequence diagram. \(\Box\)

Figure 9 introduces an advice for the pointcut and base given in Figure 5. The advice inserts a \(d\) message after the matched \(a\) and \(b\) messages. If we ignore the match blocking partial order \((?c, !c)\), and allow a match in the base diagram, then the woven result will be an invalid sequence diagram because we get a cyclic partial order relation.
Now that we have introduced the match and derivation definitions, we may elaborate why we need the \( \phi \) function as an injective mapping from the pointcut events to base events. Consider the aspect and base diagram example in Figure 10.

The aspect defines that two consecutive \( a \) messages should be replaced by a \( b \) message, and the base diagram contains four consecutive \( a \) messages. Without using ids and the mapping function \( \phi \) in the match, we could mistakenly choose a match which does not pair the correct send and receive events (rectangles in the figure surround the matched events). By matching the last two events on the L1 lifeline and the first two events on the L2 lifeline, we get a final woven result with a crossing \( b \) message (notice that the \( b \) message is a match blocking message for the two remaining \( a \) messages). Crossing messages are allowed in general, but it is unexpected and undesired in this case.

The example in Figure 10 is non-confluent, since a non-deterministic matching strategy gives one of the following three alternative derivations with two different end results:
Confluence of Aspects for Sequence Diagrams

1. $a, a, a, a \rightarrow a, b, a$
2. $a, a, a, a \rightarrow a, a, b \rightarrow b, b$
3. $a, a, a, a \rightarrow b, a, a \rightarrow b, b$

4.3 Negative pointcuts

We now extend our aspects to include also negative pointcuts, i.e. an aspect $A$ is now a triple $(d_{pc}, D_{npc}, d_a)$ consisting of a pointcut diagram $d_{pc}$, a (possibly empty) set $D_{npc}$ of negative pointcut diagrams, and an advice diagram $d_a$.

Negative pointcuts are used to exclude matches between a pointcut and a base diagram (analogous to negative application condition in GTS [12]). An example is given in Figure 11. As a general convention, we assume that messages with the same signal, sender and receiver in the pointcut and one of the negative pointcuts have the same id, unless the ids are explicitly visible in the diagram. If the convention is ambiguous, then explicit ids are required since different assignment of ids may give different results.

In Figure 11, the leftmost pointcut diagram defines that we are looking for matches of two consecutive messages $a$ and $b$ within a base diagram, while the negative pointcut diagram defines that these messages should not be followed by a $d$ message. The base diagram in the figure has only one match (marked by the rectangle) where $\phi$ matches the pointcut to the first occurrence of the two consecutive messages $a$ and $b$. No $\phi$ can make the second occurrence of the two consecutive messages $a$ and $b$ to be a match, since the next message is a $d$ message, which the negative pointcut forbids.

In Figure 12 we modify the diagrams from Figure 11 by only changing the ids. The ids are now shown explicitly since they are different from the default convention. The $a$ and $b$ messages in the pointcut now have different ids than the $a$ and $b$ messages in the negative pointcut. This means that the $a$ and $b$ messages in the negative pointcut can be mapped to the same $a$ and $b$ messages, or to different ones when trying to match the negative pointcut. For the first occurrence of $a$ and $b$ in the base diagram, we can map the $a$ and $b$ in the negative pointcut to the second $a$ and $b$ messages, and the negative pointcut prevents a match. For the second occurrence of $a$ and $b$ in the base diagram, the $a$ and $b$ messages in the negative pointcut can be mapped to the same as for the
pointcut, and the negative pointcut once again prevents a match. Hence, there are no matches in the base diagram.

We now generalize definition 2 of semantics-based match to consider also aspects with negative pointcut diagrams.

**Definition 8** *(Semantics-based match with negative pointcuts)* The mapping \( \phi \) defines a semantics-based match between an aspect with pointcut diagram \( d_{pc} \) and negative pointcut diagrams \( D_{npc} \), and a base diagram \( d_b \) if and only if both of the following conditions hold:

- there is a semantics-based match between the pointcut and the base diagram
- for any negative pointcut diagram, it is not possible to find a semantics-based match between the negative pointcut diagram and the base diagram based on any \( \phi' \) having the same mapping as \( \phi \) for all messages shared with the pointcut diagram.

Formally:

\[
\text{smatch}_\phi((d_{pc}, D_{npc}), d_b) \overset{\text{def}}{=} \text{smatch}_\phi(d_{pc}, d_b) \land \\
\forall d_{npc} \in D_{npc} : \forall \phi' : \phi \Rightarrow \phi' \Rightarrow \neg \text{smatch}_{\phi'}(d_{npc}, d_b)
\]

where \( \phi \Rightarrow \phi' \) is a short-hand for \((\forall m \in \text{msg}.d_{pc} \cap \text{msg}.d_{npc} : \phi(m) = \phi'(m)) \) with \( \text{msg}.d \) being the set of all messages in the diagram \( d \).

The generalization of definition 5 of lifeline-based matching is similar.

**Definition 9** *(Lifeline-based match with negative pointcuts)*

This definition is equal to definition 8, where ‘semantics-based’ is substituted by ‘lifeline-based’, and ‘smatch’ is substituted by ‘lmatch’.

The following lemma is a generalization of lemma 2, stating that also for aspects with negative pointcuts, lifeline-based matching gives the same result as semantics-based matching.
Lemma 4 (Lifeline-based match with negative pointcuts equals semantics-based match with negative pointcuts) The mapping $\phi$ defines a lifeline-based match between an aspect with pointcut diagram $d_{pc}$ and negative pointcut diagrams $D_{npc}$, and a base diagram $d_b$ if and only if $\phi$ defines a semantics-based match. Formally:

$$lmatch_\phi(d_{pc}, D_{npc}, d_b) \iff smatch_\phi(d_{pc}, D_{npc}, d_b)$$

Proof: From Lemma 2, the first of the two and-clauses in the definitions 8 and 9 of $smatch$ and $lmatch$ with negative pointcuts, are equal (i.e. $smatch_\phi(d_{pc}, d_b) \iff lmatch_\phi(d_{pc}, d_b)$). It remains to prove that the second (and last) and-clauses are equal, i.e. for any negative pointcut, $d_{npc} \in D_{npc}$, and any $\phi'$ extending $\phi$ (i.e. $\phi \supset \phi'$), the following holds

$$\neg smatch_\phi'(d_{npc}, d_b) \iff \neg lmatch_\phi'(d_{npc}, d_b)$$

The above equivalence relation is the contrapositive of Lemma 2, which concludes the proof. □

This concludes the necessary generalization to handle also negative pointcuts. The negative pointcuts do not affect the weaving process described in Section 4.2, they only restrict the set of valid matches on which the corresponding aspect is applied. Definition 7, that defines a direct derivation, remains the same, except for substituting $'smatch_\phi(d_{pc}, d_b)'$ with the more general $'smatch_\phi(d_{pc}, D_{npc}, d_b)'$ to include possible negative pointcuts. Lemma 3, stating that the woven result is always a valid sequence diagram, also hold for negative pointcuts since negative pointcuts only restrict when we get matches to be used in the weaving.

4.4 The arbitrary events symbol

Figure 13 shows additional expressiveness by using the arbitrary events symbol (displayed as $\parallel$). The arbitrary events symbol indicates the presence of an arbitrary number of events (including zero events). Each lifeline in the pointcut diagram may contain zero or more arbitrary events symbols. A unique identifier is associated with each arbitrary events symbol, and the set of all arbitrary events symbols is referred to as $ArbEvt$. In the advice diagram, each arbitrary events symbol from the pointcut must be preserved, on the same lifeline and in the same order relative to the other arbitrary events symbols.

Figure 13: The Arbitrary Events Symbol
Our example aspect in Figure 13 defines that we are looking for matches of the \(a\) message followed by an arbitrary number of events on both its lifelines, and then finally \(a\ b\) message. The advice inserts a \(c\) message of which the position is uniquely defined in relation to the arbitrary events symbols. The send event of the \(c\) message, \(!c\), shall be inserted directly before all the arbitrary events (and after the \(!a\) event) on lifeline \(L1\), and the receive event, \(?c\), shall be inserted directly after all the arbitrary events (and before the \(?b\) event) on lifeline \(L2\).

We only allow for irreducible matches of arbitrary events symbols, which means that a proper match \(\phi\) cannot be reduced to a \(\phi^- \subset \phi\), where \(\phi^-\) is a match for the same aspect. This requirement means that the base diagram \(a,a,b\) in Figure 13 will only match the latter \(a\) message, \(match = a,b\), where the arbitrary events symbols are bound to empty event sequences, and \(a,a,b\) is not a match since it is reducible. Notice that this aspect is a plain additive aspect that will never terminate, so we will use match marking to exclude the previously matched \(a\) and \(b\) messages from further matches of the same aspect. We have displayed the match marking by a superscript number \((^1)\) to denote that aspect number one (the one defined in the example) cannot match the elements another time. The aspect will thus be applied only once and the final woven result will be \(a,a,c,b\).

If arbitrary events symbols are used also in the negative pointcuts, their ids will determine if they must be bound to the same events as the symbols in the pointcut. For each pointcut or negative pointcut diagram, the same base event cannot be matched by two different arbitrary events symbols in the same match. However, two different arbitrary events symbols in different diagrams (e.g. one pointcut and one negative pointcut diagram, or two negative pointcut diagrams) may be bound to overlapping sets of events.

We now formally define the matching for aspect diagrams with arbitrary events symbols. The definitions use a mapping \(\psi : ArbEvt \rightarrow Event^*\), which maps each arbitrary events symbol in the aspect to a (possibly empty) sequence of base events on the same lifeline. For a diagram \(d\), the notation \(d^\psi\) is used to denote the diagram \(d\) with every occurrence of the arbitrary events symbol replaced with the corresponding event sequence according to the mapping \(\psi\). Similarly, for a set of diagrams \(D\), \(D^\psi\) denotes that the mapping \(\psi\) has been used on every diagram in \(D\).

**Definition 10 (Semantics-based match with negative pointcuts and arbitrary events symbols)**

The mappings \(\phi\) and \(\psi\) define a semantics-based match between an aspect with pointcut diagram \(d_{pc}\) and negative pointcut diagrams \(D_{npc}\), and a base diagram \(d_b\) if and only if both of the following conditions hold:

- there is a semantics-based match between \(d_{pc}^\psi\) and \(d_b\) according to definition 8
- the semantics-based match is irreducible, i.e. for all \(\phi'\) and \(\psi'\) that maps a proper subset of the events mapped by \(\phi\) and \(\psi\), there is no semantics-based match.
Formally:

\[
\text{smatch}_{\phi, \psi}((d_{pc}, D_{npc}), d_b) \overset{\text{def}}{=} \text{smatch}_\phi((d_{pc}^\psi, D_{npc}^\psi), d_b)
\]

\[
\forall \phi', \psi' : \text{range} (\phi', \psi') \subset \text{range} (\phi, \psi) \Rightarrow \\
\neg \text{smatch}_{\phi'}((d_{pc}, D_{npc}^\psi), d_b)
\]

where \(\text{range}(\phi, \psi)\) is the union of all base events mapped to by \(\phi\) and \(\psi\).

The last clause in the definition ensures that the bound arbitrary events symbols in the pointcut cannot be reduced to a proper subset and still achieve a match.

The generalization of definition 5 of lifeline-based matching is similar. The mappings \(\phi\) and \(\psi\) define a lifeline-based match between an aspect and a base diagram only if it is not possible to achieve a match by selecting a proper subset of the mapped base events.

**Definition 11 (Lifeline-based match with negative pointcuts and arbitrary events symbols)**

This definition is equal to definition 10, where 'semantics-based' is substituted by 'lifeline-based', and 'smatch' is substituted by 'lmatch'.

The following lemma is a generalization of lemma 4, stating that also for aspects with arbitrary events symbols, lifeline-based matching gives the same result as semantics-based matching.

**Lemma 5 (Lifeline-based match with negative pointcuts and arbitrary events symbols equals semantics-based match with negative pointcuts and arbitrary events symbols)**

The mappings \(\phi\) and \(\psi\) defines a lifeline-based match between an aspect with pointcut diagram \(d_{pc}\) and negative pointcut diagrams \(D_{npc}\), and a base diagram \(d_b\) if and only if \(\phi\) and \(\psi\) defines a semantics-based match. Formally:

\[
\text{lmatch}_{\phi, \psi}((d_{pc}, D_{npc}), d_b) \iff \\
\text{smatch}_{\phi, \psi}((d_{pc}, D_{npc}), d_b)
\]

**Proof:**

The proof is straightforward using lemma 4.

Once the binding of the arbitrary events symbols in the aspect is resolved (by \(\psi\)), the weaving process is the same as for diagrams without arbitrary events symbols (described in Section 4.2). When it is useful the notation for direct derivation can include the exact \(\psi\) that has been used. \(B_1 \overset{\phi_1, \psi_1}{\Rightarrow} B_2\) means a direct derivation from diagram \(B_1\) to diagram \(B_2\) by using the aspect \(A_1\) with the mapping \(\phi_1\) and the arbitrary events binding \(\psi_1\).

Lemma 3 proved that our aspects without the arbitrary events symbol guaranteed a valid woven sequence diagram. This is not the case if we allow to freely place new messages in relation to the arbitrary events symbols. Figure 14 shows an aspect where we add a new message \(x\). The event \(!x\) is placed after an arbitrary events symbol on one lifeline, and the event \(?x\) before another arbitrary events symbol on another lifeline.
We need restrictions on how new messages are placed in relation to the arbitrary events symbols. The problem occurs when the match populates the arbitrary events in such a way that the advice diagram becomes invalid. Restriction: It is not allowed to add a new partial order \((a, b)\), where \(a\) is after an arbitrary events symbol on one lifeline and \(b\) is before an arbitrary events symbol on another lifeline. With the restriction above, we should be guaranteed valid woven diagrams.

Figure 14: The Arbitrary Events Symbol

When applying this aspect on the base model in Figure 14, the arbitrary events symbols are bound to a message \(b\). In the woven diagram, the message \(x\) and the message \(b\) constitute a cycle. The reason for this cycle is that the advice diagram becomes invalid with the binding of the arbitrary events symbols.

To ensure that an advice diagram is valid, we need restrictions on how messages are placed in relation to the arbitrary events symbols. We assumed that any advice diagram \(d_a\) was valid before introducing the arbitrary events symbol. The advice diagram we get by removing all arbitrary events symbols, denoted by \(d'_a\), shall still be a valid diagram. The following lemma ensures valid woven diagrams also for aspects with arbitrary events symbols:

**Lemma 6 (Valid woven diagrams for negative pointcuts and arbitrary events symbols)** Given an aspect \(A\) with pointcut diagram \(d_{pc}\), advice diagram \(d_a\) that does not introduce a partial order \((a, b)\) with \(a\) after an \(ArbEvs\) and \(b\) before an \(ArbEvs\) symbol, a base diagram \(d_b\), and a direct derivation \(d_b \xrightarrow{\Lambda, d_b} d_w\), then the woven result \(d_w\) is a valid sequence diagram. Formally:

\[
\text{match}_\psi(d_{pc}, d_b) \land \{ d_b \} \neq \emptyset \land \{ d'_a \} \neq \emptyset
\]

\[
\forall (a, b) \in (d_a, po \setminus d_{pc}, po), p_1, p_2 \in ArbEvs, l_1, l_2 \in L : \\
\forall (a \in \text{after}(p_1, d_a[l_1]) \land b \in \text{before}(p_2, d_a[l_2])) \Rightarrow \\
\{ d_w \} \neq \emptyset
\]

**Proof:**

From Lemma 3 it suffices to prove that the additional criteria in this assumption implies that any \(\psi\) resolution of the arbitrary events symbols give valid advice diagrams. Since the pointcut has a match in the valid base diagram, then the \(\psi\) resolution gives a valid pointcut diagram. From the assumption we know that the advice diagram without arbitrary events symbols, \(d'_a\), is valid. This means that any cyclic partial order must be caused by a combination of \(\psi\) events and a partial order \((e_1, e_2)\) introduced by the advice. To constitute a cycle, the \(e_2\) event must occur before an arbitrary events symbol on one lifeline, and the \(e_1\) event must occur after an arbitrary events symbol on another lifeline. The partial order \((e_1, e_2)\) corresponds to the partial order \((a, b)\), which is excluded in the assumption of the lemma. Hence, we can conclude that the woven diagram is valid. \(\square\)
Notice that the pointcut may have messages that become invalid for certain bindings of the arbitrary events symbols. Still, such bindings will never occur in any matches as long as we require the base diagram to be valid.

5 Independence

In confluence theory it is useful to establish an independence definition. We define two direct derivations from the same base diagram to be independent if they are commutable, i.e. the two derivations can be applied in any order.

Definition 12 (Independence)

Two direct derivations \( B_1 \xleftarrow{A_1, \phi_1, \psi_1} B \xrightarrow{A_2, \phi_2, \psi_2} B_2 \) from the same diagram \( B \) are independent if and only if there exists a diagram \( B_{\text{join}} \) such that

\[
B_1 \xrightarrow{A_2, \phi_2, \psi_2} B_{\text{join}} \xleftarrow{A_1, \phi_1, \psi_1} B_2
\]

For GTS there are two cases of dependence [12]: use-delete conflict and produce-forbid conflict. A use-delete conflict occurs when one rule deletes something in the left hand side of the other rule. A produce-forbid conflict occurs when one rule produces something that is matched by a negative application condition in the left hand side of the other rule.

For our aspects, the situation is more complicated than for GTS that have only two conflict types. We identify three possible conflict types for aspects without negative pointcuts, and two possible conflict types for aspects with negative pointcuts. First, we concentrate on aspects without negative pointcuts. We show three examples with dependent derivations before giving the general lemma.

The concept of a use-delete conflict from GTS is not directly transferable to aspect diagrams as we even with plain additive aspects may have non-confluence. Consider the example in Figure 15. We have two aspects \( A_1 \) and \( A_2 \) which are both plain additive, and where the matches of two direct derivations share the same \( b \) message. By applying the two aspects on the base diagram example, we get two result diagrams which cannot be joined. Thus, the two derivations must be dependent.

The reason why these two derivations are dependent can be found by investigating the lifeline event orders of the two aspects. Aspect \( A_1 \) uses the event order \( \langle ？a, !b \rangle \) on lifeline \( L_2 \), which is implicitly deleted by the aspect \( A_2 \) since it adds the new \!d event in between the two events. So, with respect to the lifeline event orders of the aspect diagrams, there is a use-delete conflict.

The prohibited match blocking partial orders may be considered as a kind of fixed negative pointcut for all aspects. This leads to what we have called produceMB-blocked conflicts (Figure 16). The MakeMB aspect produces a match blocking partial order (the message \( mb \)) for the other derivation, and thus the two derivations are dependent.

The third example of dependent derivations, which we call produce-produce conflict, occurs when two aspects both add events before or after the same common matched event. In Figure 17 both aspect derivations add a different event after the common
The two derivations are dependent even though their pointcuts have no common messages. The two derivations are dependent because MakeMB makes a match blocking partial order for A1. The order matters and the two derivations are dependent.

To avoid produce-produce conflicts we introduce the condition that event sequences added before or after a common matched event are equal. This is a necessary condition, but not sufficient, as illustrated in Figure 18. By extending the $\phi$ mappings to map advice messages and not only pointcut messages, we can avoid the problem in Figure 18. We cannot map the two b messages in each advice to the same base messages (including the same id), and still have equal event sequences (including ids) added after the common matched events !a and ?a.

Figure 19 shows another example of a produce-produce conflict, where all the added events before or after common matched events are equal (including mapped ids). The conflict occurs because only one of the two events of a message, ?adv, is added in relation to a common matched event (?b). The other event of the added message, !adv is placed in different positions in the base model. The order of the aspect matters,
Figure 17: **produce-produce** conflict where the two derivations produce unequal event sequences after a common matched event.

Figure 18: **produce-produce** conflict where we need to map advice messages to similar base messages.
since the A1-A2 derivation produces two new adv messages that are crossing, while the A2-A1 derivation produces two new adv that are not crossing.

The three dependency types explained above lead to the following lemma:

**Lemma 7** *(Independent derivations)*. For aspects without negative pointcuts and the arbitrary events symbol, two direct derivations \( B_1 \xleftarrow{A_1, \phi_1} B_2 \xrightarrow{A_2, \phi_2} B_2 \) are independent if and only if the following criteria are met:

1. \( \neg \text{use-delete} \). Example: Figure 15. None of the two derivations deletes a direct partial order which is part of the other derivation’s match. Formally:

   \[ \forall a, b \in ev.(Ax.pc) : \]  
   \[ (a, b) \in (Ax.pc).dpo \Rightarrow (\phi_x(a), \phi_y(b)) \in B_y.dpo \]  

   where \((x, y) \in \{(1, 2), (2, 1)\} \).

2. \( \neg \text{produceMB-blocked} \). Example: Figure 16. None of the two derivations produces a match blocking partial order for the other derivation’s match. Formally:

   \[ \forall a, b \in ev.B_x : \neg \text{blocking}_{(Ax.pc, \phi)}(a, b, B_x) \]  

   where \((x, y) \in \{(1, 2), (2, 1)\} \).
3. \(\neg\)\textit{produce-produce}.

**Part A:** Examples: Figure 17 and Figure 18. If there is a lifeline where the matches of the two derivations start (end) with the same event, the two derivations cannot add unequal event sequences before (after) that event (Example: ). Formally:

\[
\forall l \in L, e_1 \in \text{Event} : \\
 e_1 = \text{first}((A_1.pc)\{l\}) \land e_2 = \text{first}((A_2.pc)\{l\}) \\
\land \phi_1(e_1) = \phi_2(e_2) \\
\Rightarrow \\
( \text{before}(e_1, A_1.a) = \emptyset \lor \text{before}(e_2, A_2.a) = \emptyset \\
\lor \phi_1(\text{before}(e_1, A_1.a)) = \phi_2(\text{before}(e_2, A_2.a)) )
\]

\[
\forall l \in L, e_1 \in \text{Event} , e_2 \in \text{Event} : \\
 e_1 = \text{last}((A_1.pc)\{l\}) \land e_2 = \text{last}((A_2.pc)\{l\}) \\
\land \phi_1(e_1) = \phi_2(e_2) \\
\Rightarrow \\
( \text{after}(e_1, A_1.a) = \emptyset \lor \text{after}(e_1, A_2.a) = \emptyset \\
\lor \phi_1(\text{after}(e_1, A_1.a)) = \phi_2(\text{after}(e_2, A_2.a)) )
\]

The \(\phi\)'s are extended to also map advice messages (and not only pointcut messages) to base messages. This criterion is satisfied if there \emph{exists} \(\phi\) mappings of the advice messages such that the before and after relations can be satisfied.

**Part B:** Example: Figure 19. For each added message, either both its events or none, are placed in relation to a common matched event. From the other criteria, it is sufficient to check that either both or none the two events of a message are placed on the same lifeline as common matched events. Formally:

\[
\forall m \in M_{\text{add}} : \\
(\ ?.m.ll \in Evt_1 LLs \land !.m.ll \in Evt_1 LLs) \\
(\ ?.m.ll \notin Evt_1 LLs \land !.m.ll \notin Evt_1 LLs)
\]

where

- \(Evt_1 = \phi_1(\text{ev} (A_1.pc)) \land \phi_2(\text{ev} (A_2.pc))\) denotes common matched events,
- \(Evt_1 LLs\) denotes the set of lifelines w/ common matched events,
- \(M_{\text{add}} = \text{msg} (A_1.a \setminus A_1.pc) \cup \text{msg} (A_2.a \setminus A_2.pc)\) denotes all added messages, and
- \(!m/?.m\) denotes the send / receive events of a message \(m\).

**Proof:** Given in appendix A.5. □

A special case of the first criterion is when a common matched message \(m\) is deleted, since \((m, !m)\) is always a direct partial order. Another special case of the first criterion can occur for two derivations that both use the same plain additive aspect, where the aspect applies match marking to avoid termination. When both derivations use the same match marking aspect (with only partially overlapping matches), then the match marking of the overlapping message(s) appears as if it was a ‘deletion’ of a
Two derivations with no common matched events can only be dependent if there is a violation of criterion 2. The third criterion is not violated when only one of the derivations adds an event sequence after a common matched event, or when only one of the derivations adds an event sequence before a common matched event. Thus, the two derivations may be independent if one derivation adds an event sequence before a common matched event, while the other derivation adds an event sequence after the same common matched event.

Independence lemma 7 gives necessary and sufficient independence criteria when there are no negative pointcuts in the aspects. These criteria ensure that the advice diagram of one of the two aspects does not prevent a match for the pointcut of the other aspects. In the case where the aspect also contains one or more negative pointcut diagrams, the three criteria of lemma 7 are still necessary, but they are no longer sufficient to ensure independence. With negative pointcuts, there are two additional conflict types that make two derivations dependent.

The first case, which we call a deleteMB-forbid conflict, is illustrated in Figure 21. In this example, aspect A1 may be applied to the base diagram as the c message is a match-blocking partial order for the candidate match between the base diagram and the negative pointcut of A1. However, aspect A2 changes the direction of the c message, so that it is no longer blocking the negative pointcut of A1. Note that independence would have been restored had A2 not only removed the match-blocking partial order, but also destroyed the candidate match for the negative pointcut, for instance by also removing the b message.

The second case is illustrated in Figure 22. In this case, aspect A2 introduces a match for the negative pointcut of A1 by changing the c message to a b message, meaning that aspect A1 may no longer be applied. This corresponds to a produce-forbid conflict from GTS. In general, one of the sequence diagram aspects may very well introduce a candidate match for a negative pointcut in the other aspect, as long as the resulting diagram also has a match-blocking partial order for that candidate match.
Confluence of Aspects for Sequence Diagrams

Figure 21: deleteMB-forbid conflict where one derivation removes a match blocking partial order for a negative pointcut of the other aspect

Figure 22: produce-forbid conflict where one derivation produces a match for a negative pointcut of the other aspect

However, this is not the case in Figure 22. The usage of arbitrary events symbols in aspect diagrams does not affect the independence theory, and no additional independence criteria are needed. This is because the binding of the arbitrary events symbols is resolved in the match, and the weaving must preserve the events bound by the arbitrary events symbols. Hence, with a specific $\psi$, the aspect can be seen as being arbitrary event symbol free.

The examples in Figure 21 and Figure 22 lead to the following lemma for aspects which may also contain one or more negative pointcuts:

**Lemma 8** (Independent derivations with negative pointcuts and arbitrary events symbols). Two direct derivations $B_1 \xleftarrow{A_1, \delta_1, \phi_1} B \xrightarrow{A_2, \delta_2, \phi_2} B_2$ are independent if and only if the following criterion is met in addition to the criteria from lemma 7:
4. \( \neg \text{deleteMB-forbid} \) and \( \neg \text{produce-forbid} \). None of the two derivations produces a match for one of the other derivation’s negative pointcuts. This may be ensured either by keeping / producing a match-blocking message (the converse of criterion 2), or by making sure that for each negative pointcut, at least one of the partial orders is deleted / not produced by the derivation (the converse of criterion 1). Formally:

\[
\forall d \in A.x.Npc : \forall \phi'_x : \\
\phi_x \Rightarrow \phi'_x \\
( (\exists a, b \in ev.B_y : blocking(d,\phi'_x(a, b, B_y)) ) \\
\lor (\exists a, b \in ev.d : (a, b) \in d.dpo \land (\phi'_x(a), \phi'_x(b)) \notin B_y.dpo) )
\]

where \((x, y) \in \{(1, 2), (2, 1)\}\) and \(A.Npc\) is the set of negative pointcuts in the aspect \(A\).

Proof: Given in appendix A.7.

\[\square\]

6 Confluence is Undecidable

Plump [16] has shown that confluence is undecidable for terminating GTS by reduction of the well-known undecidable Post Correspondence Problem (PCP) [17]. This section shows that confluence is undecidable for one of the most expressive forms of aspect diagrams:

**Theorem 1 (Undecidable).** It is undecidable to determine if an arbitrary finite set of terminating aspect diagrams with deletion, negative pointcuts and the arbitrary events symbol is confluent.

The rest of this section is used to prove the theorem. First, we explain the PCP problem, then we present a method to construct a set of aspect diagrams based on an arbitrary PCP instance. We show that these aspect diagrams are terminating. Finally, we show that the constructed aspect diagrams are confluent if and only if the PCP instance has no solution. This is sufficient to prove Theorem 1.

An example instance of PCP from Wikipedia [21] (top part of Figure 23) has four numbered pairs of words over an alphabet with the letters \(a\) and \(b\). A solution to the PCP instance will be a sequence of these pairs such that the concatenation of the top letters equals the concatenation of the bottom letters. The sequence 1, 4, 3, 1 is a solution. Notice that a word pair may be repeated such as with pair number 1. Otherwise we could produce all possible combinations to decide PCP.

The decision problem is to decide if there exists a solution to a given PCP instance. A PCP instance consists of \(n\) word pairs where we denote pair number \(i\) as \((a_i, b_i)\). Furthermore, \(a_i\) and \(b_i\) are words consisting of characters from some alphabet \(\Gamma\).

The idea is to encode solution proposals, i.e. a sequence of indices, as a base diagram (topmost diagram in Figure 24). We use message signals to represent indices between 1 and \(n\), characters in the alphabet \(\Gamma\), and the special signals start and end. A
suggestion involves three lifelines. The Propose lifeline contains a sequence of indices representing a proposal. Aspects (defined in the next subsection) are defined so that lifeline A/B is first used to produce a full sequence of characters for the α/β part of the proposed word pairs, and finally to test if those two character sequences are equal.

For simplicity in the proof we use extensively messages where the sender and receiver lifelines are equal (which is allowed for sequence diagrams), e.g. the end message on the Propose lifeline. It is possible to do the proof without such messages by introducing a few additional lifelines.

A base diagram representing a PCP solution proposal will have two normal forms, the *success normal form* and the *fail normal form* as shown in the bottom part of Figure 24.

### 6.1 Reduction of PCP by making aspects

Based on a PCP instance $I$ we automatically produce a set of aspects, $\text{aspects}(I)$, as shown in Figures 25-28. This subsection explains the rationale behind these aspects.
We present aspect templates representing a set of aspects ranging over the possible indices \( \{1, \ldots, n\} \) or over the alphabet \( \Gamma \). The symbols \( a_i \) for all \( i \in \{1, \ldots, p\} \) and \( b_j \) for all \( i \in \{1, \ldots, q\} \) denote characters over the alphabet \( \Gamma \). \( p \) and \( q \) are natural numbers, where their value depend on the index \( i \). The aspects are carefully designed so that they will enforce a specific weave order: 1) init aspects, 2) test aspects, and 3) clean aspects.

An init aspect (init1 or init2) replaces an index \( i \) on the Propose lifeline by the character messages, corresponding to the word pair \((\alpha_i, \beta_i)\), onto the A and B lifelines. When all indices are consumed on the Propose lifeline, the test aspects will take over.

The init aspects are shown in Figure 25.

The test aspects (Figure 26) are mutually exclusive. The test1 aspects consume messages with equal character names from both the A and B lifelines, and these aspects are the only test aspects that keep the success message. The other test aspects (test2, test3 and test4) replace the success message with a fail message. The test2 aspects detect unequal character messages on the A and B lifelines, while the test3 and test4 aspects detect an unequal number of messages on the A and B lifelines. The test aspects will continue until they either remove all characters from both the A and B lifelines, or until the fail message is introduced.

The three sets of clean aspects (cleanP, cleanA, and cleanB) will continue to weave on failure marked diagrams until the fail normal form is reached. The clean aspects are shown in Figure 27.

The fail aspects (Figure 27) ensure that we can always produce the fail normal form from base diagrams that encode a PCP proposal. The rationale is that a base diagram has two normal forms (success and fail) only in cases where the base diagram encodes
We have defined three negative pointcuts to avoid weaving on ill-defined diagrams (Figure 28). These negative pointcuts all apply to each of the aspects having an index in the pointcut, i.e. \textit{init1}, \textit{init2}, \textit{cleanP} and \textit{fail}. The negative pointcuts ensure that there are only other indices (or nothing) between a matched index and its \texttt{end} message on the \texttt{Propose} lifeline. Identifiers (not shown) are used to bind the \texttt{i} and \texttt{end} messages in the pointcuts to the same \texttt{i} and \texttt{end} messages in the negative pointcuts. The first negative pointcut template contains the expression \#m : * \{m \neq 1..n\}, which means that the negative pointcut matches any message signal except valid indices.

\textbf{Lemma 9} \textit{aspects(\textit{I}) is terminating}

\textit{Proof}: No aspects add index-messages on the \texttt{Propose} lifeline. Thus, aspects removing index-messages from the \texttt{Propose} lifeline cannot be repeated infinitely. It suffices to prove that the set of aspects, excluding the \textit{fail}, \textit{init1}, \textit{init2} and \textit{cleanP} aspects, are
Lemma 10 If $I$ has a solution, then $\text{aspects}(I)$ is not confluent

Proof: A base diagram with the encoded solution, $i_1, \ldots, i_k$, in proposal state has both the success and fail normal forms.

We get the success normal form by first applying the aspect $\text{init}1$ once, then the aspect $\text{init}2$ $k - 1$ times. Since we assume that $i_1, \ldots, i_k$ is a solution, of the PCP instance $I$, we know that there will be equivalent sequences of character messages on the $A$ and $B$ lifelines. Thus, the aspect $\text{test}1$ can be applied to consume all these character messages until we reach the success normal form.
negative pointcuts:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Propose</td>
<td>#m: * {m \not= 1..n}</td>
<td>Propose</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>Propose</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

The three negative pointcuts apply to the aspects: fail, init1, init2, cleanP

Figure 28: aspects(I): Negative pointcuts for aspects fail, init1, init2 and cleanP

We get the fail normal form by first applying the fail aspect once, followed by applying the aspect cleanP k – 1 times. Thus, aspects(I) is not confluent.

Lemma 11 If I does not have a solution, then aspects(I) is confluent

Proof: From Newman’s Lemma and that aspects(I) is terminating (Lemma 9), it is sufficient to show that aspects(I) is locally confluent. The proof proceeds by trying to find a pair of direct derivations which is not joinable, i.e. to show that the aspects are not locally confluent. From definition 12, we know that such a pair of derivations must be dependent.

From Lemma 8, there are four criteria that can make two direct derivations dependent. We now consider these four criteria in the reverse order.

Criterion 4 is violated if we have a produce-forbid or a deleteMB-forbid conflict. There are no aspects introducing a message spanning two lifelines, so there cannot be a produce-forbid conflict for the two latter negative pointcuts. Only success and fail messages are introduced as non-index messages on the Propose lifeline, and the success and fail messages will always get the !start event directly above on the Propose lifeline. This means that we cannot get a produce-forbid conflict for the first negative pointcut either. deleteMB-forbid conflicts cannot occur. This is because all aspects only add and delete messages where the sender and receiver lifelines are the same.

Even though criterion 4 has not been formally defined for negative pointcuts with symbolic messages (e.g. * as message signal), it is easy to see that we cannot get produce-forbid conflicts here.

For criterion 3 that defines produce-produce conflicts, observe that for all aspects events are only added in between the matched start and end messages for all the three lifelines. Hence, criterion 3 can obviously not be violated.

For criterion 2 that defines produceMB-blocked conflicts, it is not possible to produce match blocking partial orders for the other derivation since for all added messages the sender and receiver lifelines are the same.

We can concentrate on criterion 1 that defines use-delete conflicts. There are only two pairs of aspects that can take part in dependent derivations with use-delete conflicts: 1) cleanA and cleanB, and 2) fail and init1. The negative pointcuts exclude other combinations. It is trivial to see that cleanA and cleanB can be commuted.
We now check the last possibility of non-confluence which are the two derivations

\[ B_1 \xleftarrow{\text{init}1} B \xrightarrow{\text{fail}} B_2 \]

These two derivations may have events external to the matches, i.e. before and after the match on the three involved lifelines, or anywhere on any other lifeline. Those external events will however not be affected when we join the two derivations, and we ignore such events to simplify the argumentation. We explain how both \( B_1 \) and \( B_2 \) leads to the fail normal form.

We apply the init2 aspect on \( B_1 \) until all index messages are removed from the Propose lifeline. We then apply the test1 aspect as long as possible (maybe zero times). Eventually, because \( I \) does not have a solution, we will be able to apply one of the aspects test2, test3 or test4 which changes the success message to a fail message. The aspects cleanA and cleanB are applied until we reach the fail normal form.

On \( B_2 \) we apply the cleanP aspects until we reach the fail normal form. Since \( B_1 \) and \( B_2 \) are joinable, we have shown that aspects(\( I \)) is confluent.

Proof of Theorem 1: For every PCP instance \( I \), the set of aspects constructed by aspects(\( I \)) is terminating (Lemma 9). Furthermore, \( I \) has a solution if and only if aspects(\( I \)) is not confluent (Lemmas 10 and 11). Since PCP is undecidable, the confluence of terminating aspects (w/ deletion, negative pointcuts and the arbitrary events symbol) must also be undecidable.

Figure 29 illustrates that lemma 11 depends on using negative pointcuts. Otherwise we get non-confluence for ill-formed base diagram examples, even if the PCP instance does not have a solution. Without the first negative pointcut, we may have a non-index message, such as trash, in the indices list, and still be able to apply aspects. The aspects fail and init1 can be applied to obtain two different normal forms.

![Figure 29: Undecidability proof breaks without negative pointcuts](image-url)
7 Critical Pairs

In this section we restrict the expressiveness of aspects to exclude negative pointcuts and the arbitrary events symbol. For this class of sequence diagram aspects we establish a critical pair analysis to determine confluence.

For term rewriting systems, all critical pairs are joinable if and only if there is confluence [10]. For GTS, confluence holds if all critical pairs are strongly joinable [16], where strong joinability is joinability and the additional requirement that all nodes preserved in the two derivations of the critical pair, must be maintained and be isomorphic in the joined result. This additional requirement is relevant only when there are unnamed nodes in the graph. For sequence diagrams, all messages and lifelines are named, and there is no difference between joinability and strong joinability.

In the following, we first (in Sect. 7.1) define critical pairs in the context of sequence diagrams, and present a systematic way to construct them. Sect. 7.2 explains why joinability of critical pairs does not necessarily imply confluence for sequence diagrams, in contrast to for term rewriting systems and GTS. As a consequence, a notion of extended critical pair is defined in Sect. 7.3, for which it is proved that joinability is equivalent with confluence.

7.1 Minimal context critical pairs

Analogously to term rewriting systems and GTS, we define a notion of critical pair for sequence diagram aspects. The term minimal context critical pair is used to distinguish these critical pairs from extended critical pairs which will be defined in Sect. 7.3. The following definition defines the notion of a minimal context base diagram, which will be used in the definition of minimal context critical pair. Intuitively, a minimal context base diagram with respect to two aspects, is a base diagram containing (possibly overlapping) matches for the two pointcut diagrams, and nothing more.

Definition 13 (Minimal context base diagram). A diagram $B$ is a minimal context base diagram with respect to two aspects $A_1$ and $A_2$ and the mappings $\phi_1$ and $\phi_2$ if all messages in $B$ are matched by at least one of the two pointcuts, i.e.

\[
\text{match}_{\phi_1}(A_1, pc, B) \land \text{match}_{\phi_2}(A_2, pc, B) \land \\
\text{msg}.B = \phi_1(\text{msg}(A_1, pc)) \cup \phi_2(\text{msg}(A_2, pc))
\]

A minimal context critical pair is now defined as two dependent direct derivations from the same minimal context base diagram:

Definition 14 (Minimal context critical pair). Two (different) direct derivations

\[
B_1 \xleftarrow{\ A_1, \phi_1} B \xrightarrow{\ A_2, \phi_2} B_2
\]

consitute a minimal context critical pair if the two derivations are dependent and $B$ is a minimal context base diagram for $A_1$ and $A_2$. 
Here, \( A_1 \) and \( A_2 \) may represent the same aspect. When \( A_1 = A_2 \), the mappings \( \phi_1 \) and \( \phi_2 \) must differ to make the two derivations different.

We now describe a systematic way to construct the minimal context critical pairs. The strategy is to investigate all two pairs of aspects (including pairs of the same aspect) to see if it is possible to construct two dependent derivations:

1. Start with the pointcut of \( A_1 \) and create an initial base diagram \( B_{\text{init}} = A_1.pc \) and a mapping \( \phi_1 \) such that for any message \( m \) in the pointcut of \( A_1 \), \( \phi_1(m) = m \). Put a red cross in all positions between two consecutive events on the same lifeline in \( B_{\text{init}} \). No events can be added to \( B_{\text{init}} \) at these positions since that would break the match between the base diagram and the pointcut of \( A_1 \).

2. Extend \( B_{\text{init}} \) into \( B \) by adding messages from the pointcut of \( A_2 \) and creating a mapping \( \phi_2 \) such that
   - (a) \( \phi_2 \) defines a match between \( A_2 \) and \( B \), and
   - (b) for each message \( m_2 \) in the pointcut of \( A_2 \)
     - (i) either, \( \phi_2(m_2) = m_2 \) and \( m_2 \) is not a message in \( B_{\text{init}} \),
     - (ii) or, \( \phi_2(m_2) = m_b \) where \( m_b \) is a message in \( B_{\text{init}} \). In this case, \( m_b \) will be a common matched message for \( A_1 \) and \( A_2 \).

3. \( B \) is now a minimal context base diagram for \( A_1 \) and \( A_2 \). The two direct derivations \( B_1 \xleftarrow{A_1,\phi_1} B \xrightarrow{A_2,\phi_2} B_2 \) is a minimal context critical pair if the two derivations are dependent according to Lemma 7.

The second step needs to be repeated for all possible additions of the messages in the pointcut of \( A_2 \) to \( B_{\text{init}} \). It is necessary to also consider crossing messages when constructing the minimal context base diagram, such as with the base diagram in Figure 16.

### 7.2 Minimal context critical pairs are not good enough

In contrast to term rewriting systems and GTS, an investigation only of minimal context critical pairs is not good enough to claim local confluence of our aspect diagrams. We illustrate the problem by two examples.

In the first example (Figure 30) there are three aspects. The aspects \( A_1 \) and \( A_2 \) add an \texttt{adv} message, while the aspect \( A_3 \) replaces two consecutive \texttt{adv} messages with a single \texttt{adv} message. This set of aspects has two critical pairs. The first critical pair (shown in the middle part of the figure) is a base diagram with the five messages \( a, b, c, d \) and \( e \), and its two derivations using the aspect \( A_1 \) and the aspect \( A_2 \). The \( A_2 \) derivation leads directly to a normal form, because the \texttt{!adv} event on the \( L_2 \) lifeline prevents the application of aspect \( A_1 \), and aspect \( A_3 \) is not applicable as there are no two consecutive \texttt{adv} messages in the diagram. The \( A_1 \) derivation in the critical pair results in a diagram on which it is possible to apply aspect \( A_2 \), and then aspect \( A_3 \). Finally, we get the same normal form as with the \( A_2 \) derivation of the critical pair, i.e. the critical pair is joinable.
The second critical pair (not shown in the figure) is a base diagram with three consecutive adv messages, and with two overlapping matches for two different A3 derivations. This critical pair has already been joined since the two derivations produces the same result, which is two consecutive adv messages.

In spite that all critical pairs are joinable, the bottom part of Figure 30 shows that the set of aspects is not confluent. As our base diagram we have added a new message, x, in between the two matches in the base diagram of the A1 – A2 minimal context critical pair. The x message is between the matches on the L1 lifeline since the ?x event is after all the matched events in the A1 derivation (!a), while it is before all the events in the A2 derivation (?c).

Notice that when applying the A1 and A2 aspects on this base diagram it is unam-
Figure 31: Joinable critical pairs does not imply local confluence due to added match blocking partial order.
biguous where to place the new $\texttt{?adv}$ events on the L1 lifeline. In the $A1$ derivation it is placed directly after $!a$ (and thus before $?x$), and for the $A2$ derivation it is placed directly before $?c$ (and thus after $?x$). This is because the new events in the advice relates to the old events matched by the pointcut.

The result is that the $?x$ event prevents application of the aspect $A3$, which was used in joining the critical pair, since the $?x$ event splits $A3$’s required event order on lifeline L1. Instead we reach two normal forms which are different, and we have shown that the set of aspects is not confluent.

In the second example (Figure 31) there are also three aspects. The three aspects are plain additive and they all add an $\texttt{adv}$ message. $A1 – A2$ derivations for a base diagram in the middle part of Figure 31 shows the only minimal context critical pair. This critical pair is joinable by applying the $A3$ and $A1$ aspects on the two woven diagrams in the critical pair.

In the bottom part, we have introduced a new message $x$, which does not prevent the first $A1$ and $A2$ derivations. However, the $A3$ aspect is no longer applicable since $(!x, ?x)$ becomes a match blocking partial order. We reach two normal forms which are different, and we have shown that the set of aspects is not confluent.

### 7.3 Extended critical pairs

In this section we define a way to extend each minimal context critical pair such that all the extended critical pairs are joinable if and only if the set of aspects is confluent. The minimal context critical pair is extended in a systematic way such that it is maximally difficult to join the extended critical pair. This is accomplished by reducing the applicable aspects in two ways: 1) the introduction of new events (Figure 30), and 2) the introduction of match blocking partial orders (Figure 31). Both these extensions may prevent matches and hence the application of aspects.

We investigate the counter example in Figure 30 to see why joinability of minimal context critical pairs is not good enough. All lifeline positions of the minimal context base diagram in the critical pair are analysed to see how additional events in the base diagram may affect the joinability.

Figure 32 shows how additional events may be categorized as one of three types, depending on their lifeline position relative to the events in the minimal context base diagram: $\alpha$, $\beta$, $\omega$ and prohibited. An event that occurs in an $\alpha$-position, is placed before both matches of the critical pair with the respect to a lifeline. The $?x$ event in the bottom of Figure 31 is in an $\alpha$-position. An event that occurs in a $\beta$-position, lies between the two pointcut matches relative to one lifeline, i.e. after all matched events from the first pointcut and before all matched events from the second pointcut. The $?x$ event in the bottom of Figure 30 is in a $\beta$-position. An event that occurs in an $\omega$-position is placed after both matches of the critical pair relative to a lifeline. The $!x$ event in the bottom of Figure 31 is in an $\omega$-position.

In prohibited positions in the critical pair we cannot place any events since they would prevent at least one of the two matches. E.g. an event between the $?a$ event and the $!b$ event prevents the $A1$ derivation of the critical pair. Prohibited positions are shown as crossed out squares in Figure 32.
In an extended critical pair, we introduce messages that lead to potentially match blocking partial orders for the aspects used to join the two derivations in the critical pair. At the same time, the introduced messages cannot introduce match blocking partial orders that prevents the two derivations of the critical pair. The messages are only sent or received at $\chi$-positions $= \alpha \cup \beta \cup \omega$. We use the special symbol, $\chi$, as the message symbol, and assume this message symbol is not part of any of the aspect pointcuts.

Figure 33 shows all the relevant message directions that can introduce match blocking partial orders in the relevant subpart of the lifelines. The three ellipses represent the set of $\alpha$-, $\beta$-, and $\omega$-positions. The directed arrows indicate between which two kinds of positions, an introduced message is relevant to insert. A message which is sent from an $\alpha$-position cannot introduce a match blocking partial order for any aspects in the relevant lifeline subparts, and thus there is no directed arrow from the $\alpha$-positions ellipse. From the figure, we see the relevant message directions: from an $\omega$-position to a $\beta$-position, from an $\omega$-position to an $\alpha$-position, from a $\beta$-position to a $\beta$-position, and from a $\beta$-position to an $\alpha$-position.

**Definition 15** ($\chi$-message). Given a minimal context base diagram $B$, a $\chi$-message is

- a message with signal $\chi$, and
Confluence of Aspects for Sequence Diagrams

- a message between two relevant $\chi$-positions, where the relevant directions are given by Figure 33,
- a message where the send and receive events occur on the same lifeline, must be a message from and two a $\beta$-position

**Definition 16 (Extended critical pair).** Given a minimal context critical pair

$$ B_1 \xleftarrow{A_1, \phi_1} B \xrightarrow{A_2, \phi_2} B_2 $$

We denote its extended critical pair by

$$ B^\chi_1 \xleftarrow{A_1, \phi_1} B^\chi \xrightarrow{A_2, \phi_2} B^\chi_2 $$

where $B^\chi$ is constructed from $B$ as follows:

- insert one $\chi$-message in each relevant direction (as defined in definition 15) if:
  - the $\chi$-message does not introduce a match blocking partial order for any of the two matches $\phi_1$ or $\phi_2$.
  - the $\chi$-message can be inserted such that it does not introduce cyclic partial orders, i.e. invalid sequence diagram
- $B^\chi$ only contains a $\chi$-message from and to the same $\beta$-position if there are no other $\chi$-messages with its send or receive event in the same $\beta$-position

We refer to $B^\chi$ as the extended base diagram. An extended base diagram that does not contain any $\chi$-messages, corresponds to the minimal context base diagram.

For the $A1 - A2$ minimal context critical pair in Figure 30 we now investigate the 10 possible $\chi$-positions shown in the top left part of Figure 34. For each of these 10 positions we consider the possible $\chi$-messages being sent from its position. No $\chi$-messages can be sent from any of the $\alpha$-positions 1, 4, 6, and 8.

**Position 2:** Messages from position 2 to positions 4, 6 or 8 are impossible, since they introduce match blocking partial orders for $A_1$’s match. A $\chi$-message from position 2 to position 9 shall be part of the extended base diagram according to definition 16.

**Position 3:** Messages from position 3 to positions 4, 6 or 8 are impossible, since they introduce match blocking partial orders for both $A_1$ and $A_2$’s matches. A message from position 3 to position 9 is match blocking for $A_2$’s match. Hence, no $\chi$-messages are added from position 3.

**Position 5:** Messages from position 5 to positions 1 or 2 are impossible, since they introduce cyclic partial orders in the extended base diagram. Messages from position 5 to 6 or 8 are match blocking for both $A_1$ and $A_2$’s matches, while a message from position 5 to position 9 is match blocking for $A_2$’s match. Hence, no $\chi$-messages are added from position 5.

**Position 7:** Messages from position 7 to positions 2 or 9 are match blocking for $A_2$’s match. Messages from position 7 to positions 1 or 4 introduce cyclic partial orders, while a message from position 7 to position 8 is match blocking for both $A_1$ and $A_2$’s matches. Hence, no $\chi$-messages are added from position 7.
Position 9: Messages from position 9 to positions 1 or 4 are match blocking for A1’s match. A message from position 9 to position 6 introduces cyclic partial orders. A \(\chi\)-message from position 9 to position 2 shall be part of the extended base diagram according to definition 16.

Position 10: Messages from position 10 to positions 1, 4 or 6 all introduce cyclic partial orders. A message from position 10 to position 2 is match blocking for A2’s match. Hence, no \(\chi\)-messages are added from position 10.

Finally, our extended base diagram is constructed as shown in Figure 34. Notice that there are two added events in both the positions 2 and 9. By ordering these differently, we get four different extended base diagrams. One of these four diagrams contains cyclic partial orders, and cannot be used. The other three alternatives are equally good for our purpose, and we non-deterministically choose one.

For our other example in Figure 31, the extended base diagram becomes identical (except for having a message symbol \(x\) instead of \(\chi\)) to the base model in the bottom of Figure 31 that is used to show non-confluence.

Theorem 2 (Confluence). A set of terminating aspects (without the arbitrary events
symbol and without negative pointcuts) is confluent if and only if all the extended critical pairs are joinable.

Proof:

only-if-direction. This is trivial, since only one non-joinable extended critical pair serves as a counter-example against confluence.

if-direction. It is sufficient to prove that non-confluence implies that there exists at least one non-joinable extended critical pair. For contradiction we assume non-confluence and that all extended critical pairs are joinable.

With non-confluence there exists two direct derivations $B_1 \xleftarrow{A_1,\phi_1} B \xrightarrow{A_2,\phi_2} B_2$, where $B_1 \parallel B_2$. From definition 12, we know that these two derivations must be dependent. Let $B' = \phi_1(\text{msg}.(A_1, \text{pc})) \cup \phi_2(\text{msg}.(A_2, \text{pc}))$

Let $B_1'$ and $B_2'$ be the corresponding derivation results in $B_1 \xleftarrow{A_1,\phi_1} B' \xrightarrow{A_2,\phi_2} B_2'$. These two derivations constitute a minimal context critical pair.

We get an extended critical pair $$(B_1')^x \xleftarrow{A_1,\phi_1} (B')^x \xrightarrow{A_2,\phi_2} (B_2')^x$$

From our assumption (that shall give a contradiction) we conclude that $(B_1')^x \parallel (B_2')^x$.

Thus there exist derivation sequences: $(B_1')^x \rightarrow B_{\text{join}} \leftarrow (B_2')^x$, for some diagram $B_{\text{join}}$.

All the derivations in $(B_1')^x \rightarrow B_{\text{join}}$ can be repeated (each derivation step uses the same aspect and match) in $B_1 \rightarrow B_{\text{join}}$

This claim holds for the following reasons: 1) no $\chi$-messages are matched or affected by any derivation, 2) the messages that are taken back does not introduce new match blocking partial orders for any of the used matches, since the extended critical pair is constructed to have the maximum amount of potential match blocking partial orders in the relevant area, 3) the messages that are taken back in $\alpha$-positions and $\omega$-positions can be seen as wrappers around the derivations, and 4) the messages that are taken back in $\beta$-positions are placed where there previously was at least one $\chi$-message event.

Correspondingly, we can conclude that all the derivations in $(B_2')^x \rightarrow B_{\text{join}}$ can be repeated (each derivation step uses the same aspect and match) in $B_2 \rightarrow B_{\text{join}}^2$. We deduce that $B_{\text{join}} = B_{\text{join}}^2$, since all derivations are similar. We have only replaced the $\chi$-messages that remain unchanged in the derivations, by some other messages that also remain unchanged in the derivations.

We have shown that $B_1 \parallel B_2$, which is a contradiction that concludes our proof. $\square$

The production of all extended critical pairs, and the joinability check, can be fully automated. All the minimal context critical pairs are constructed by taking all pairs of aspects to see if the pointcuts can be be combined to constitute minimal context base diagrams in dependent derivations. From definition 16 we produce the set of all extended critical pairs from the minimal context critical pairs. Since we assume that the set of aspects is terminating, it follows from Newman’s lemma that the joinability corresponds to the equivalence of normal forms. Hence, confluence can be determined by a fully automated algorithm.
8 Related Work

Whittle et al. [20] use syntactic-based sequence diagram aspects to support variability modeling. The models are translated into abstract syntax as graphs and GTS rules in the GTS tool AGG [19], which reports on conflicting critical pairs. They do not investigate the decidability of confluence with respect to the expressiveness of the aspect diagrams. For the class of aspect diagrams where we use the extended critical pair analysis to decide confluence (yes or no answer in all cases), they will get two kinds of answers, either yes or maybe.

Whittle et al. do not provide a formal nor a precise definition of a match, and they do not present the details of how the sequence diagram aspects are mapped to graph transformation rules. Unfortunately, this means that a further comparison against our work is not possible.

We are not aware of any related work addressing confluence analysis based on the concrete syntax of sequence diagram aspects. The rest of this section discusses if our confluence results are applicable to other sequence diagram aspect proposals.

Solberg et al. [18] define aspects for sequence diagrams by tagging the base diagram elements to be affected, which means that there is no non-deterministic aspect application and a confluence analysis is superfluous.

Clarke and Walker [2] and Deubler et al. [3] both define sequence diagram aspects limited to single event matching only and with synchronous messages. Although our decidability result of theorem 2 seems valid for these two approaches, the advice normally introduces events on lifelines not part of the match, which in the general case leads to non-confluence. With single event matching the extended critical pairs reduce to traditional critical pairs.

Klein et al. [8] define semantics-based sequence diagram aspects with four alternative matching strategies. Theorem 2 can be applied to their enclosed part and strict part matchings to decide confluence. The other two matching strategies of Klein et al. [8] assume implicitly that there are arbitrary events allowed in any position. However, they do not have an unambiguous definition on how to merge new events with the arbitrary events, which means that we always have non-confluence in the general case. Unless special care is taken, the merging of new events with arbitrary events may lead to invalid woven sequence diagrams. Klein et al. have not described support for negative pointcuts.

9 Future Work

As future work it remains to investigate if it is decidable to determine confluence for other classes of aspect diagrams, e.g. allowing either the arbitrary events symbol or negative pointcuts, but not both, or plain additive aspect diagrams with both the arbitrary events symbol and negative pointcuts.

For the classes of aspect diagrams where confluence is undecidable, future work should still identify a set of sufficient criteria to conclude that confluence holds.

For simplicity we have not included symbolic message signals (e.g. * to represent arbitrary message signals) in our formalism. It should be quite straightforward to
Confluence of Aspects for Sequence Diagrams

extend the match definitions to support matching message signals based on regular expressions for strings. However, for the construction of critical pairs, symbolic message signals may be more difficult. Two aspects may for instance have related messages, where one message signal is a specialization of the other. We need further investigation to see how this influences the confluence analysis.

Since our confluence results depend upon terminating sets of aspects, future work to establish a termination theory would be a strong complement to the confluence theory presented in this paper. We consider to implement a confluence test algorithm in our aspect weaver tool, to support the extended critical pair analysis.

10 Conclusions

This paper presents a sequence diagram aspect language with formal match and weave definitions. The aspect language includes support for arbitrarily large pointcut descriptions, negative application conditions, and allowing an arbitrary number of events in predefined positions. A formal trace model for sequence diagrams is used as the basis for the match definition. It is proven that our match and weave definitions ensure valid woven sequence diagrams.

We have shown how the features of an aspect language for sequence diagrams constitute a trade-off between high expressiveness and decidability of confluence. This paper proves that confluence is decidable for one class of aspect diagrams (as opposed to the more general graph transformation systems), while it is undecidable for another more expressive class of aspect diagrams. In the decidable case we needed to extend traditional critical pair analysis to be able to detect all the possible conflicts.

Acknowledgement

The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project). We thank Stein Krogdahl for his valuable feedback.

References


Confluence of Aspects for Sequence Diagrams


A Appendix with proofs

A.1 Helping lemma for lemma 1 (All pointcut traces are matched)

Lemma 12 If the mapping \( \phi \) defines a semantics-based match between a pointcut diagram \( d_{pc} \) and a base diagram \( d_{b} \), then there is a direct partial order between two pointcut events in the pointcut diagram if and only if there is a direct partial order between the corresponding mapped events in the base diagram.

Formally:

\[
\text{smatch}_\phi(d_{pc}, d_{b}) \Rightarrow \\
\forall a, b \in \text{ev} \cdot d_{pc} : ((a, b) \in d_{pc}.dpo \Leftrightarrow (\phi(a), \phi(b)) \in d_{b}.dpo)
\]

Proof:

(1) Assume: \( \text{smatch}_\phi(d_{pc}, d_{b}) \)

Proof: \( (a, b) \in d_{pc}.dpo \Leftrightarrow (\phi(a), \phi(b)) \in d_{b}.dpo \) for arbitrary \( a, b \in \text{ev} \cdot d_{pc} \)

(2) Choose \( t_{pc} \in d_{pc} \) and \( t_{b} \in d_{b} \) such that \( \text{match}_\phi(t_{pc}, t_{b}) \)

Proof: The assumption in step (1) and definition 2 of \( \text{match} \).

(2.1) \( a \in t_{pc} \land b \in t_{pc} \)

Proof: (1.1), (2.1) and \( d_{pc} \) being a basic sequence diagram.

(2.3) \( a \in t_{b} \land b \in t_{b} \)

Proof: (2.1), (2.2), \( \phi \) is the identity mapping, and definition 1 of \( \text{match} \).

(2.4) \( a \in \text{ev} \cdot d_{b} \land b \in \text{ev} \cdot d_{b} \)

Proof: (2.1) and (2.3).

(2.5) \( (a, b) \in d_{pc}.dpo \Leftrightarrow (a, b) \in d_{b}.dpo \)

(3) Case: \( a \) is the send and \( b \) the receive event of the same message, i.e.

\( a.m = b.m \)
Proof: By (1)1 and (2)4, \( m \) is a message in both \( d_{pc} \) and \( d_{b} \). (2)5 then follows from the message invariant stating that for all messages, there is a direct partial order between the send and the receive event of the message.

(3)2. Case: \( a \) and \( b \) occur on the same lifeline, i.e.
\[
  a.l = b.l
\]
(4)1. \((a, b) \in d_{pc}, \text{dpo} \land a.l = b.l\)
\[
  \Leftrightarrow \text{substr}(a, b), t_{pc} \upharpoonright a.l)
\]
Proof: (2)1, (2)2, \( d_{pc} \) being a basic sequence diagram, and the relationship between direct partial orders and the traces of a sequence diagram.

(4)2. \((a, b) \in d_{b}, \text{dpo} \land a.l = b.l\)
\[
  \Leftrightarrow \text{substr}(a, b), t_b \upharpoonright a.l)
\]
Proof: (2)1, (2)3, \( d_{b} \) being a basic sequence diagram, and the relationship between direct partial orders and the traces of a sequence diagram.

(4)3. \text{substr}(a, b), t_{pc} \upharpoonright a.l) \Leftrightarrow \text{substr}(a, b), t_b \upharpoonright a.l)

Proof: By (2)2 and (2)3, \( a \) and \( b \) is in both \( t_{pc} \) and \( t_b \). By (2)1 and definition (1) of \text{match}, \( t_{pc} \) is a substring of \( t_b \). Hence, \( t_{pc} \upharpoonright a.l \) is a substring of \( t_b \upharpoonright a.l \) and \( (a, b) \) is a substring of either both or none of them.

(4)4. Q.E.D.

Proof: (4)1, (4)2, (4)3.

(3)3. Q.E.D.

Proof: The cases are exhaustive as a direct partial order is either a consequence of the message invariant or the result of two events occurring (directly) after each other on the same lifeline.

(2)6. Q.E.D.

Proof: (2)5.

(1)2. Q.E.D.

Proof: \( \Rightarrow \)-rule and \( \forall \)-rule.

\[ \square \]

A.2 Formal proof of Lemma 2 (Lifeline-based equals semantics-based match)

\[ l\text{match}_{d}(d_{pc}, d_{b}) \Leftrightarrow s\text{match}_{d}(d_{pc}, d_{b}) \]

Proof:

(1)1. Assume: \( l\text{match}_{d}(d_{pc}, d_{b}) \)

Prove: \( s\text{match}_{d}(d_{pc}, d_{b}) \)

(2)1. \( \forall l \in L : \text{substr}(\phi(d_{pc}([l])), d_{b}([l])) \)

Proof: (1)1 and Def. of \text{match}.

(2)2. \( \forall a, b \in \text{ev} \in d_{pc} : (a, b) \in d_{pc}. \text{dpo} \Rightarrow (\phi(a), \phi(b)) \in d_{b}. \text{dpo} \)

(3)1. Assume: \( \forall a, b \in \text{ev} \in d_{pc} : (a, b) \in d_{pc}. \text{dpo} \)

Prove: \( (\phi(a), \phi(b)) \in d_{b}. \text{dpo} \)

(4)1. Case: \( a = \lambda x, b = ?x \)

Proof: \( (\phi(!x), \phi(?x)) \in d_{b}. \text{dpo} \) follows directly from how \( \phi \) works.

(4)2. Case: \( a \) and \( b \) occur on the same lifeline \( l \)
Confluence of Aspects for Sequence Diagrams

(5)1. $d_{pc}(l) = h_1 \sim a \sim b \sim h_2$, with some event sequences $h_1$ and $h_2$.
   Proof: Follows directly from (3)1 and the definition of $dpo$.
(5)2. $\text{substr}((\phi(h_1) \sim \phi(a) \sim \phi(b) \sim \phi(h_2), d_{pc}(l)))$
   Proof: (2)1 and (5)1.
(5)3. Q.E.D.
   Proof: (5)2 and the definition of $dpo$.
(4)3. Q.E.D.
   Proof: $dpo$ relations is either the message invariant or occurrences on the same lifeline. Hence, the cases (4)1 and (4)2 are exhaustive.
(3)2. Assume: $\forall a, b \in ev.d_{pc}: (\phi(a), \phi(b)) \in d_{b, dpo}$
   Proof: $(a, b) \in d_{pc,dpo}$
   (4)1. Case: $\phi(a) = !x, \phi(b) = ?x$
       Proof: $(a, b) = (y, y)$ for some $y$ due to how $\phi$ works.
   (4)2. Case: $\phi(a)$ and $\phi(b)$ occur on the same lifeline $l$
       (5)1. $h_1 \sim \phi(a) \sim \phi(b) \sim h_2$ is the event order on lifeline $l$ in $d_{b}$, for some event sequences $h_1$ and $h_2$
       Proof: The definition of $dpo$
       (5)2. $h_3 \sim a \sim b \sim h_4$ is the lifeline $l$ in $d_{pc}$, for some event sequences $h_3$ and $h_4$
       Proof: (5)1, (2)1, and how $\phi$ works
(5)3. Q.E.D.
   Proof: (5)2 and the definition of $dpo$
(4)3. Q.E.D.
   Proof: $dpo$ relations is either the message invariant or occurrences on the same lifeline. Hence, the cases (4)1 and (4)2 are exhaustive.
(3)3. Q.E.D.
   Proof: (3)1, (3)2, $\leftrightarrow$-rule, and $\Rightarrow$-rule.
(2)3. $\exists l_{pc} \in \llbracket d_{pc} \rrbracket, t_b \in \llbracket d_b \rrbracket$
   Proof: All diagrams are valid.
(2)4. $\forall e_1, e_2 \in ev.d_{pc}: (e_1, e_2) \in d_{b, dpo} \Rightarrow \neg((e_1, a), (a, e_2) \in d_{b, dpo})$
   Proof: Assumption (1)1, definition of match blocking partial order, and definition of $\text{match}$ which excludes match blocking partial orders.
(2)5. $\exists h_b \in \llbracket d_b \rrbracket$, where $t_b = h_1 \sim m \sim h_2$, and where $(ev.h_1 \cup ev.h_2) \cap \phi(ev.d_{pc}) = \emptyset \land ev.m = \phi(ev.d_{pc})$, i.e. $m$ contains all and only the mapped events, while $h_1$ and $h_2$ contains no mapped events.
   Proof: (2)4 and that the trace set $\llbracket d_b \rrbracket$ includes all possible permutations that fulfill the $po$ relation.
(2)6. $\exists h_b \in \llbracket d_b \rrbracket$, where $t_b = h_1 \sim \phi(t_{pc}) \sim h_2$
   Proof: (2)2, (2)5, and that the trace set $\llbracket d_b \rrbracket$ includes all possible permutations that fulfill the $po$ relation.
(2)7. Q.E.D.
   Proof: (2)6 and definitions of $\text{match}$ and $\text{match}$.
(1)2. Assume: $\text{match}(d_{pc}, d_b)$
   Proof: $\text{match}(d_{pc}, d_b)$
(2)1. $\exists l_{pc} \in \llbracket d_{pc} \rrbracket, t_b \in \llbracket d_b \rrbracket : \text{substr}(\phi(t_{pc}), t_b)$

49
Proof: Assumption (1)2 and definitions of $\text{smatch}$ and $\text{match}$.

\[ \forall l \in (L) : \text{substr}(\phi(t_{pc} \uparrow l), t_b \uparrow l) \]
Proof: (2)1 and how $\text{substr}$ works.

\[ \forall l \in L : d_{pc}[l] = t_{pc} \uparrow l \]
Proof: All diagrams are basic sequence diagrams and traces must preserve the event orders per lifeline.

\[ \forall l \in (L) : \text{substr}(\phi(d_{pc}[l]), d_b[l]), \text{i.e. } \text{cmatch}_\phi(d_{pc}, d_b) \]
Proof: (2)2 and (2)3

\[ \text{Assume: There exists a match blocking partial order, i.e. } \exists a, b \in (ev.d_b \setminus \phi(ev.d_{pc})), e_1, e_2 \in \phi(ev.d_{pc}) : (e_1, a), (a, b), (b, e_2) \in d_b.po \]
Proof: \( \bot \)
\[ (3)1. \text{Any base trace, } t_b \in [\! [ d_b ]\!], \text{contains the event sequence: } e_1 \sim h_1 \sim a \sim h_2 \sim b \sim h_3 \sim e_2, \]
for some event sequences $h_1$, $h_2$ and $h_3$.
Proof: (2)5 and how traces adhere to the $po$ relation.

\[ (3)2. \text{Any pointcut trace, } t_{pc} \in [\! [ d_{pc} ]\!], \text{contains the event sequence: } e_1 \sim g \sim e_2, \]
for some event sequence $g$ that does not contain the $a$ and $b$ events.
Proof: Assumption (2)5, how traces adhere to the $po$ relation, and how $\phi$ works.

\[ (3)3. \text{If } t_{pc} \in [\! [ d_{pc} ]\!], t_b \in [\! [ d_b ]\!] \text{ such that } \text{substr}(t_{pc}, t_b) \]
Proof: (3)1 and (3)2.

(3)4. Q.E.D.
Proof: By contradiction. (3)3 and assumption (1)2.

(2)6. Q.E.D.
Proof: (2)4 and (2)5.

(1)3. Q.E.D.
Proof: (1)1 and (1)2, $\Leftarrow$-rule, and $\Rightarrow$-rule.

\[ \square \]

A.3 Formal proof of Lemma 4 (Lifeline-based match with negative pointcuts equals semantics-based match with negative pointcuts)

Proof:

\[ (1)1. \phi \text{ defines a lifeline-based match between the pointcut and the base diagram if and only if } \phi \text{ defines a semantics-based match between the pointcut and the base diagram, i.e.:} \]
\[ \text{lmatch}_\phi(d_{pc}, d_b) \iff \text{smatch}_\phi(d_{pc}, d_b) \]
Proof: This is lemma 2.

\[ (1)2. \text{Choose arbitrary negative pointcut diagram } d_{npc} \in D_{npc}. \]

\[ (1)3. \text{Choose arbitrary } \phi' \text{ such that } \phi \Rightarrow \phi', \text{i.e. for all messages shared between } d_{pc} \text{ and } d_{npc}, \phi \text{ and } \phi' \text{ maps to the same message in } d_b. \]

\[ (1)4. \phi' \text{ does not define a lifeline-based match between the negative pointcut diagram } d_{npc} \text{ and the base diagram } d_b \text{ if and only if } \phi' \text{ does not define a semantics-based match between the negative pointcut and the base diagram, i.e.:} \]
\[ \text{~lmatch}_{\phi'}(d_{npc}, d_b) \iff \text{~smatch}_{\phi'}(d_{npc}, d_b) \]
Proof: This is the contrapositive of lemma 2.
A.4 Helping lemma for lemma 7 (Independent derivations)

**Lemma 13**

\[
\forall l \in L, e \in \text{Event}, e_1 \in \text{Event}, e_2 \in \text{Event}:
B^{A_1, \phi_1} \rightarrow B_1 \rightarrow B_2 \wedge e_1 = \text{first}(\text{A1.pc}[l]) \wedge e_2 = \text{first}(\text{A2.pc}[l]) \wedge e = \phi_1(e_1) = \phi_2(e_2)
\]

Conclusion:

\[
\text{before}(e, B_2) = \text{before}(e, B) \sim \phi_1(\text{before}(e_1, \text{A1.a})) \sim \phi_2(\text{before}(e_2, \text{A2.a}))
\]

**Proof:**

(1) **Assume:** For arbitrary \( l \in L, e \in \text{Event}, e_1 \in \text{Event}, e_2 \in \text{Event}:

1. \( B^{A_1, \phi_1} \rightarrow B_1 \)
2. \( B_1 \rightarrow B_2 \)
3. \( e_1 = \text{first}(\text{A1.pc}[l]) \)
4. \( e_2 = \text{first}(\text{A2.pc}[l]) \)
5. \( e = \phi_1(e_1) = \phi_2(e_2) \)

**Prove:**

\[
\text{before}(e, B_2) = \text{before}(e, B) \sim \phi_1(\text{before}(e_1, \text{A1.a})) \sim \phi_2(\text{before}(e_2, \text{A2.a}))
\]

(2) **Choose** \( h_1, h_2 \in \text{Event}^* \) such that

\[
B[l] = h_1 \sim \phi_1((\text{A1.pc}[l]) \sim h_2 \text{ and } B_1[l] = h_1 \sim \phi_1((\text{A1.a}[l]) \sim h_2
\]

**Proof:** (1):1:1 and definition 7 of a direct derivation.

(2) **2.** \( e \in \phi_1((\text{A1.a}[l]) \)

(3) **1.** \( e \in \phi_1((\text{A1.pc}[l]) \)

**Proof:** (1):1:3 and (1):1:5.

(3) **2.** \( e \in B_1[l] \)

**Proof:** By (1):1:4 and (1):1:5, \( e \) must be in any base diagram using matching the pointcut of A2 (with respect to \( \phi_2 \)). By (1):1:2 and definition 7 of a direct derivation.
derivation, \( \phi_2 \) defines a match between \( B_1 \) and the pointcut of \( A_2 \). Hence, \( e \) must be an event in \( B_1 \). By \( \langle 1 \rangle 1, l \) is the lifeline of \( e \), giving that \( e \) is an event on \( B_1[l] \).

\( \langle 3 \rangle 3. \ e \notin h_1 \land e \notin h_2 \)

**Proof:** (2)1 \( (B[l] = h_1 \sim \phi_1((A_1,pc)[l]) \sim h_2) \), (3)1 and uniqueness of events.

\( \langle 3 \rangle 4. \ e \in \phi_1((A_1,pc)[l]) \)

**Proof:** (2)1 \( (B_1[l] = h_1 \sim \phi_1((A_1,pc)[l]) \sim h_2) \), (3)2 and (3)3.

\( \langle 3 \rangle 5. \text{ Q.E.D.} \)

(2)3. Choose \( h_3, h_4 \in \text{Event}^* \) such that \( \phi_1((A_1,pc)[l]) = h_3 \sim h_4 \) and \( \text{first}(h_4) = e \)

**Proof:** (2)2.

(2)4. \( B_1[l] = h_1 \sim h_3 \sim h_4 \sim h_2 \)

**Proof:** (2)1 and (2)3.

(2)5. Choose \( h'_3, h'_2 \in \text{Event}^* \) such that
\( B_1[l] = h'_3 \sim \phi_2((A_2,pc)[l]) \sim h'_2 \) and
\( B_2[l] = h'_3 \sim \phi_2((A_2,pc)[l]) \sim h'_2 \)

**Proof:** (1)1:2 and definition 7 of a direct derivation.

(2)6. \( h_1 \sim h_3 = h'_1 \)

**Proof:** (2)4 and (2)5, as the two expressions for \( B_1[l] \) must be equal, together with (2)3 \( \text{first}(h_4) = e \), (1)1:4 and (1)1:5.

(2)7. \( B_2[l] = h_1 \sim h_3 \sim \phi_2((A_2,pc)[l]) \sim h'_2 \)

**Proof:** (2)5 and (2)6.

(2)8. \( e \in \phi_2((A_2,pc)[l]) \)

**Proof:** Symmetrical to the proof of (2)2.

(2)9. Q.E.D.

**Proof:** By (2)7 and (2)8, \( \text{before}(e, B_2) = h_1 \sim h_3 \sim \text{before}(e, \phi_2((A_2,pc)[l])) \). By (2)1 and (1)1, \( h_1 \) is the same as \( \text{before}(e, B) \).

By (2)3, \( h_3 \) is the same as \( \text{before}(e, \phi_1((A_1,pc)[l])) \), which by definition of \( \text{before} \) and \( \phi_1 \) equals \( \phi_1(\text{before}(e_1, A_1, a)) \). Similarly,
\( \text{before}(e, \phi_2((A_2,pc)[l])) \) equals \( \phi_2(\text{before}(e_2, A_2, a)) \).

(1)2. Q.E.D.

**Proof:** \( \forall \)-rule and \( \Rightarrow \)-rule.

\( \square \)

### A.5 Proof of lemma 7 (Independent derivations)

The proof consists of four parts. First, it is proven that the three criteria follow from independence. Then, it is proven that the three criteria together ensure independence.

For simplicity, we use \( x = 1 \) and \( y = 2 \) in the proofs, but the proofs are equally valid for \( x = 2 \) and \( y = 1 \).

(1)1. **Assume:** \( B_1^{A_1, a_1} \xrightarrow{\mathcal{A}_1} B \xrightarrow{\mathcal{A}_2, a_2} B_2 \) are independent

**Prove:** \( \forall a, b \in \text{ev}((A_1,pc)) : (\phi_1(a), \phi_1(b)) \in B.dpo \Rightarrow (\phi_2(a), \phi_2(b)) \in B_2.dpo \)

(1)2. **Assume:** \( (\phi_1(a), \phi_1(b)) \in B.dpo \) for arbitrary \( a, b \in \text{ev}((A_1,pc)) \)

**Prove:** \( (\phi_2(a), \phi_2(b)) \in B_2.dpo \)

(3)1. Choose \( B_{\text{join}} \) such that \( B_1 \xrightarrow{\mathcal{A}_2, a_2} B_{\text{join}} \xleftarrow{\mathcal{A}_1, a_1} B_2 \)
Confluence of Aspects for Sequence Diagrams

Proof: (1)1 and definition 12 of independence.

(3)2. $\text{smatch}_\emptyset(A1, pc, B)$

Proof: (1)1 ($B \xrightarrow{A1, \phi_1} B_1$) and definition 7 of a direct derivation.

(3)3. $(a, b) \in (A1, pc).dpo$

Proof: (2)1, (3)2 and lemma 12.

(3)4. $\text{smatch}_\emptyset(A1, pc, B_2)$

Proof: (3)1 ($B_2 \xrightarrow{A1, \phi_1} B_{\text{join}}$) and definition 7 of a direct derivation.

(3)5. $(\phi_1(a), \phi_1(b)) \in B_2, dpo$

Proof: (3)3, (3)4 and lemma 12.

(3)6. Q.E.D.

(2)2. Q.E.D.

Proof: $\forall$-rule and $\Rightarrow$-rule.

(1)2. Q.E.D.

(1)1. Assume: $B_1 \xrightarrow{A1, \phi_1} B \xrightarrow{A1, \phi_2} B_2$ are independent

Proof: $\forall a, b \in \text{ev}(B_1) : \neg \text{blocking}_\phi(a, b, B_1)$

(2)1. Choose $B_{\text{join}}$ such that $B_1 \xrightarrow{A2, \phi_2} B_{\text{join}} \xleftarrow{A1, \phi_1} B_2$

Proof: (1)1 and definition 12 of independence.

(2)2. $\text{smatch}_\emptyset(A2, pc, B_1)$

Proof: (2)1 ($B_1 \xrightarrow{A2, \phi_2} B_{\text{join}}$) and definition 7 of a direct derivation.

(2)3. $\text{lmatch}_\emptyset(A2, pc, B_1)$

Proof: (2)2 and lemma 2.

(2)4. $\forall a, b \in \text{ev}(B_1) : \neg \text{blocking}_\phi(a, b, B_1)$

Proof: (2)3 and definition 5 of $\text{lmatch}$.

(2)5. Q.E.D.

(1)2. Q.E.D.

\[\square\]

The following proof proves criterion 3 for the first/before-case. The end/after-case is proved in a symmetrical manner.

(1)1. Assume: $B_1 \xrightarrow{A1, \phi_1} B \xrightarrow{A2, \phi_2} B_2$ are independent

Proof: $\forall l \in \mathcal{L}: e_1 \in \text{Event}, e_2 \in \text{Event}:

\begin{align*}
  e_1 &= \text{first}((A1, pc)[l]) \land e_2 = \text{first}((A2, pc)[l]) \land \phi_1(e_1) = \phi_2(e_2) \Rightarrow \\
  (\text{before}(e_1, A1, a) = \emptyset \lor \text{before}(e_2, A2, a) = \emptyset \lor \phi_1(\text{before}(e_1, A1, a)) = \\
  \phi_2(\text{before}(e_2, A2, a)))
\end{align*}

(2)1. Assume: $e_1 = \text{first}((A1, pc)[l]) \land e_2 = \text{first}((A2, pc)[l]) \land \phi_1(e_1) = \phi_2(e_2)$ for arbitrary $l \in \mathcal{L}: e_1 \in \text{Event}, e_2 \in \text{Event}

Proof: $\text{before}(e_1, A1, a) = \emptyset \lor \text{before}(e_2, A2, a) = \emptyset$

\begin{align*}
  \lor \phi_1(\text{before}(e_1, A1, a)) = \phi_2(\text{before}(e_2, A2, a))
\end{align*}

(3)1. Let: $e = \phi_1(e_1) = \phi_2(e_2)$, with $l$ the lifeline of $e$.

Proof: (2)1.

(3)2. Choose $B_{\text{join}}$ such that $B_1 \xrightarrow{A2, \phi_2} B_{\text{join}} \xleftarrow{A1, \phi_1} B_2$

Proof: (1)1 and definition 12 of independence.

(3)3. $\text{before}(e, B_{\text{join}}) = $
\begin{align*}
\text{before}(e, B) & \sim \phi_1(\text{before}(e_1, A1.a)) \\
& \sim \phi_2(\text{before}(e_2, A2.a))
\end{align*}

Proof: (1)1, (2)1, (3)1, (3)2 and lemma 13 (with \( B_2 = B_{\text{join}} \)).

\begin{align*}
\langle 1 \rangle & \langle 2 \rangle & \langle 3 \rangle & \langle 4 \rangle & \langle 5 \rangle & \langle 6 \rangle \\
\text{before}(e, B_{\text{join}}) & = & \text{before}(e, B) & \sim \phi_2(\text{before}(e_2, A1.a)) \\
& & & \sim \phi_1(\text{before}(e_1, A2.a))
\end{align*}

Proof: (1)1, (2)1, (3)1, (3)2 and lemma 13 (with \( B_1 = B_2 \) and \( B_2 = B_{\text{join}} \)).

\begin{align*}
\langle 3 \rangle 5. & \text{before}(e_1, A1.a) = \emptyset \lor \text{before}(e_2, A2.a) = \emptyset \\
& \phi_1(\text{before}(e_1, A1.a)) = \phi_2(\text{before}(e_2, A2.a))
\end{align*}

Proof: (3)3 and (3)4. (The two expressions for \( \text{before}(e, B_{\text{join}}) \) must be equal, which can only be the case if the two sub-expressions \( \phi_1(\text{before}(e_1, A1.a)) \) and \( \phi_2(\text{before}(e_2, A2.a)) \) are equal, or at least one of them is empty.)

\begin{align*}
\langle 4 \rangle 2. & \text{For } x \in \{1, 2\}, (\phi_x(\text{before}(e_1, A.x.a)) = \emptyset) \iff (\text{before}(e_x, A.x.a) = \emptyset) \\
& \text{Proof: By definition of } \phi. \text{ (The mapping results in an empty sequence of base events if and only if we start with an empty sequence of aspect events.)}
\end{align*}

\begin{align*}
\langle 4 \rangle 3. & \text{Q.E.D.} \\
& \text{Proof: } (4)1 \text{ and } (4)2.
\end{align*}

\begin{align*}
\langle 3 \rangle 6. & \text{Q.E.D.} \\
\langle 2 \rangle 2. & \text{Q.E.D.} \\
& \text{Proof: } \forall \text{-rule and } \Rightarrow \text{-rule.
}\end{align*}

\begin{align*}
\langle 1 \rangle 1. & \text{Assume: 1. None of the two derivations deletes a direct partial order which is} \\
& \text{part of the other derivation’s match, i.e.:} \\
& \forall a, b \in ev.(A.x.pc) : \\
& (\phi_x(a), \phi_x(b)) \in B.dpo \Rightarrow \\
& (\phi_x(a), \phi_x(b)) \in B_x.dpo
\end{align*}

where \((x, y) \in \{(1, 2), (2, 1)\}\).

\begin{align*}
2. & \text{None of the two derivations produces a match blocking partial order} \\
& \text{for the other derivation’s match, i.e.:} \\
& \forall a, b \in ev.B_x : \sim \text{\textit{blocking}}_\phi(a, b, B_x)
\end{align*}

where \((x, y) \in \{(1, 2), (2, 1)\}\).

\begin{align*}
3. & \text{If there is a lifeline where the matches of the two derivations start} \\
& \text{(end) with the same event, the two derivations cannot add unequal} \\
& \text{event sequences before (after) that event, i.e.:} \\
& \forall l \in \mathcal{L}, e_1 \in \text{Event}, e_2 \in \text{Event} : \\
& e_1 = \text{first}(A1.pc)(l) \land e_2 = \text{first}(A2.pc)(l) \land \\
& \phi_1(e_1) = \phi_2(e_2) \\
& \Rightarrow \\
& (\text{before}(e_1, A1.a) = \emptyset) \\
& \lor \text{before}(e_2, A2.a) = \emptyset \\
& \lor \phi_1(\text{before}(e_1, A1.a)) = \phi_2(\text{before}(e_2, A2.a))
\end{align*}
\[ \forall l \in \mathcal{L}, e_1 \in \text{Event}, e_2 \in \text{Event} : \\
\left( e_1 = \text{last}(\langle A1, pc \rangle[l]) \wedge e_2 = \text{last}(\langle A2, pc \rangle[l]) \wedge \phi_1(e_1) = \phi_2(e_2) \right) \Rightarrow \\
\left( \text{after}(e_1, A1.a) = 0 \right) \\
\vee \left( \text{after}(e_1, A2.a) = 0 \right) \\
\vee \left( \phi_1(\text{after}(e_1, A1.a)) = \phi_2(\text{after}(e_2, A2.a)) \right) \]

**Prove:** \( B_1 \xrightarrow{A1, \phi_1} B \xrightarrow{A2, \phi_2} B_2 \) are independent

(2.1) **Let:** \( l \) be an arbitrary lifeline in \( \mathcal{L} \).

(2.2) **Choose** \( B_{1,2} \) such that \( B_1 \xrightarrow{A2, \phi_2} B_{1,2} \)

(3.1) submodule(\( \phi_2(\langle A2, pc \rangle[l]), B_1[l] \))

(4.1) \( \forall a, b \in \text{ev}(A2, pc) : (a, b) \in (A2, pc).dpo \Rightarrow (\phi_2(a), \phi_2(b)) \in B_1.dpo \)

(5.1) smatch_{A2, pc}(A2, pc, B)

Proof: (1)1 \( (B \xrightarrow{A2, \phi_2} B_2) \) and definition 7 of a direct derivation.

(5.2) \( \forall a, b \in \text{ev}(A2, pc) : ((a, b) \in (A2, pc).dpo \Rightarrow (\phi_2(a), \phi_2(b)) \in B.dpo \)

Proof: (5)1 and lemma 12.

(5.3) Q.E.D.

Proof: (5)2 and (1)1.1.

(4.2) For any diagram \( d \):

\( \forall a, b \in \text{ev}(d), \forall l \in \mathcal{L} : \text{substr}(\langle a, b \rangle, d[l]) \Rightarrow (a, b) \in d.dpo \)

Proof: definition of substr(\( t_1, t_2 \)) and dpo.

(4.3) \( \forall a, b \in \text{ev}(A2, pc), \forall l \in \mathcal{L} : \\
\text{substr}(\langle a, b \rangle, (A2, pc)[l]) \\
\Rightarrow \text{substr}(\phi_2(a), \phi_2(b)), B_1[l]) \)

Proof: (4)1 and (4)2.

(4.4) submodule(\( \phi_2(\langle A2, pc \rangle[l]), B_1[l] \))

Proof: By (4)3, any two consecutive events on the same lifeline in the pointcut of \( A2 \), is mapped to two consecutive base events in \( B_1 \)

(note that \( \phi_2(a), \phi_2(b) \) by definition of \( \phi \)). Hence, any consecutive sequence of events on one lifeline in the pointcut of \( A2 \) is mapped to a consequent sequence of events on the same lifeline in \( B_1 \).

(4.5) Q.E.D.

(3.2) submodule_{A2, pc}(A2, pc, B_1)

(4.1) \( \text{cmatch}_{A2, pc}(A2, pc, B_1) \)

(5.1) \( \text{cmatch}_{A2, pc}(A2, pc, B_1) \)

(6.1) \( \forall l \in \mathcal{L} : \text{substr}(\phi_2(\langle A2, pc \rangle[l]), B_1[l]) \)

Proof: (2)1, (3)1 and \( \forall \)-rule.

(6.2) Q.E.D.

Proof: definition 3 of cmatch.

(5.2) \( \forall a, b \in \text{ev}, B_1 : \neg \text{blocking}_{A2, pc}(a, b, B_1) \)

Proof: (1)1:2.

(5.3) Q.E.D.

Proof: (5)1, (5)2 and definition 5 of lmatch.

(4.2) Q.E.D.
Grønmo, Runde and Møller-Pedersen

Proof: Lemma 2.

3. Choose \( h_1, h_2 \in Event^* \) such that
\[
B_1[l] = h_1 \sim \phi_2((A2,pc)(l)) \sim h_2
\]
Proof: (3)1 and definition of \( substr(t_1, t_2) \) from section 3.

4. Let: \( B_{1,2}[l] = h_1 \sim \phi_2(A2.a(l)) \sim h_2 \)

5. \( B_1 \xrightarrow{A^2,\phi_1} B_{1,2} \)
Proof: (2)1, (3)2, (3)3, (3)4, \( \lor \)-rule and definition 7 of a direct derivation.

6. Q.E.D.

2. Choose \( B_{2,1} \) such that \( B_2 \xrightarrow{A^1,\phi_1} B_{2,1} \)
Proof: Symmetrical to the proof of (2)2.

4. \( B_{1,2} = B_{2,1} \)

1. \( B_{1,2}[l] = B_{2,1}[l] \)

1. Case: On the lifeline \( l \), the events in the pointcuts of \( A1 \) and \( A2 \) maps to disjoint set of events in \( B \), i.e.
\[
ev.(\phi_1((A1,pc)(l))) \cap ev.(\phi_2((A2,pc)(l))) = \emptyset
\]

Proof sketch: As the matches are disjoint, the event sequences in \( B_{1,2}[l] \) and \( B_{2,1}[l] \) are trivially equal.

5.1. Choose \( t_1, t_2 \in Event^* \) such that
\[
B[l] = t_1 \sim \phi_1((A1,pc)(l)) \sim t_2
\]
Proof: (1)1 (\( B \xrightarrow{A^1,\phi_1} B_1 \)) and definition 7 of a direct derivation.

5.2. Case: \( substr(\phi_2((A2,pc)(l)), t_1) \)

6.1. Choose \( t_3, t_4 \in Event^* \) such that
\[
B[l] = t_3 \sim \phi_2((A2,pc)(l))
\sim t_4 \sim \phi_1((A1,pc)(l)) \sim t_2
\]
Proof: (5)1 and (5)2.

6.2. \( B_1[l] = t_3 \sim \phi_2((A2,pc)(l))
\sim t_4 \sim \phi_1((A1,a)(l)) \sim t_2
\)
Proof: (6)1, (1)1 (\( B \xrightarrow{A^1,\phi_1} B_1 \)) and definition 7 of a direct derivation.

6.3. \( B_{1,2}[l] = t_3 \sim \phi_3((A2,a)(l))
\sim t_4 \sim \phi_1((A1,a)(l)) \sim t_2
\)
Proof: (6)2, (2)2 (\( B_1 \xrightarrow{A^2,\phi_1} B_{1,2} \)) and definition 7 of a direct derivation.

6.4. \( B_2[l] = t_3 \sim \phi_3((A2,a)(l))
\sim t_4 \sim \phi_1((A1,pc)(l)) \sim t_2
\)
Proof: (6)1, (1)1 (\( B \xrightarrow{A^2,\phi_1} B_2 \)) and definition 7 of a direct derivation.

6.5. \( B_{2,1}[l] = t_3 \sim \phi_3((A2,a)(l))
\sim t_4 \sim \phi_1((A1,a)(l)) \sim t_2
\)
Proof: (6)4, (2)3 (\( B_2 \xrightarrow{A^1,\phi_1} B_{2,1} \)) and definition 7 of a direct derivation.

6.6. Q.E.D.
Proof: (6)3 and (6)5.

5.3. Case: \( substr(\phi_2((A2,pc)(l)), t_1) \)
Proof: Symmetrical to the proof of (5)2.

5.4. Q.E.D.
Proof: (5)2 and (5)3, as the cases are exhaustive by (4)1 and (5)1.
Confluence of Aspects for Sequence Diagrams

(4.2) Case: On the lifeline $l$, the events in the pointcuts of $AI$ and $A2$ maps to at least one common event in $B$, i.e.

\[ \text{ev}(\phi_1([A1.pc]([l]))) \cap \text{ev}(\phi_2([A2.pc]([l]))) \neq \emptyset \]

(5.1) Let: $ct$ be the sequence of $B([l])$-events matched by both pointcuts, i.e.

\[ \text{substr}(ct, B([l])) \land \\
\forall e \in \text{Event} : e \in ct \Rightarrow \\
e \in \text{ev}(\phi_1([A1.pc]([l]))) \cap \text{ev}(\phi_2([A2.pc]([l]))) \]

(5.2) Choose $h_1, h_2 \in \text{Event}^*$ such that

\[ B_{1,2}([l]) = h_1 \sim ct \sim h_2 \]

Proof: (5.1), as (1):1 ensures that the common event sequence $ct$ cannot be changed by any of the aspects.

(5.3) Choose $t_1, t_2 \in \text{Event}^*$ such that

\[ B_{2,1}([l]) = t_1 \sim ct \sim t_2 \]

Proof: Symmetrical to the proof of (5.2).

(5.4) $h_1 = t_1$

(6.1) Case: The first event in $ct$ is the first event on $l$ matched by both $AI$ and $A2$, i.e.

\[ \text{first}(ct) = \phi_1(\text{first}([A1.pc]([l]))) = \phi_2(\text{first}([A2.pc]([l]))) \]

(7.1) Let: $e_1 = \text{first}([A1.pc]([l]))$

\[ e_2 = \text{first}([A2.pc]([l])) \]

\[ e = \text{first}(ct) = \phi_1(e_1) = \phi_2(e_2) \]

Proof: (6.1).

(7.2) $h_1 = \text{before}(e, B)$

\[ \sim \phi_1(\text{before}(e_1, A1.a)) \sim \phi_2(\text{before}(e_2, A2.a)) \]


(7.3) $t_1 = \text{before}(e, B)$

\[ \sim \phi_2(\text{before}(e_2, A2.a)) \sim \phi_1(\text{before}(e_1, A1.a)) \]

Proof: (1):1, (2):3, (5):3, (7):1 and lemma 13 (with $B_1 = B_2$ and $B_2 = B_{2,1}$).

(7.4) $h_1 = t_1$


(7.5) Q.E.D.

(6.2) Case: $A1.pc$ matches at least one event before the first common match event on $l$, while the first event matched by $A2$ is the first event in $ct$, i.e.

\[ \text{first}(ct) \neq \phi_1(\text{first}([A1.pc]([l]))) \land \text{first}(ct) = \phi_2(\text{first}([A2.pc]([l]))) \]

(7.1) Choose $s_1, s_2, s_3, s_4 \in \text{Event}^*$ such that $B([l]) = s_1 \sim s_2 \sim ct \sim s_3 \sim s_4$

and $\phi_1([A1.pc]([l]) = s_2 \sim ct \sim s_3$ (i.e. $s_1$ is the sequence of $B$-events before the $AI$-match, $s_2$ is the first part of the $AI$-match, not matched by $A2$, $s_3$ is the (possibly empty) final part of the $AI$-match (not matched by $A2$), and $s_4$ is the remaining events not matched by $AI$ (but possibly by $A2$)).


(7.2) $s_2 \neq \emptyset$

\(\langle 7 \rangle 3. \) Let: \(e = \text{first}(ct) = \text{first}(\phi_2((A2.pc)|l])\)
\[\begin{align*}
e_1 &= \phi_1^{-1}(e) \\
e_2 &= \phi_2^{-1}(e)
\end{align*}\]
Proof: \((5)1\) and \((6)2.\)
\(\langle 7 \rangle 4. \) \(\text{before}(e, B) = s_1 \sim \phi_1(\text{before}(e, A1.pc))\)
Proof: \((7)1\) and definition of \(\text{before}\)
\(\langle 7 \rangle 5. \) \(\text{before}(e, B_1) = s_1 \sim \phi_1(\text{before}(e, A1.a))\)
Proof: \((1)1\) \((B \xrightarrow{A1.a} B_1), \langle 7 \rangle 4, \) definition 7 of a direct derivation and definition of \(\text{before}\).
\(\langle 7 \rangle 6. \) \(\text{before}(e, B_{1,2}) = s_1 \sim \phi_1(\text{before}(e, A1.a))\)
\[\sim \phi_2(\text{before}(e, A2.a))\]
Proof: \((2)2, \langle 7 \rangle 5, \) definition 7 of a direct derivation and definition of \(\text{before}\).
\(\langle 7 \rangle 7. \) \(\text{before}(e, B_{1,2}) = s_1 \sim \phi_1(\text{before}(e, A1.a))\)
Proof: \((7)6\) as \(\phi_2(\text{before}(e, A2.a))\) must be empty by \((6)2, \langle 7 \rangle 1\) and \((1)1:1.\)
\(\langle 7 \rangle 8. \) \(\text{before}(e, B_2) = s_1 \sim s_2\)
\[\sim \phi_2(\text{before}(e, A2.a))\]
Proof: \((1)1\) \((B \xrightarrow{A2.a} B_2), \langle 7 \rangle 4, \) definition 7 of a direct derivation and definition of \(\text{before}\).
\(\langle 7 \rangle 9. \) \(\text{before}(e, B_2) = s_1 \sim s_2\)
Proof: \((7)8\) as \(\phi_2(\text{before}(e, A2.a))\) must be empty by \((6)2, \langle 7 \rangle 1\) and \((1)1:1.\)
\(\langle 7 \rangle 10. \) \(\text{before}(e, B_{2,1}) = s_1 \sim \phi_1(\text{before}(e, A1.a))\)
Proof: \((2)3, \langle 7 \rangle 9, \) definition 7 of a direct derivation and definition of \(\text{before}\).
\(\langle 7 \rangle 11. \) \(h_1 = \text{before}(e, B_{1,2})\)
Proof: \((5)2, \langle 7 \rangle 3\) and definition of \(\text{before}\).
\(\langle 7 \rangle 12. \) \(t_1 = \text{before}(e, B_{2,1})\)
Proof: \((5)3, \langle 7 \rangle 3\) and definition of \(\text{before}\).
\(\langle 7 \rangle 13. \) \(h_1 = h_2\)
Proof: \((7)7, \langle 7 \rangle 10, \langle 7 \rangle 11\) and \(\langle 7 \rangle 12.\)
\(\langle 7 \rangle 14. \) Q.E.D.
\(\langle 6 \rangle 3. \) Case: \(A2.pc\) matches at least one event before the first common match event on \(l,\) while the first event matched by \(AI\) is the first event in \(ct.\) i.e.
\[\text{first}(ct) \neq \phi_2(\text{first}((A2.pc)|l)])\]
\[\sim \text{first}(ct) = \phi_1(\text{first}((A1.pc)|l))\]
Proof: Symmetrical to the proof of \((6)2.\)
\(\langle 6 \rangle 4. \) Q.E.D.
Proof: \((6)1, \langle 6 \rangle 2\) and \(\langle 6 \rangle 3\) as the cases are exhaustive (by \((5)1, \) both \(AI\) and \(A2\) cannot include matched events before the first event in \(ct,\) as \(ct\) should contain all common matched events).
\(\langle 5 \rangle 5. \) \(h_2 = t_2\)
Confluence of Aspects for Sequence Diagrams

**Proof:** Symmetrical to the proof of \( \langle 5 \rangle 4 \).

\( \langle 5 \rangle 6 \). Q.E.D.

**Proof:** \( \langle 5 \rangle 2, \langle 5 \rangle 3, \langle 5 \rangle 4 \) and \( \langle 5 \rangle 5 \).

\( \langle 4 \rangle 3 \). Q.E.D.

**Proof:** \( \langle 4 \rangle 1 \) and \( \langle 4 \rangle 2 \) as the cases are exhaustive.

\( \langle 3 \rangle 2 \). Q.E.D.

**Proof:** \( \langle 2 \rangle 1, \langle 3 \rangle 1 \) and \( V \)-rule.

\( \langle 2 \rangle 5 \). Let: \( B_{\text{join}} = B_{1,2} = B_{2,1} \)

**Proof:** \( \langle 2 \rangle 4 \).

\( \langle 2 \rangle 6 \). Q.E.D.

**Proof:** By definition 12 of independence, as \( \langle 2 \rangle 2, \langle 3 \rangle 2 \) and \( \langle 3 \rangle 5 \) gives \( B_{1, A_2, \phi_2} \)

\[ B_{\text{join}} \overset{A_1, \phi_1}{\leftrightarrow} B_{2} \]

\( \langle 1 \rangle 2 \). Q.E.D.

\( \Box \)

### A.6 Helping lemma for lemma 8 (Independent derivations with negative pointcuts and arbitrary events symbols)

**Lemma 14** The mapping \( \phi \) defines a candidate match between a pointcut diagram \( d_{pc} \) and a base diagram \( d_b \) if and only if there is a direct partial order between two pointcut events in the pointcut diagram exactly when there is a direct partial order between the corresponding mapped events in the base diagram. Formally:

\[
\textit{cmatch}_\phi(d_{pc}, d_b) \Leftrightarrow \\
\forall a, b \in \text{ev}.d_{pc} : ((a, b) \in d_{pc}.dpo \Rightarrow (\phi(a), \phi(b)) \in d_b.dpo)
\]

### A.7 Proof of lemma 8 (Independent derivations with negative pointcuts and arbitrary events symbols)

\( \langle 1 \rangle 1 \). **Assume:** \( B_1 \overset{A_1, \phi_1}{\leftrightarrow} B \overset{A_1, \phi_2}{\rightarrow} B_2 \) are independent

**Prove:** Criteria 4, i.e.

\( \forall d \in A_1.Npc : \forall \phi'_1 : \\
\phi_1 \Rightarrow \phi'_1 \Rightarrow \\
( (\exists a, b \in \text{ev}.B_2 : \text{blocking}_{d,d_{pc}}(a, b, B_2)) \\
\lor \\
(\exists a, b \in \text{ev}.d : (a, b) \in d.dpo \\
\land (\phi'_1(a), \phi'_1(b)) \notin B_2.dpo) )
\)

\( \langle 2 \rangle 1 \). **Assume:** \( \phi_1 \Rightarrow \phi'_1 \)

**Prove:** \( (\exists a, b \in \text{ev}.B_2 : \text{blocking}_{d,d_{pc}}(a, b, B_2)) \\
\lor \\
(\exists a, b \in \text{ev}.d : (a, b) \in d.dpo \\
\land (\phi'_1(a), \phi'_1(b)) \notin B_2.dpo) \\
\) for arbitrary \( \phi'_1, d \in A_1.Npc \)

\( \langle 3 \rangle 1 \). **Choose** \( B_{\text{join}} \) such that \( B_1 \overset{A_2, \phi_2}{\rightarrow} B_{\text{join}} \overset{A_1, \phi_1}{\leftrightarrow} B_2 \)

---

59
Proof: (1) 1 and definition 12 of independence.

(3.2) \text{smatch}_b((A_1 \text{pc}, A_1.N\text{pc}), B_2)

Proof: (3.1) \((B_2 \xrightarrow{A_1.d_1} B_{\text{join}})\) and definition 7 of a direct derivation.

(3.3) \text{¬smatch}_b(d, B_2)

Proof: (2.1), (3.2) and definition 8 of \text{smatch}.

(3.4) \text{¬lmatch}_b(d, B_2)

Proof: (3.3) and lemma 2.

(3.5) \exists \alpha, \beta \in \text{ev}.B_2 : \text{blocking}_{d, \beta_1}(a, b, B_2)

\lor \text{¬cmatch}_d(d, B_2)

Proof: (3.4), definition 5 of \text{lmatch} and first-order logic.

(3.6) Case: \exists \alpha, \beta \in \text{ev}.B_2 : \text{blocking}_{d, \beta_1}(a, b, B_2)

(4.1) Q.E.D.

(3.7) Case: \text{¬cmatch}_d(d, B_2)

(4.1) \exists \alpha, \beta \in \text{ev}.d : ((e_1, e_2) \in d.dpo \land (\phi_1'(e_1), \phi_1'(e_2)) \notin B_2.dpo) \lor

((e_1, e_2) \notin d.dpo \land (\phi_1'(e_1), \phi_1'(e_2)) \in B_2.dpo)

Proof: (3.7) and definition 3 of \text{cmatch}.

(4.2) Case: Choose \(e_1, e_2 \in \text{ev}.d\) such that \((e_1, e_2) \in d.dpo \land (\phi_1'(e_1), \phi_1'(e_2)) \notin B_2.dpo)

(5.1) Q.E.D.

Proof: (4.2) with \(e_1 = a\) and \(e_2 = b\).

(4.3) Case: Choose \(e_1, e_2 \in \text{ev}.d\) such that \((e_1, e_2) \notin d.dpo \land (\phi_1'(e_1), \phi_1'(e_2)) \in B_2.dpo)

(5.1) Case: \(\phi_1'(e_1)\) and \(\phi_1'(e_2)\) are the send and receive event of the same message.

Proof: Impossible, as we would then also have \((e_1, e_2) \in d.dpo\) by definition of \(\phi\) and the message invariant, which contradict the case assumption \((e_1, e_2) \notin d.dpo\).

(5.2) Case: \(\phi_1'(e_1).\ll = \phi_1'(e_2).\ll\)

(6.1) Let \(l = e_1.\ll = e_2.\ll\)

Proof: (5.2) and definition of \(\phi\).

(6.2) Case: \((e_2, e_1) \in d.dpo\)

(7.1) \((\phi_1'(e_2), \phi_1'(e_1)) \notin d_0.dpo\)

Proof: (4.3), (6.2) and definition of \(dpo\).

(7.2) Q.E.D.

Proof: Let \(e_2 = a\) and \(e_1 = b\).

(6.3) Case: \(\exists \alpha \in \text{ev}.d[.l], dpo : (e_1, e_3) \in d.dpo \land (e_3, e_2) \in d.dpo\)

(7.1) \((\phi_1'(e_1), \phi_1'(e_3)) \notin d_0.dpo\)

Proof: (4.3), (6.1) and (6.3) and definition of \(dpo\).

(7.2) Q.E.D.

Proof: Let \(e_1 = a\) and \(e_3 = b\).

(6.4) Case: \(\exists \alpha \in \text{ev}.d[.l], dpo : (e_2, e_1) \in d.dpo \land (e_3, e_1) \in d.dpo\)

(7.1) \((\phi_1'(e_2), \phi_1'(e_1)) \notin d_0.dpo\)

Proof: (4.3), (6.1) and (6.4) and definition of \(dpo\).

(7.2) Q.E.D.
Proof: Let $e_3 = a$ and $e_1 = b$.

(6)5. Q.E.D.

Proof: (6)2, (6)3 and (6)4 as the cases are exhaustive by (4)3 and (6)1 (as $e_1$ and $e_2$ are on the same lifeline, there must be a partial order between them, but not a direct partial order from $e_1$ to $e_2$).

(5)3. Q.E.D.

Proof: (5)1 and (5)2 as the cases are exhaustive by definition of $dpo$.

(4)4. Q.E.D.

Proof: (4)2 and (4)3 as the cases are exhaustive by (4)1.

(3)8. Q.E.D.

Proof: (3)6 and (3)7 as the cases are exhaustive by (3)5.

(2)2. Q.E.D.

Proof: $\forall$-rule and $\Rightarrow$-rule.

(1)2. Q.E.D.

(1)1. Assume: Criteria 1–4 holds.

Proof: Independence.

Proof: The proof is equal to the corresponding proof of lemma 7, with the exception of step 1.2.2 (i.e. step <3>.2 under <2>.2 under <1>.1):

(2)2. Choose $B_{1,2}$ such that $B_1 \xrightarrow{A_2,\phi_1} B_2$

(3)2. $\text{smatch}_{\phi_1}(A_2,\text{pc}, A_2, \text{Npc}, B_1)$

(4)1. $\text{lmatch}_{\phi_1}(A_2, \text{pc}, A_2, \text{Npc}, B_1)$

(5)1. $\text{lmatch}_{\phi_1}(A_2, \text{pc}, B_1)$

Proof: This is sub-proof 1.2.3.1 (<4>1) in the proof of lemma 7.

(5)2. $\forall d \in A_2, \text{Npc} : \forall \phi'_2 : \phi_2 \Rightarrow \phi'_2 \Rightarrow \neg \text{lmatch}_{\phi'_2}(d, B_1)$

(6)1. Assume: $\phi_2 \Rightarrow \phi'_2$

Proof: $\neg \text{lmatch}_{\phi'_2}(d, B_1)$

for arbitrary $\phi'_2$ and $d \in A_2, \text{Npc}$

(7)1. $\neg \text{cmatch}_{\phi'_2}(d, B_1) \lor$

$\neg (\forall a, b \in \text{ev}, B_1 : \neg \text{blocking}_{d, a, b}(a, b, B_1))$

(8)1. Case: $\exists a, b \in \text{ev}, B_1 : \text{blocking}_{d, a, b}(a, b, B_1)$

Proof: First-order logic.

(8)2. Case: $\exists a, b \in \text{ev}, d : (a, b) \in d, \text{dpo} \land$

$(\phi'_2(a), \phi'_2(b)) \notin B_1, \text{dpo}$

(9)1. $\neg \forall a, b \in \text{ev}, d : ((a, b) \in d, \text{dpo}) \Rightarrow$

$(\phi'_2(a), \phi'_2(b)) \in B_1, \text{dpo}$

Proof: (8)2 and first-order logic.

(9)2. Q.E.D.

Proof: $\neg \text{cmatch}_{\phi'_2}(d, B_1)$ follows from lemma 14.

(8)3. Q.E.D.

Proof: The cases are exhaustive by criterion 4 and (6)1.

(7)2. Q.E.D.

Proof: definition 3 of $\text{cmatch}$.
Proof: $\forall$-rule and $\Rightarrow$-rule.

(5)3. Q.E.D.

Proof: (5)1, (5)2 and definition 9 of $lmatch$.

(4)2. Q.E.D.

Proof: Lemma 4.

(1)2. Q.E.D.
Appendix I

Paper 8: From Sequence Diagrams to State Machines
– with help from Combined Fragments
Abstract

The setting for this paper is a UML-based software engineering process, where sequence diagrams are used to model the requirements, and state machines are used to implement the system. There is a great deal of overlap in the information captured by these two diagram types, and the designer needs to be careful when designing the state machines so that they are consistent with the previously defined sequence diagrams. Consistency can be defined by an existing refinement theory. This paper proposes a transformation from sequence diagrams to state machines. The transformation is helpful within our described modeling process, which is based on the refinement theory. We take advantage of the added expressiveness in UML 2 where combined fragments (e.g. conditional behavior, loop) can be used to define more precise sequence diagrams than in previous UML versions. The main contribution of the paper is a set of transformation rules. The rules are based on graph transformation and extended with tailored transformation support for combined fragments.

1 Introduction

UML-based software development processes [1] prescribe the usage of multiple diagram types. These diagram types provide different views of the system, and which diagram types to be used depends on the different phases of the development process. Also, these diagram types are used in different ways, ranging from highly informal idea sketches to formal specifications of a system that can be used to automatically produce test synthesis or executable artefacts of a running system.

UML sequence diagrams are popular for capturing requirements. These diagrams represent example executions of the system which can be understood also by non-technical stakeholders of a system. UML state machines are normally used in a later phase than sequence diagrams to define more precise specifications or even complete specifications of the system.

Although sequence diagrams and state machines are used in different phases and are made with different diagram types, there is a great deal of overlap
between the two specifications. The behavior defined by the sequence diagrams should also be recognized as behavior by the state machines.

In the last decade there has been a lot of efforts to transform from sequence diagram-like specification languages to state-based languages (e.g. [11, 17, 18, 15]). Some of the previous UML-based efforts (e.g. [17]) have been applied before combined fragments were introduced in UML 2, and to our best knowledge none of the previous approaches have linked their transformation approach to a formal refinement theory. With the limited expressiveness in UML 1.x, the generated state-based system would either contain duplicated states, or the modeler would need to tag the sequence diagrams with additional information in a notation which is not part of the sequence diagram notation.

The combined fragments in UML 2 includes possibilities to model optional behavior, conditional behavior and loops, and these can have guard expressions and be arbitrarily nested. This added expressiveness compared to previous UML versions, makes it possible to specify detailed and precise diagrams. We prescribe a modeling process where the transformation from sequence diagrams to state machines is an important part. In order to be successful, the sequence diagrams need to be specified in a precise manner, with sufficient usage of combined fragments, prior to the transformation to state machines.

The set of sequence diagrams is normally a partial specification, which means that the generated state machines should be further detailed and refined to become a complete specification. It is desirable that the manual updates of the state machines are not in conflict with the sequence diagrams, particularly if the sequence diagrams are defined and approved by other stakeholders than those developing the state machines. An existing refinement theory by Runde et al. [13, 14] and tools make it possible to automatically validate if the updated state machines remain consistent with the set of sequence diagrams (e.g. a tool developed by Brændshøi [2]).

The main contribution of the paper is a graphical transformation language with which we can define a transformation from sequence diagrams to state machines. The transformation language is graph transformation-based [3], where the rules use the concrete syntax of sequence diagrams and state machines. This can be challenging since the abstract syntax of sequence diagrams is quite different from the concrete syntax. This makes it interesting to study if and how graph transformation can be applied to sequence diagrams and in particular for the combined fragments.

The remainder of this paper is structured as follows. In Section 2 we introduce sequence diagrams, state machines, traces and the refinement theory; Section 3 describes our proposed modeling process in relation to the refinement theory, and where a transformation from sequence diagrams to state machines is a valuable part; Section 4 describes preliminaries on graph transformation; Section 5 describes how we adapt graph transformation to sequence diagrams; Section 6 presents our graph transformation rules; Section 7 compares our approach with related work; and finally Section 8 concludes the paper.
2 Sequence Diagrams, State Machines and Refinement

As our example, we model the interaction between a user, an automatic gas pump, and a bank to verify the inserted credit card. Figure 1 shows a sequence diagram and a corresponding state machine to represent the behavior of the second lifeline object (GasPump) in the sequence diagram. The sequence diagram has two lifelines with the types 'User' and 'GasPump', and two messages with the signals 'insertCard' and 'requestPin'. A lifeline, visualized with a rectangle and a dashed line below, represents an interacting entity on which events take place in an order from top to bottom on the dashed line. Each message is represented by two events, a send event (at the source of the message arrow) and a receive event (at the target of the message arrow). In this paper we only consider sequence diagrams with asynchronous messages. We omit the optional rectangles to visualize when a lifeline is active, since these are more relevant for synchronous messages.

The state machine has one initial state with a transition leading to the state named 'Idle'. Transitions have the form: trigger [guard] / effect. A trigger corresponds to a receive event, and an effect corresponds to a sequence of send events. Transitions without trigger, guard and effect are called empty transitions, such as the transition from the initial state to the 'Idle' state. The transition from 'Idle' to 'S1' has no explicit guard (implying that the guard is always true), a trigger 'insertCard' and 'requestPin' as its effect. For brevity we use the message names directly as both triggers and effects, even though the latter more precisely could be displayed with send as prefix, e.g. send requestPin.

STAIRS [13] gives the semantics of a sequence diagram using traces that represent possible executions. The semantics of a sequence diagram can be described as a set of positive traces and a set of negative traces. Positive traces define valid behavior and negative traces define invalid behavior, while all other traces are defined as inconclusive. In the sequence diagram of Figure 1, there is exactly one positive trace < send insertCard, receive insertCard, send requestPin, receive requestPin >. Negative traces are described by special operators (e.g. neg), which are not used in the diagram of Figure 1. Hence, all other traces than the single positive trace, are inconclusive.

The leftmost part of Figure 2 shows a graphical notation of the universe of traces, where a rectangle is divided into positive (p), inconclusive (i) and negative (n) traces. In reality there are infinitely many inconclusive traces for the
sequence diagrams, and infinitely many negative traces for the state machines.

Figure 2: Universe of traces and refinement

The rest of Figure 2 shows the three kinds of sequence diagram refinement that are defined by STAIRS [14]:

1. **positive supplementing.** A previously undescribed scenario is described as positive behavior

2. **negative supplementing.** A previously undescribed scenario is described as negative behavior

3. **narrowing.** Some previously described positive behavior is described as negative behavior

In our mapping each lifeline corresponds to a state machine. Send events are prefixed by `!`, and receive events are prefixed by `?`. The set of positive/negative traces can be filtered with respect to a lifeline by removing all events that does not occur on the lifeline.

A state machine generates a language, where all sentences in the language correspond to positive traces. The allowed words are the triggers and effects of the state machine, where triggers are prefixed by `?`, and effects are prefixed by `!`. The path from the source to the target of a transition generates part of a sentence: `<?trigger,!effect>`. A sentence is grammatically correct if it can be generated from a path starting in the initial state and ending in a final state. The language of the `GasPump` state machine has a single possible sentence `⟨?insertCard,!requestPin⟩`, which is identical to the single positive trace of the `GasPump` lifeline.

The set of sequence diagrams describing a system will normally have a non-empty set of inconclusive traces, which we call a partial specification. An actual implementation may choose to implement the inconclusive traces as either positive or negative. A state machine on the other hand, has no inconclusive traces and is thus a complete specification.

Since the set of sequence diagrams is only a partial specification, the automatically produced state machines are only intended to be a good starting point for a manual refinement. This makes it important that the produced state machines are readable.
3 A Modeling Process from Sequence Diagrams to State Machines

In Figure 3 we show our recommended modeling process of five steps, starting with the early phase of simple sequence diagrams and ending with the final state machines that can be used to generate Java code [9]. The artefact of each step is shown in separate rectangles containing example diagrams. For each artefact we also show the universe of traces to illustrate how the relative sizes of the three trace sets (positive, inconclusive, negative) evolve throughout the modeling process.

Step 1. Scenarios can easily be described with intuitive and simple diagrams showing example executions in the to-be-implemented system. These initial sequence diagrams (in step 1 and 2) should not be too detailed and they should use few or no combined fragments, since this could be counterproductive in the idea and brainstorming phase.

The separation between step 1 and step 2 is more to explain the effects of the modeling process, rather than being a natural milestone for the modeler.

Step 2. We use positive and/or negative supplementing to refine the specification of sequence diagrams from step 1. We make multiple diagrams involving the same lifelines, where similar behavior often occurs in several diagrams. Sometimes the similar behavior is accidentally equal, and other times it represents the same system state. The latter example should be merged prior to generation of state machines so that we don’t produce duplicated states. It is impossible to automatically distinguish the unintended similar behavior from...
the actual similar behavior as explained by Whittle and Schumann [17].

**Step 3.** We propose instead, in step 3, that the user takes advantage of the combined fragments which are new in UML 2 sequence diagrams, to manually merge similar behavior from multiple diagrams into a single diagram. The advantage is that this all happens in the context of the well-known sequence diagrams with no need to clutter the sequence diagrams with other expressions, nor a need to master another description language. Another benefit is that there is an existing tool available that can be used to check that the modified sequence diagrams are refinements of the previous sequence diagrams [12].

The combined fragments to merge similar behavior includes: 1) **alt** operator to express the differing behavior inside its operands, while similar behavior occurs prior to or after the operator, and 2) **loop** operator to express repeated behavior where the number of repetitions varies between different execution scenarios. A combined fragment is displayed with a rectangle that spans the involved lifelines, an operator type shown in the top left corner of the rectangle, and dashed lines as operand separators in cases with multiple operands.

In step 3, the modeler should also detail the diagrams such as decomposing a lifeline into a set of lifelines, and by adding guards to combined fragments. The merging of diagrams and decomposition of lifelines will not affect the set of positive or negative traces. Adding guards, on the other hand, changes some traces from positive to negative (narrowing).

Step 3 ensures that each lifeline only occurs in a single diagram. This can always be achieved by using enough combined fragments. For convenience, unrelated scenarios involving the same lifeline can be kept in several diagrams. The manual work in step 3 can be followed by a transformation that merges all lifelines into the same diagram. This transformation can introduce one outermost **alt** operator with one operand for each of the unrelated scenarios. Such a transformation is semantics-preserving with respect to the traces that a set of sequence diagrams represent.

The step 3 artefact represents a *contract* which an implementation must fulfill. We interpret all the positive traces as mandatory behavior which must be implemented, while the negative traces describe prohibited behavior. A state machine that fulfills the contract must therefore have sentences corresponding to all the positive traces and no sentences corresponding to the negative traces, while we optionally may provide sentences corresponding to inconclusive traces.

**Step 4.** Our automated generation **sd2sm**, in step 4, makes a state machine where a sentence is grammatically correct if and only if the corresponding trace is positive. This means that by default all the inconclusive traces are not implemented, and these traces become negative. Hence, step 4 performs a negative supplementing.

**Step 5.** In step 5, the modeler refines the generated state machines so that they are detailed enough to express a full implementation. Furthermore, the modeler may also freely increase the number of implemented traces, but restricted to those that are inconclusive in the contract (positive supplementing). All modification of the state machines should be checked to see if they represent a breach of contract. A breach of contract should be reported to the modeler by
highlighting the sequence diagram that is no longer supported. Either the mod-

er should undo the last state machine modification, or the sequence diagrams

should be updated. Brændshøi has implemented an automated tool that checks

if a state machine is a ‘proper implementation’ of a set of sequence diagrams

[2].

The rest of the paper describes the automated transformation of step 4 from

sequence diagrams to state machines. In the next section we introduce the graph

transformation-based rule language.

4 Concrete Syntax-based Graph Transformation

The graph concept [3] is based on nodes and directed edges in which we can

represent most of today’s diagram types. A diagram type has a metamodel

where each type in the metamodel is represented as a graph node, and graph

edges represent relationships between the nodes.

A model transformation can then be defined by a set of graph transformation

rules. A graph transformation rule consists of exactly one left hand side graph

(LHS), a (possibly empty) set of negative application condition graphs (NACs),

and exactly one right hand side graph (RHS). The LHS defines a subgraph for

which we are looking for matches within the graph to be transformed. A NAC

prevents application of a rule if the LHS combined with the NAC has a match.

None of the NACs can have a match in order to apply a rule. When a rule is

matched by a LHS, then the matched LHS within the source graph is replaced

by the RHS of the matched rule.

Graphs have a predefined visual representation, called the abstract syntax.
The abstract syntax visualizes all nodes in a similar way, and all edges in a

similar way. Typically, a node is visualized with a rectangle separated into two

compartments. The first compartment denotes the instance identifier and node

type, while the second compartment contains a list of attribute values. An edge

is normally visualized with an arrow, where the edge type is placed next to the

arrow.

An identifier, displayed next to a graph element, denotes a shared element

between the LHS and the NACs/RHS. Elements shared between the LHS and

the RHS are preserved by the rule, while the other elements in the LHS are

deleted, and the other elements in the RHS are added. We follow the widely

used principle (known as double pushout), where the dangling condition [7]

ensures that a rule, involving node deletion, is only applied when there will be

no dangling edges in the resulting graph.

The concrete syntax of a diagram type uses a tailored visualization with

icons and rendering rules depending on the element types. To improve the

usability for the graph transformation designer, we define the transformation

rules upon concrete syntax and refer to this approach as concrete syntax-based

graph transformation (CGT).

When applying graph transformation at the concrete syntax of a diagram

type, the basic principle is to define a mapping between the concrete and ab-
abstract syntax of a diagram type. Then, both the rules and the models at the concrete syntax are translated into abstract syntax, the rules are applied, and finally the result is translated back to concrete syntax. Our experience is that this principle is directly suitable for many diagram types including state machines [4].

We have previously introduced a collection operator [4] that can be used in a graph transformation rule to match and transform a set of similar subgraphs. This makes it possible to use a single rule where several rules were necessary without the collection operator. The collection operator is illustrated in the transformation rules presented in Section 6.

5 Applying CGT to Sequence Diagrams

Since sequence diagrams are quite different from graphs and other diagram types, we need a specialized CGT treatment. While graphs have no ordering among its incident edges, the events on a lifeline are ordered. Sequence diagrams are properly supported in CGT by replacing each event sequence in the LHS by the RHS event sequence per lifeline.

Figure 4 shows our choice of abstract syntax for sequence diagrams. The event sequence of a lifeline is represented by a linked list of Event typed nodes. We use special marker event nodes to represent the start (kind="first") and end of the event list (kind="last"). In addition each event node has an ll-typed edge back to the lifeline on which the event occurs. These choices allow us to relatively easily map the CGT rules into abstract syntax rules. As we

Figure 4: CGT for sequence diagrams
see in Figure 4 such a rule matches event sequences anywhere on a lifeline and replaces the matched event sequence by the RHS event sequence, as desired.

For combined fragments we define that the LHS sequence diagram matches inside a combined fragment, but LHS structures cannot cross the combined fragment borders or cross an operand separator.

The next two subsections introduce transformation mechanisms tailored for matching and transforming combined fragments. These two mechanisms apply to any type of combined fragment, and the latter mechanism is also applicable to state regions for UML state machines.

### 5.1 Fragment projection

In our transformation \texttt{sd2sm}, only a single lifeline is relevant for each generated state machine. This fact can be directly exploited with a new mechanism, called \textit{fragment projection}, that can project a combined fragment onto a single lifeline. The fragment projection in general filters a matched combined fragment onto a proper subset of its lifelines such that an event is only kept if both its receiver and sender lifelines are part of the combined fragment in the RHS. If the combined fragment to be projected contains nested combined fragments, then these are preserved, except for having a subset of the original events.

Figure 5 shows the effects of applying a rule with fragment projection from three lifelines onto two lifelines. The x and z messages are deleted, since they involve the third lifeline which no longer takes part in the combined fragment. Fragment projection is used in the rule \texttt{SplitFragment}, which is introduced in Section 6.

![Fragment projection diagram](image)

**Figure 5:** Applying a rule involving fragment projection

### 5.2 Compartment Operator: Fragment Operands and State Regions

A combined fragment with operator \texttt{opt}, \texttt{loop}, \texttt{break} or \texttt{neg} contains exactly one operand, while for other operators (e.g. \texttt{alt}, \texttt{par}) it contains an arbitrary
number of operands. In some rules there is a need to express the matching of a single operand, or arbitrary many operands in the LHS, and to sometimes keep only the operand parts in the RHS of a rule.

In a concrete syntax rule, it is not straightforward how to distinguish between the combined fragment operator itself and its operands. A similar challenge applies to state regions of state machines, which are also displayed in separate compartments of a state. We introduce the compartment operator as a new graphical element to be used in CGT rules. It is displayed as a rectangle with a label depending on the model construct (‘operand’/’region’ for combined fragments/state regions), and it is placed inside its parent (fragment operator/state). Since this operator has a clear border between the compartment content and its parent, it is well suited to use in CGT rules.

Multiple operands/regions may be expressed by explicitly drawing several compartment operators, or by placing a collection operator around a compartment operator. Figure 6 shows a rule that matches all the operands of an alt fragment involving the two lifelines of type $L_1$ and $L_2$. The result of applying the rule on an alt fragment with two operands is that the alt fragment is removed, and that we get two copies of the $L_1$ and $L_2$ lifelines, one copy for each alt operand. Events before or after the alt fragment on the $L_1$ and $L_2$ lifelines will be preserved in both the copies, e.g. the send and receive events of the $x$ message.

![Figure 6: Applying a rule with compartment operator](image)

Compartment operators are also used in the rules Alt, Loop and Par, which are introduced in Section 6.

6 Transformation Rules

In this section we present the transformation rules, and we show how the rules gradually transform from a sequence diagram into state machines.
We use the term source model for the model to be transformed, and target model for the final result of the transformation. The source model in Figure 7 (labeled 1) is a sequence diagram for a gas pump scenario. A user inserts a payment card (insertCard). The gas pump requests the pin code from the user (requestPin) and the user enters the pin code (pinCode). A bank validates the pin code (validate and result), and an alt operator models the two possible outcomes: 1) valid pin code: The user is informed to start fuel (startFuel) and the user indicates end of fueling by hanging up the gas pump (hangUp), or 2) invalid pin code: The user is informed that the entered pin code is invalid (invalidPin). In both cases, the scenario ends by ejecting the card (cardOut).

In the transformation sd2sm, each lifeline corresponds to a state machine. When producing a state machine, it is sufficient to look at the single corresponding lifeline with its events and how these events are structured within the combined fragments. A prerequisite to this claim is that each lifeline occurs only in one sequence diagram, which is ensured by introducing the combined fragments in step 2 of the method described in Section 3.

The intermediate models in the transformation process contains sequence diagrams, state machines and helper edges with type name state to link each lifeline to its state machine.

The transformation rules are grouped in five layers (0 to 4). The idea of organizing rules into different layers is taken from the AGG tool [16]. A layer has an index, and all rules in the layer with the lowest index are applied nondeterministically within the layer and as long as possible, i.e., until no more rules are applicable in the layer. Then the application of rules proceeds to the next layer, and we may also loop over the layers when the layer with the highest
Layers 0 and 1. A sequence diagram with $n$ lifelines are, by the rules in layers 0 and 1, replaced by $n$ separate sequence diagrams, each with a single lifeline and where all the original lifelines occur once. In these new sequence diagrams the original combined fragment structure is preserved, while only the events concerning the single lifeline are kept.

In the transformation process we want to treat each lifeline in isolation. The lifeline events shall be mapped to triggers and effects in the corresponding state machine, and mapped events are to be removed. However, an event is always part of a message with another event. Thus, removing an event of a message is not possible without also removing the entire message including the other event of the message which normally belongs to another lifeline.

Our transformation encodes a single message by four reflexive messages, i.e. messages where the sender and receiver lifelines are the same. For simplicity we assume that send and receive are not used as original message names, and that these pseudomessages mean that the following message is either a send or a receive event respectively. The rule SplitMessage (Figure 7) encodes all the original non-reflexive messages, and the rule SplitSelfMessage (not shown) encodes all the original reflexive messages. These two rules constitute layer 0.

When none of the two rules from layer 0 can be applied, all remaining messages are reflexive, and we can safely use the fragment projection mechanism in the rule SplitFragment (Figure 7) of layer 1 to copy and project the fragment operators onto each lifeline.

The collection operator expresses that there can be an arbitrary number of lifelines, which ensures that any combined fragment operator is matched. In the RHS the collection operator also includes the fragment operator, which means that the fragment operator will be copied onto each lifeline. The fragment projection mechanism ensures that only events concerning the respective lifelines are kept inside the fragment operator. The model labeled 2 in Figure 7 shows the lifeline GasPump after the rules in layers 0 and 1 are finished. The SplitMessage rule is applied nine times followed by one application of the rule SplitFragment.

Layer 2. Layer 2 consists of three rules that prepares each lifeline to be transformed into a state machine (Figure 8). The rule InitSM adds a new state machine with an initial state with an empty transition leading to the Idle state. Furthermore, it adds a helper edge of type state (pointing to the current state) from the lifeline to the Idle state.

The rule InitLifelineMessageOnTop adds a reflexive message with name top to be the very first message on the lifeline. Again we assume for simplicity that top is not used as a message name in the source model.

A similar rule InitLifelineFragmentOnTop initializes the lifeline when a combined fragment, and not a message event, is the first occurrence on a lifeline. The NACs ensure that the InitSM rule is applied exactly once, and that one of the two init lifeline rules are applied exactly once, on each lifeline. The model labeled 3 in Figure 8 shows the result after the rules in layer 2 are finished.

Layer 3. The rules in layer 3 pops the top-most 'occurrence' on a lifeline and adds corresponding behavior to the state machine which belongs to the life-
Figure 8: GasPump: From SD to SM. Intermediate model (labeled 3) and three rules (layer 2): InitSM, InitLifelineMessageOnTop, InitLifelineFragmentOnTop

A top-most 'occurrence' is either a combined fragment or it is an encoded event (send+message or receive+message). The rule Send (Figure 9) pops an encoded send event (two reflexive messages) and adds a corresponding effect on the incoming transition to the current state.

Figure 9: GasPump: From SD to SM. Intermediate model (labeled 4) and two rules (layer 3): Receive and Send

The rule Receive (Figure 9) pops an encoded receive event (two reflexive
messages), adds a state which now becomes the current state, and adds a transition with trigger labeled by the receive message name. The transition goes from the previous current state to the new current state.

The model labeled 4 in Figure 9 shows the result after applying the rule sequence <Receive, Send, Receive, Send, Receive>.

The rule Alt (Figure 10) pops an alt fragment, makes the current state into a composite state by adding internal behavior: initial, Idle and final states, an inner composite state for each alt operand.

We produce a transition from the Idle state to each inner composite state, where the transition guard is equal to the corresponding alt operand guard. For each alt operand we also produce a new lifeline with the alt operand content and where the lifeline has a current state edge to the Idle state of the inner composite state. Finally the original lifeline where we popped the alt operator, gets a new state as its current state, and the old current state gets a transition leading to the new current state. The model labeled 5 in Figure 10 shows the result after applying the Alt rule.

The rule FinalState (Figure 10) deletes a sequence diagram with a top message, and its current state is replaced by a final state. Due to the dangling condition, such a deletion is only allowed when the sequence diagram has no other connecting messages than the top message. The model labeled 6 in
Figure 10 shows the result after applying the rule sequence \(<\text{Send}^3, \text{Receive}, \text{FinalState}^3\)>, where an exponent indicates multiple applications of the same rule.

The rule Loop (Figure 11) pops a loop fragment, makes the current state into a composite state, and adds a reflexive transition for the composite state with a guard equal to the loop fragment guard. The lifeline from which we popped the loop fragment gets a new state as its current state, and the old current state gets a transition leading to the new current state. In addition the rule produces a new lifeline for the loop content with the Idle state of the inner composite state as its current state.

![Transformation rule (layer 3): Loop](image)

The rule Par (in Figure 12) pops a par fragment and makes the current state into a composite state with one region for each par operand to represent the concurrent behavior. The rule Opt (in Figure 12) for an opt fragment is a special case of the alt operator, where we introduce a new composite state with two inner branches. One branch which goes straight to the final state in case the guard from the opt operator is false, and another branch for the body of the opt operator when the guard from the opt operator is true. The rule Neg simply removes a neg operator without any effects on the state machine since only positive behavior influences our initial state machines.

Layer 4. Figure 13 shows the target state machine model (still only corresponding to the GasPump lifeline). We have applied three flattening rules of layer 4 to produce a more readable and concise state machine. The flattening rules collapse composite states that have been produced by the combined fragment rules (e.g. the Alt and Loop rules). Figure 13 shows only the three flattening rules that is used to collapse composite states produced by the Alt rule. A few additional flattening rules, not shown, are needed for the other combined fragment rules.

The FlattenIntoChoice rule starts the flattening process of the state machine produced for an alt operator. The rule removes the composite state and its initial, idle and final state. Furthermore, it introduces a choice and a merge with the same guarded branches as there were within the composite state. The FlattenSubState1 and FlattenSubState2 rules collapses the composite states within each of the guarded branches between the choice and the merge. We need two rules to handle two different cases. The first case is when the composite state has no inner states except the initial and final states. The other case

15

From Sequence Diagrams to State Machines
is when there are additional inner states.

Since the transformation produces one state machines per lifeline, these state machines should be combined to one state machine. Each state machine in this combined state machine is placed in a separate region. This means that all the state machines are started in parallel.

The transformation rules are implemented in the graph transformation tool AGG. The transformation is tested with success on some examples, including the GasPump example shown in this paper. The AGG tool only supports abstract syntax rules, and we have manually translated from concrete syntax to abstract syntax rules. In future work it is desirable to automate the translation from concrete syntax rules to abstract syntax rules for sequence diagrams and state machines, as we have demonstrated previously for activity models [5].

Figure 14 shows the AGG rule for Receive. Reflexive messages are represented with only one event on the lifeline to get more concise graphs. Notice also that the LHS of the rule needs to match the parent of the current state of the lifeline. Then this parent can be set as the parent of the new state in the RHS of the rule. The rest of the translation from concrete to abstract syntax is straightforward and follows the principles from Figure 4.
From Sequence Diagrams to State Machines

Figure 13: GasPump: From SD to SM. The final resulting state machine corresponding to the GasPump lifeline

Figure 14: The transformation rule `Receive` as represented by an abstract syntax rule in AGG
7 Related Work

Our methodology is quite similar to the one prescribed by Whittle and Schumann [17] and Ziadi et al. [18]. Whittle and Schumann need OCL expressions to express similar behavior across multiple diagrams, while we and Ziadi et al. take advantage of the combined fragments which were introduced in UML 2 after the work of Whittle and Schumann.

Ziadi et al. [18] define their transformation by pseudocode operating on algebraic definitions of sequence diagrams and state machines, while our transformation is based on graph transformation. Our support for guards in \texttt{alt/loop} and support for \texttt{par/opt/neg} is new compared to their approach.

Harel et al. [8] define a transformation from Live Sequence Charts to State to UML state charts. They include support for mandatory behavior which is not covered in our work. Multiple diagrams involving the same lifeline lead to orthogonal states in their approach, while we use multiple composite states in our approach. As opposed to their work, our proposed modeling process is linked to a formal refinement theory, and the validity of refinements can be checked in all phases including the manual modification of the generated state machines. Our transformation is defined by graph transformation rules that use the concrete syntax of sequence diagrams and state machines, while their transformation is defined as mathematical formulas.

If we combine our approach of generating state machines from sequence diagrams with sequence diagram aspects [6, 10], then weaving at the sequence diagram level should be avoided. Such weaving clutters the main sequence diagrams, since the aspects are woven into multiple places, which leads to non-optimal state machines. A better strategy is thus to postpone the weaving and to transform the sequence diagram aspects into state machine aspects, or to aspects at an even later stage such as AspectJ if the state machines are used to produce Java code.

Our previous work of semantics-based weaving of sequence diagrams [6] is not suitable within our proposed methodology in this paper. This is because the weaving does not preserve the structure of the sequence diagrams with the original combined fragments. This will lead to non-optimal state machines.

Sun [15] specifies a transformation from state charts to state machines in the AToM tool. Their transformation is restricted to the combined fragments \texttt{alt} and \texttt{loop}, while we also have rules for the \texttt{opt, par} and \texttt{neg} operators. For the comparable part we have about half as many rules as Sun. Furthermore, our rules are defined entirely by graphs, while Sun extensively needs textual pre- and post-conditions to specify much of the transformation logic for each rule.

8 Conclusions

Although there is much previous work on generating state-based specifications from interaction specifications, our approach is novel in two ways: 1) our modeling process relates to an existing refinement theory and combined fragments
From Sequence Diagrams to State Machines

are exploited in the different steps of the modeling process, and 2) the graph transformation to specify the transformation rules are based on the concrete syntax of sequence diagrams and state machines.

The usage of concrete syntax in the graph transformation rules appears to make the rules more intuitive and more concise than traditional graph transformation rules which are specified in abstract syntax. To our best knowledge no other works have defined specialized transformation mechanisms to support combined fragments.

We leave it as future work to develop tool support for our proposed modeling process. Such tool support should integrate existing refinement checker tools [12, 2] to provide messages of refinement violations.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project grant no. 215412 (EU FP7 STREP).

References


19


