Compositional Verification of Model-Level Refactorings Based On Graph Transformations

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Dénes András Bisztray MSc

Department of Computer Science

University of Leicester

May 2009

Compositional Verification of Model-Level Refactorings Based On Graph Transformations

Dénes András Bisztray

Abstract

With the success of model-driven development as well as component-based and service-oriented systems, models of software architecture are key artifacts in the development process. To adapt to changing requirements and improve internal software quality such models have to evolve while preserving aspects of their behaviour. These behaviour preserving developments are known as refactorings.

The verification of behaviour preservation requires formal semantics, which can be defined by model transformation, e.g., using process algebras as semantic domain for architectural models. Denotational semantics of programming languages are by definition compositional. In order to enjoy a similar property in the case of model transformations, every component of the source model should be distinguishable in the target model and the mapping compatible with syntactic and semantic composition.

To avoid the costly verification of refactoring steps on large systems and create refactoring patterns we present a general method based on compositional typed graph transformations. This method allows us to extract a (usually much smaller) rule from the transformation performed, verify this rule instead and use it as a refactoring pattern in other scenarios.

The main result of the thesis shows that the verification of rules is indeed sufficient to guarantee the desired semantic relation between source and target models. A formal definition of compositionality for mappings from software models represented as typed graphs to semantic domains is proposed. In order to guarantee compositionality, a syntactic criterion has been established for the implementation of the mappings by typed graph transformations with negative application conditions. We apply the approach to the refactoring of architectural models based on UML component, structure, and activity diagrams with CSP as semantic domain.

Acknowledgements

Through the past three years, there were several people who influenced my work, made my days in Leicester delightful and thus helped this thesis to be completed.

First of all, I learned from Reiko Heckel what research is about. His scientific exactitude was a great inspiration to me. His work reminded me that theory and practice should be always in balance. I am grateful for all his thorough help with papers that made me realise the importance of good presentation. He was always there when I got stuck, needed a discussion and helped me to bring my research on course again. I thank Reiko for being my guide through the labyrinths of science.

I would probably have missed the opportunity to work with Reiko, had not Tihamér Levendovszky drawn my attention to the SENSORIA scholarship. I thank Tihamér for giving me the golden tip.

I gratefully acknowledge the financial support of project SENSORIA, IST-2005-016004.

I would like to thank my examiners Gabriele Taentzer and José Luiz Fiadeiro for their thorough and detailed review of this thesis. Their appreciated criticism enabled its substantial improvement.

The collaboration with Hartmut Ehrig was invaluable. I am indebted to him for his help with the proofs and contributed work. His ideas and remarks taught me a great deal about the practical application of category theory.

The nicest experience in my research was the cooperation with Karsten Ehrig. His constant smile and joyful demeanour made him a colleague to be missed, the department felt empty after his leave. His unconditional help and instantaneous response to remarks about EMT unburdened my implementation. I am grateful to him for introducing me to the Eclipse world.

The time spent in the Computer Science Department has been great fun due to my friendly colleagues. It has been a great pleasure to share an office with Mark Parsons, João Abreu and Stephen Gorton. Although the days spent in G1 passed, I still remember the bracing conversations and that fisherman statue. It was always a

pleasure to meet Artur Boronat, Stephan Reiff-Marganiec and Fer-Jan de Vries in the department. Although not directly involved in my research, their friendly presence, stimulating discussions and useful ideas improved my perception on computer science and scientific work. I also thank Fer-Jan for proofreading the thesis and pointing out the ambiguities.

I thank my friends Dave, Martin, Andrew, Keith, Geraldine, George and Annie for all their help and the unforgettable conversations. I would also like to thank Matt, Vreni and Lucy for the all the enjoyable climbing sessions in the Tower. They brightened my days in Leicester.

Of course I thank my family for their support, their interest and for always being there when I needed them most. I conclude with the most important of all. I would like to give thanks to God for bringing all these people into my life and leading me through these long years of PhD.

Contents

Introduction

Let us start *in medias res*. In order to understand a problem, any kind of problem, the four most important things we must know are [Kre02]:

1. that there is a problem to solve (motivation);
2. what the problem is exactly (scope and challenges);
3. the techniques or strategies involved in the solution (method), and
4. what steps have been taken to solve it (solution)

This chapter is organised by the above four-point checklist to show what this thesis contains. After presenting the problem in nutshell, we outline the thesis structure.

Motivation

Nothing endures but change, as the philosopher says [Lae25]. As much as anywhere else, this applies to the world of software. A software in constant use must continually evolve, otherwise it becomes progressively less satisfactory [Leh96]. In order to improve the internal structure, performance, or scalability of software systems, such changes may be required that preserve the observable behaviour of systems.

Let us present a simple motivational example for behaviour-preserving improvements from everyday life. A user withdraws cash from an ATM with his cashcard. When the amount is selected for withdrawal, the same amount is expected to be dispensed in hard cash and subtracted from the corresponding account. The development of ATMs has gone a long way since the invention of the PIN and cashcard in 1965; more advanced algorithms and networking protocols were introduced and the overall architecture of the ATM was redesigned since then. However, no matter what clever improvement the engineers may apply to the system, everyone still expects that after selecting the £100 button, that amount is dispensed, and deducted from the account balance.

Behaviour is the total of events and actions that a system can perform and the order in which they can be executed [Bae05]. As actions are the chosen unit of observation, the observable or external aspect of the behaviour consist of actions perceivable by the environment. While the design and structure of the ATM was improved, its external behaviour — the dispensation of cash and deduction from the balance — remained the same.

A development process that improves the internal structure of existing systems without altering the external behaviour is called *refactoring* [FBB+99, NG03]. That is, the requirement of behaviour preservation is the difference between generic software development and refactoring.

Checking behaviour preservation of complex and large systems can be very costly, while the actual refactoring might only affect a relatively small fragment of the overall system. It would be advantageous if we could focus our verification on those parts of the system that have been changed. The complexity of evolving software increases unless work is done to maintain or reduce it. As already stated, refactoring is needed to cope with increasing complexity. However, the functionality of software must be continually increased to maintain user satisfaction over its lifetime [LRW+97]. Thus refactoring is not a one-time activity; refactoring must be performed continually.

Refactoring activities are often recurring as well. Like design patterns, there are common patterns of refactoring. Software companies, however, spend a lot of money and effort in reinventing code and methods that have been previously developed [THD03]. The possibility of extracting the changed parts of the system, and verifying behaviour preservation enables the recognition of commonly recurring refactoring problems. The recurring and verified refactorings should be generalised to enable their use in various situations. Essentially, a method or definition is required to create patterns analogous to the design patterns that describe generic refactorings within one modelling language.

Scope and Challenges

Today, when applications tend to be distributed and service-oriented, the most important changes take place at the architectural level. We view software architecture as defined in [GP95]: the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. Even if the architectural changes are structural, they have to take into account the behaviour encapsulated inside the components that are being replaced or

reconnected. In analogy to the programming level we speak of *architectural refactoring* if the behaviour of the software system expressed at the architecture level is due to be preserved.

As software systems of today are inherently complex, an abstract representation of their architecture is necessary. As models are offering an abstract view of a complex system, they are ideal to represent software architecture. Thus, architectural refactorings are addressed at the level of models. When refactoring is performed on a system it is possible to identify those parts that have been changed. Our aim is to verify the behaviour preservation of the changed parts of the system, that is, extract these changed parts from the system to form standalone *refactoring rules*. Then, the behaviour preservation of the refactoring for the whole system follows from the behaviour preservation of the refactoring rule. We call this *rule-level verification*. To harness these results, we would like to generalise recurring refactoring rules to create refactoring patterns. In order to achieve these goals, the following steps need to be taken:

1. To express redesign at the model-level, a *modelling language* is required capturing the structural and behavioural aspects of the system described at the level of architecture.

2. To verify behaviour preservation, the modelling language has to be provided with formal notion of behaviour, i.e. *semantics*.

3. It is necessary to have a *formal representation* of the chosen modelling language, the refactoring step and the semantic domain.

4. Based on the formal representation, a *semantic mapping* is required that maps the modelling language onto the semantic domain. As rule-level verification cannot be applied generally, conditions of *compositionality* have to be established and satisfied by the *semantic mapping*.

5. A method of creating a refactoring rule is essential; techniques are required that extract the changed parts of the system in such a way, that their application to a different system guarantees behaviour preservation.

Method

Following the steps sketched in the previous section, let us discuss the techniques and methods used.

On the concrete level, as a *modelling language* we use *combined structure modelling language (CSML)*, a UML-based language, which provides the means to describe both structure (by component and static structure diagrams) and behaviour (by activity diagrams) of software systems [OMG06b].

The formal behaviour, or *semantics* of the relevant fragment of the architecture is expressed in a denotational style using CSP [Hoa85] as a semantic domain. The *semantic relation* of behaviour preservation can be expressed conveniently using one of the refinement and equivalence relations on CSP processes. In our case study we use *trace refinement* for its theoretical simplicity, but divergence and failure refinement is equally suitable.

Architectural models and denotational semantics can be represented as instances of metamodels with a mathematical model provided by type and instance graphs. Thus, refactorings are described by graph transformations and a refactoring is a graph transformation step from a source to a target model.

The semantic mapping, denoted by *sem*, from architectural models expressed in CSML to CSP processes is defined by means of graph transformation rules.

Note that there are two distinct graph transformations as shown in Figure 1. The refactoring is represented by a graph transformation step while the semantic map is specified by a graph transformation consisting of multiple steps given by a graph transformation system. Due to the mathematical formulation of our approach, the concrete modelling language (e.g. CSML) and denotational semantics (e.g. CSP) can be replaced by other, subject to certain requirements introduced later.

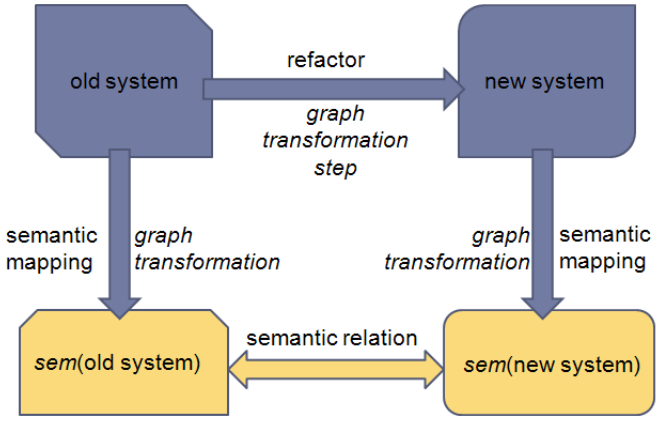


Figure 1: Overview of the transformations

Based on these ingredients, we can formalise the behaviour verification problem: a model transformation $M_1 \to M_2$ is behaviour-preserving if $sem(M_1)\ \mathcal{R}\ sem(M_2)$ where $\mathcal{R}$ is the desired relation on the semantic domain. As mentioned, the larger

M_1 and M_2 are, the more expensive the verification of relation $\mathcal{R}$ is. Thus we focus our verification on the changed parts of the model, i.e. verify the refactoring *rules* rather than the actual steps.

A *compositionality* property for the semantic mapping is necessary to verify the behaviour preservation at rule level. In general, compositionality is similar to the property of denotational semantics. As for simple mathematical expressions, we assume that the the meaning of expression $2 + 5$ is determined by the meaning of 2, 5 and the $+$ operator, i.e. $[[2 + 5]] = [[2]] \bigoplus [[5]]$.

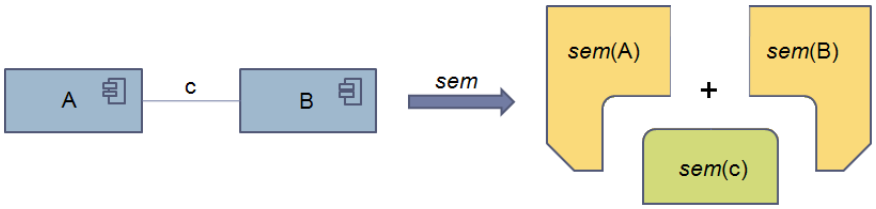


Figure 2: Compositional semantic mapping

In terms of model transformations, a compositional semantic mapping is presented in Figure 2. A system consisting of components A and B with a connector c is mapped to a semantic domain through transformation *sem*. The result is a set of semantic expressions where $sem(A)$, $sem(B)$ and $sem(c)$ are distinguishable and their composition is the semantics of the whole system.

Compositionality is an important property for denotational semantics and thus for model transformations that establish a mapping between existing modelling artifacts and their denotational semantics. Without compositionality the modular specification and verification of model transformations would be impossible.

A typical semantic verification scenario is depicted in Figure 3. A modelling language (ML) is mapped to a semantic domain (SD) and from the models, programming language (PL) code is generated. To verify semantic consistency, semantics for the programming language has to be defined typically through a mapping $PL \rightarrow SD$. The generated source code is semantically correct if the triangle commutes. Although the different model instances are numerous, they are composed from the basic ele-

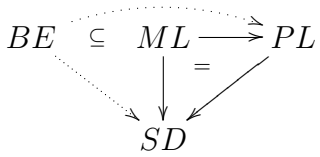


Figure 3: Semantic verification

ments (BE) of the modelling language. In case of compositional transformations, the mapping can be described in terms of the basic building blocks, enabling the modular verification of various semantic properties.

Architectural refactorings at the model-level are unlikely to be created directly from semantics-preserving rules. Existing rule catalogues focus on object-oriented systems and are effectively liftings to the model level of refactoring rules for OO programs. Rather, an engineer using a modelling tool performs a manual model transformation $M_1 \rightarrow M_2$ from which a verifiable refactoring rule has to be extracted first. In this we follow the idea of *model transformation by example* [Var06] where model transformation rules expressed as graph transformations are derived from sample transformations.

Solution

The semantic mapping was implemented using the Tiger EMF Transformer [Tig07] tool. It was designed to fulfill the compositionality conditions and thus enable the behavioural verification of rules. The reason for such a property to work is twofold. On one hand, the various equivalence and refinement notions in CSP are closed under context. This means that a relation between two sets of CSP expressions will hold also after embedding both of them into the same context (i.e. into the same set of CSP expressions). On the other hand, a graph production rule changes only a subset of nodes in a graph; the context (i.e. the unchanged nodes of the initial graph) remains the same. These two observations enabled the definition and implementation of a compositional mapping. Compositionality provides the preservation of closure under context for the semantic mapping.

We have shown formally that the verification of refactoring *rules*, rather than steps, is possible: assuming a refactoring $G \implies H$, via the graph transformation rule $p : L \rightarrow R$, if the relation $\mathcal{R}$ holds for $sem(L)\ \mathcal{R}\ sem(R)$ then $sem(G)\ \mathcal{R}\ sem(H)$ also holds. To make it feasible, we defined the notion of *compositionality* for any total functions between sets of graphs (representing models) defined by graph transformations. Conditions are provided and proved that guarantee compositionality for simple graph transformations and graph transformations with negative application conditions.

Extracting the refactoring rule is not necessarily obvious; there can be complicated refactorings that span change on the behaviour of multiple components. To determine the mechanics of producing a rule, we performed extractions on proven and successful

refactorings. In general, for a transformation $G \implies H$ with $sem(G)\ \mathcal{R}\ sem(H)$, we extract the smallest rule such that:

1. when applying it on G at the appropriate match, the transformation step produces H
2. $sem(L)\ \mathcal{R}\ sem(R)$, i.e. the *semantic relation* $\mathcal{R}$ holds also at rule-level.

The rule extraction consists of two steps. In the first step, the difference between G and H is extracted, to form a minimal production rule which is shown to fulfill *requirement 1*. As the minimal rule may represent a semantically incomplete part of the domain, a necessary context needs to be added to fulfill *requirement 2*. Both the difference extraction and context determination is performed interactively by the refactoring developer.

Publications

The contributions of this thesis extend and generalise material published in several research papers. The original idea of rule-level verification of business processes (represented as activity diagrams) was published in [BH07]. The problem statement and implementation idea of the semantic mapping as a graph transformations system based on triple graph grammars were published in [VAB+08]. In [BHE08] our complete approach has been summarised: the modelling domain was extended from business processes to architectural models and the correctness of rule-level verification was formalised and proven. The implementation of the semantic mapping was elaborated in [DB08] with the concept of triple graph grammars dropped due to performance issues. The problems of rule extraction and tool support were published in [Bis08] and elaborated in [BHE09c]. The concept of compositionality and the related theorems were published in [BHE09b, BHE09a].

Outline

After this short introduction to the topic, let us present the structure of the thesis. It consists of two main parts: the first two chapters present the context of the thesis, while the next five introduce the main contributions. The last chapter concludes the thesis.

Chapter 1 introduces theories and practices that our contributions are based on or use directly. It starts with models and model transformations in general. Then it elaborates on graphs, graph transformation theory and negative application conditions. Approaches of graph based model transformations finish the chapter.

In *Chapter 2* the work related but not directly connected to our contributions is discussed. First, the verification methods of temporal properties are presented that deals with model and graph transformations. Then we proceed to bidirectional transformations. The next section presents a short taxonomy of refactoring approaches. Finally, we position our work by discussing the related approaches themselves

The next three chapters present an elaborate case study for the theoretical contributions: a compositional semantic mapping between a modelling language and semantic domain. In *Chapter 3* the source language of the transformation is introduced: combined structure modelling language (CSML), a UML-based language for software architecture modelling. It uses the UML component, composite structure and activity diagrams combined in a common metamodel. *Chapter 4* deals with CSP, the chosen semantic domain and target language of the transformation. Both the term and graph-based representation is introduced with examples of the correspondence between the two. And finally *Chapter 5* presents the semantic mapping itself with its working mechanics elaborated.

The main theoretical contributions are detailed in *Chapter 6*. The concept of compositionality is formalised, the correctness of rule-level verification is proven. The compositionality theorem is established and proved. Conditions are provided for compositionality for semantic mappings implemented as graph transformations.

Chapter 7 completes the circle. It applies the theory to the practice: the theoretical results of rule-level verification are used on the architectural domain. First, the rule-level verification based on the approach of *rule extraction by example* is presented. Then the generalisation of extracted rules is discussed aiming for the creation of refactoring patterns.

Chapter 1

Background

This chapter is an introduction to the theoretical background of our research. It starts with the general concept of modelling in Section 1.1, and model transformations in Section 1.2. Section 1.4 presents the fundamental definitions of graph transformations, while the advanced theory is introduced in Section 1.5. The graph transformation theory is extended to accommodate negative application conditions in Section 1.6. Finally in Section 1.7, the two relevant graph-based model-transformation approaches are introduced.

1.1 Modelling and UML

A *model* is a simplified abstract view of the complex reality. Using models to describe increasingly complicated software systems is particularly useful. Models provide abstractions, which allow developers to focus on the relevant properties of the system, and ignore unnecessary complications [MM03]. We use the notion of model as *"a description of a system written in a well-defined language. A well-defined language is a language with well defined form (syntax) and meaning (semantics), which is suitable for automated interpretation by a computer"* [KWB03]. A model has an abstract and a concrete syntax. The abstract syntax is often defined in terms of a *metamodel*, which is an explicit model of the constructs and well-formedness rules needed to build specific models within a domain of interest. The concrete syntax is the (graphical or textual) representation of the model.

Although it would be ideal to represent the system with one concise model, a system description requires multiple *views*: each view represents a projection of the complete system that shows a particular aspect. A view requires a number of *diagrams* that visualise the information of that particular aspect of the system. The

concepts used in the diagrams are *model elements* that represent common object-oriented concepts such as classes, objects and messages, and their relationships, including associations, dependencies and generalisation. These are the basic elements for modelling an object-oriented software system. A model element is used in several different diagrams, but it always has the same meaning and symbol [EPLF03].

Unified Modelling Language (UML) [OMG06b] is a modelling language with well-defined abstract syntax used to specify, visualise, construct and document the artifacts of an object-oriented software-intensive system under development [oC02]. The abstract syntax of the UML is described with metamodels. The UML metamodel is a model that defines the characteristics of each UML model element, and its relationships to other model elements. The metamodel is defined using an elementary subset of UML, and is supplemented by a set of formal constraints written in the Object Constraint Language (OCL) [OMG06a, Sel05]. This combination (constructs and rules) represents a formal specification of the abstract syntax of UML that is used to represent UML models.

1.2 Model Transformations

The significance of model transformations can be understood from the perspective outlook of model-driven architecture. The original drive behind model transformation was the mapping of a platform independent model (PIM) to a platform specific model (PSM). The software architect builds the PIM, which does not contain the details of a specific platform. The PSM enables the implementation of the system with the desired architecture. There can be several target architectures, and thus several PSMs. The mapping provides specifications for the transformation of a PIM into a PSM [MM03]. A typical example of such a transformation is transforming an UML-based PIM to Enterprise Java Beans (EJB) [Sun09] models.

As mentioned, it is possible to have several modelling perspectives or *views* of the system. Depending on relevance, usually one view is highlighted. It is often necessary to convert to different views of the system at an equivalent level of abstraction, or in other cases to convert models of one perspective from one abstraction level to another [Ala04]. Model transformations facilitate achieving these goals.

A generic definition of a rule-based model transformation is given by [KWB03]: *the automatic generation of a target model from a source model according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model*

in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

We classify model transformation based on [MCvG05] into the following categories:

- *Endogenous and Exogenous transformations.* The distinction is based on the language in which the source and target models of a transformation are expressed. *Endogenous* transformations use the same language, while *exogenous* transformations transform between different languages or different models of the same language.

- *Horizonal and Vertical transformation.* The distinction is based on the abstraction level in which the source and target models are. A transformation is *horizontal* if both models are at the same abstraction level, while *vertical* transformations transform between different level of abstraction.

To apply these definitions, we position our transformations introduced eariler. The semantic mapping is an *exogenous* and *horizontal* transformation, as it transforms between different languages but same abstraction level. The refactoring step is an *endogenous* and *horizontal* transformation. It transforms within the same modelling language and abstraction level.

1.3 Design Patterns and Refactoring

The notions of refactoring and refactoring patterns are introduced in this section. Because of the similarity to design patterns, the differences and commonalities of these concepts are explained.

In the area of software maintenance, there are three related concepts: *restructuring*, *reengineering* and *refactoring*. To ensure clarity we present their definition and relation to each other.

The most general and high-level concept of these three is the *reengineering* [DDN08]. Reengineering, as defined in [CCI90] is *"the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."*. Reengineering is concerned with the creation of a well-structured system from legacy or deteriorated code [FR98]. As shown in Figure 1.1, reengineering may contain restructuring and refactoring.

The explicit requirement of behaviour preservation emerged with the concept of *restructuring* [Arn86, GN93]. The precise definition of restructuring according to

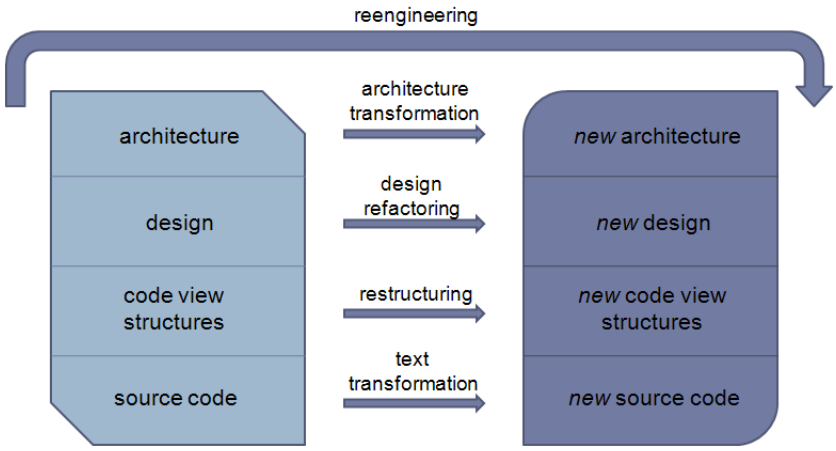


Figure 1.1: Overview of reengineering

[CCI90] is *"the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system."* As shown in Figure 1.1, reengineering is dealing with code structure views, which are implementation level models of the system. The slight difference between the original definition of refactoring and restructuring apart from the programming paradigm, is the level of its application.

The word *refactoring* was first used in [Opd92], where it meant an object-oriented version of restructuring. In [FBB+99], refactoring is defined as *"the process of changing a [object oriented] software system in such a way that it does not alter the external behaviour of the code, yet it improves its internal structure"*. The main idea of refactoring is to redistribute classes, variables and methods across the class hierarchy in order to facilitate future adaptations and extensions [FBB+99]. Although being the narrowest in its original meaning, the notion of refactoring *(verb)* is now used as a generic term for behaviour preserving transformation of an arbitrary software artifact. When refactoring, developers no longer concern with adding functionality to the system. Thus a *refactoring (noun)* means a single technique or procedure that improves the design of existing software systems [NG03].

To understand what is a *refactoring pattern*, we elaborate on the concept of *design*

patterns. *Design patterns* are elegant problem-solution pairs that codify exemplary, tried-and tested design principles to commonly occurring problems in software engineering [GHJV94, NG03]. A design pattern usually consists of four essential elements [GHJV94]:

- The *pattern name* is used as a handle to describe a design problem, its solutions, and consequences in a couple of words.
- The *problem* describes when to apply the pattern. It explains the problem and its context.
- The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations. It is not a concrete implementation, but rather an abstract description of a design problem and how a general arrangement of elements solves it.
- The *consequences* are the results and trade-offs of applying the pattern.

The idea of *refactoring pattern* is the coupling of *refactoring* and *design patterns*: classic solutions to recurring design problems of existing code [Ker04].

The main difference between them is their application: design patterns are used during the design process of the application while refactoring is done during maintenance. Design patterns loosen the binding between program components thus creating a flexible design. Such a flexible and elegant design enables later program evolution with minimal change. However, there are some problems that may arise:

- The design process must undergo a couple of iterations to make good use of design patterns. Often, there is no time for creating flexible design.
- The program can be prepared for changes that can be foreseen. Although the good use of design patterns make a program as generic as possible, it is possible that the required flexibility is not present in all parts of the software.

This required flexibility or design excellence can be introduced later by applying refactoring patterns to the application.

In refactoring pattern description, the problem statement is always a description of a class or object structure that is symptomatic of inflexible design [GHJV94]. This relatively concrete and precise problem statement enables the implementation of refactoring patterns [Ecl09, VS09] in software development environments. There

are also approaches that aim to automate the application of refactoring patterns as detailed in Section 2.3.

Ideas and design principles found in design patterns and refactoring patterns are often interchangeable. It is possible that a refactoring pattern leads to a design pattern and vice verse.

The difference between *a refactoring* (or a *refactoring step*) and a *refactoring pattern* is the generality and level of documentation. While a refactoring is a solution to a single problem, the refactoring pattern (like the design pattern) is generic and well documented. A refactoring can be the application of a refactoring pattern.

The difference between a *refactoring pattern* and *refactoring rule* is often blurred. In our understanding, the difference is the level of formalisation: a refactoring rule is such a pattern that is implemented in a tool or formalised with mathematical means.

1.4 Graph Transformations

In this section the fundamental concepts of graphs and graph transformations are introduced.

1.4.1 Basic Concepts

First we present the definitions of graphs, typed graphs and typed graph morphisms for clarity.

Definition 1.4.1. ***(Graph and Graph Morphism** [Ehr87])* *A* graph $G = (V, E, s, t)$ *consists of a set V of nodes (also called vertices), a set E of edges, and two functions $s, t : E \to V$, the source and target functions:*

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

Given graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2$, a graph morphism *$f : G_1 \to G_2$, $f = (f_V, f_E)$ consists of* two functions *$f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$.*

$$\begin{array}{ccc} E_1 & \overset{s_1}{\underset{t_1}{\rightrightarrows}} & V_1 \\ \downarrow{\scriptstyle f_E} & = & \downarrow{\scriptstyle f_V} \\ E_2 & \overset{s_2}{\underset{t_2}{\rightrightarrows}} & V_2 \end{array}$$

A graph morphism f is injective *(resp.* surjective*) if both functions f_V, f_E are*

injective (or surjective, respectively); f is called isomorphic *if it is bijective, that is both injective and surjective.*

In this algebraic representation, a graph is considered as a two sorted *algebra* where the sets of vertices V and edges E are the carriers, while the source $s : E \to V$ and target $t : E \to V$ are two unary operators [CMR+97].

The composition property of graph morphisms is one of the necessary ingredients to show that graphs form a category (Def. A.1.1).

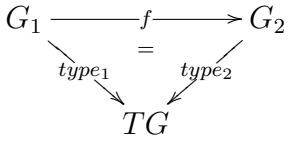
Fact 1.4.1. ***(Composition of Graph Morphisms** [Ehr87]) Given two graph morphisms $f = (f_V, f_E) : G_1 \to G_2$ and $g = (g_V, g_E) : G_2 \to G_3$, the composition $g \circ f = (g_V \circ f_V, g_E \circ f_E) : G_1 \to G_3$ is again a graph morphism.*

As mentioned in Section 1.1, a metamodel is an explicit model with set of well-formedness rules. The model part can be conveniently expressed as a type graph. A typed graph consists of a graph and a corresponding type graph. The type graph defines a set of types that are assigned to the nodes and edges of the graph by a typing morphism.

Definition 1.4.2. ***(Typed Graph** [Ehr87]) A* type graph *is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ where V_{TG} and E_{TG} are called the vertex and edge type alphabets, respectively.*

A tuple $(G, type)$ of a graph G together with a graph morphism $type : G \to TG$ is called a typed graph *over TG.*

Definition 1.4.3. ***(Typed Graph Morphism)** Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a typed graph morphism $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \circ f = type_1$.*



In order to use categorical constructs on graphs, it is necessary to show that graphs form a category.

Corollary 1.4.1. ***(Category of Graphs** [Mar96])*

- *The class of all graphs (as defined in Definition 1.4.1) as objects and of all graph morphisms (see Definition 1.4.1) forms the category* **Graphs**, *with the composition given in Fact 1.4.1, and the identities are the pairwise identities on nodes and edges.*

- *Given a type graph TG, typed graphs over TG and typed graph morphisms (see Definition 1.4.3) form the category* $\mathbf{Graphs_{TG}}$.

1.4.2 Double Pushout Approach

The core of a graph transformation is a graph production $p : L \to R$ consisting of a pair of graphs L and R. L is called the left-hand side graph (LHS) and R is called the right-hand side graph (RHS). Applying rule p to a source graph means finding a match of L in the source graph and replacing it with R, thus creating the target graph. The technical difficulty is in performing this operation.

In the DPO approach, a graph K is used. K is the common interface of L and R, i.e. their intersection. Hence, a rule is given by a span $p : L \leftarrow K \rightarrow R$.

Definition 1.4.4. ***(Graph Production** [CMR+97])* *A* (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *consists of (typed) graphs L,K,R, called the left-hand side, gluing graph (or interface graph) and the right-hand side respectively, and two injective (typed) graph morphisms l and r.*

Given a (typed) graph production p, the inverse production *is defined by* $p^{-1} = R \xleftarrow{r} K \xrightarrow{l} L$.

A graph transformation starts by finding a match m of L in the source graph G. Then, the vertices and edges of $L \setminus l(K)$ are removed from G. Similar to the rule-level interface graph K, an intermediate graph D is created, $D = (G \setminus m(L) \cup l(K))$. Since D has to be a graph, no dangling edges are allowed. To ensure this, G has to be the gluing of $m(L \setminus l(K))$ and D, that is a pushout complement (Def. A.1.2). To produce the target graph H, graph D has to be glued together with $R \setminus l(K)$.

Definition 1.4.5. ***(Graph Transformation** [CMR+97])* *Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a (typed) graph G with a (typed) graph morphism $m : L \to G$, called the match, a* direct (typed) graph transformation $G \overset{p,m}{\Rightarrow} H$ *from G to a (typed) graph H is given by the following double-pushout (DPO) diagram, where* (1) *and* (2) *are pushouts in the category* **Graphs** *(or* $\mathbf{Graphs_{TG}}$ *respectively):*

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ \downarrow{\scriptstyle m} & (1) & \downarrow{\scriptstyle k} & (2) & \downarrow{\scriptstyle n} \\ G & \xleftarrow{f} & D & \xrightarrow{g} & H \end{array}$$

A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of direct (typed) graph transformations is called a (typed) graph transformation *and is denoted by $G_0 \overset{*}{\Rightarrow} G_n$. For $n = 0$, we have*

the identity (typed) graph transformation $G_0 \overset{id}{\Rightarrow} G_0$. *Moreover, for* $n = 0$ *we allow also graph isomorphisms* $G_0 \cong G'_0$, *because pushouts and hence also direct graph transformations are only unique up to isomorphism.*

The gluing condition is a constructive approach to formulate a syntactic criterion for the applicability of a (typed) graph production.

Definition 1.4.6. ***(Gluing Condition** [CMR+97]) Given a (typed) graph production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, *a (typed) graph* G, *and a match* $m : L \rightarrow G$ *with* $X = (V_X, E_X, s_X, t_X)$ *for all* $X \in L, K, R, G$, *we define:*

- *The* gluing points GP *are those nodes and edges in* L *that are not deleted by* p, *i.e.* $GP = l_V(V_K) \cup l_E(E_K) = l(K)$.
- *The* identification points IP *are those nodes and edges in* L *that are identified by* m, *i.e.* $IP = \{v \in V_L \mid \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L \mid \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$.
- *The* dangling points DP *are those nodes in* L *whose image under* m *are the source or target of an edge in* G *that does not belong to* $m(L)$, *i.e.* $DP = \{v \in V_L \mid \exists e \in E_G \setminus m_E(E_L) : s_G(e) = m_V(v) \text{ or } t_G(e) = m_V(v)\}$.

Production p *with match* m *satisfy the* gluing condition *if all identification points and all dangling points are also gluing points, ie.e* $IP \cup DP \subseteq GP$.

A graph transformation is a sequence of productions applied to a graph. A set of production rules that may applied to a graph is defined as a graph transformation system. A graph grammar is basically a graph transformation system with a fixed start graph.

Definition 1.4.7. ***(GT System, Graph Grammar** [CMR+97]) A* typed graph transformation system $GTS = (TG, P)$ *consists of a type graph* TG *and a set of typed graph productions* P.

A typed graph grammar $GG = (GTS, S)$ *consists of a typed graph transformation system* GTS *and a typed start graph* S.

We may use the abbreviation GT system *for* typed graph transformation system.

1.5 Theory of Graph Transformations

Following the DPO approach, we introduce the advanced concepts that are essential to the contributions presented. The four most important concepts are concurrency,

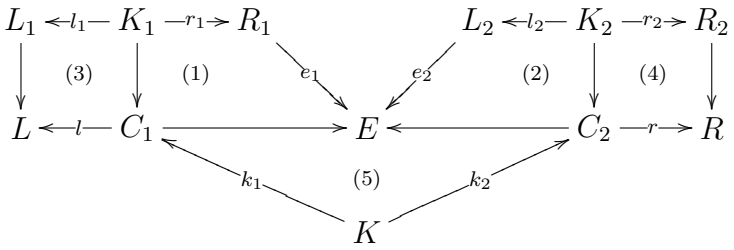
extension, confluence and termination. Concurrency enables the execution of two sequentially dependent productions via a concurrent production. Extension analyses the problem of extending a graph transformation to a larger graph. Termination and confluence together provide a graph transformation system functional behaviour.

1.5.1 Concurrency

A concurrent graph transformation is basically the sequential merge of a production rule sequence. Given a sequence of graph transformations $G \overset{p_1}{\Rightarrow} H_1 \overset{p_2}{\Rightarrow} G'$ we construct a so-called E-concurrent production $p_1 *_E p_2$ such that $G \overset{p_1 *_E p_2}{\Rightarrow} G'$. This E-concurrent rule is created with an *epimorphic overlap graph* E that is a subgraph of H_1. E is created through a jointly surjective morphism pair $(e_1 : R_1 \to E, e_2 : L_2 \to E)$.

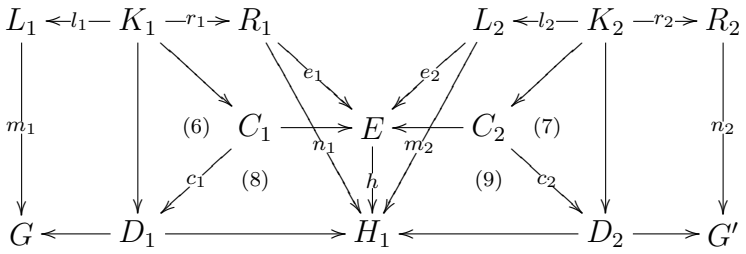
Definition 1.5.1. ***(Jointly Surjective Morphism Pair)*** *A pair of morphisms $f_1 : A \to E$ and $f_2 : B \to E$ in an arbitrary category is* jointly surjective*, if for all $e \in E$ there exists either an $a \in A$ for which $e = f_1(a)$ or a $b \in B$ for which $e = f_2(b)$.*

Definition 1.5.2. ***(E-dependency Relation and E-concurrent Production and Transformation** [EEPT06])* *Given two productions p_1 and p_2 with $p_i = (L_l \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ for $i = 1, 2$, an object E with morphisms $e_1 : R_1 \to E$ and $e_2 : L_2 \to E$ is an* E-dependency relation *for p_1 and p_2 if (e_1, e_2) is jointly surjective and the pushout complements* (1) *and* (2) *over $K_1 \xrightarrow{r_1} R_1 \xrightarrow{e_1} E$ and $K_2 \xrightarrow{l_2} L_2 \xrightarrow{e_2} E$ exist:*



Given an E*-dependency relation with jointly surjective (e_1, e_2) for the productions p_1 and p_2. The* E-concurrent production *$p_1 *_E p_2$ is defined by $p_1 *_E p_2 = (L \xleftarrow{l \circ k_1} K \xrightarrow{r \circ k_2} R)$ as shown in the above diagram, where* (3) *and* (4) *are pushouts and* (5) *is a pullback.*

A transformation sequence $G \overset{p_1, m_1}{\Rightarrow} H_1 \overset{p_2, m_2}{\Rightarrow} G'$ is called E-related *if there exists $h : E \to H_1$ with $h \circ e_1 = n_1$ and $h \circ e_2 = m_2$ and there are morphisms $c_1 : C_1 \to D_1$ and $c_2 : C_2 \to D_2$ such that* (6) *and* (7) *commute and* (8) *and* (9) *are pushouts:*



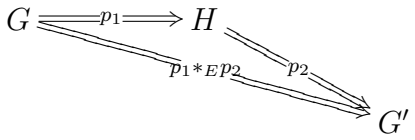
If the E-dependency relation exists, the following theorem shows that not only it is possible to construct a direct graph transformation from two sequential productions p_1 and p_2 via the E-concurrent rule $p_1 *_E p_2$, but also to sequentialise this E-concurrent production.

Theorem 1.5.1. ***(Concurrency Theorem** [EEPT06]) Given two (typed) graph productions p_1 and p_2 and an* E*-concurrent (typed) graph production $p_1 *_E p_2$, we have:*

1. Synthesis. *Given an E-related transformation sequence $G \Rightarrow H \Rightarrow G'$ via p_1 and p_2, then there is a* synthesis construction *leading to a direct transformation $G \Rightarrow G'$ via $p_1 *_E p_2$.*

2. Analysis. *Given a direct transformation $G \Rightarrow G'$ via $p_1 *_E p_2$, then there is an* analysis construction *leading to an E-related transformation sequence $G \Rightarrow H \Rightarrow G'$ via p_1 and p_2*

3. Bijective correspondence. *The synthesis and analysis constructions are inverse to each other up to isomorphism, provided that every (e_1, e_2) is a jointly surjective pair:*



1.5.2 Extension

Extension is useful, when there is a larger graph, however the graph transformation changes only a small part of it. We assume two graphs G'_0, the large graph, and G_0 the small subgraph connected via an extension morphism $k_0 : G_0 \to G'_0$. A graph transformation $G_0 \overset{*}{\Rightarrow} G_n$ is then extended to $G'_0 \overset{*}{\Rightarrow} G'_n$ with same rules applied in the same order. This extension is obtained through an extension diagram.

Definition 1.5.3. ***(Extension Diagram** [EEPT06])* *An* extension diagram *is a diagram* (1)*, as shown below,*

$$\begin{array}{ccc} G_0 & \overset{*}{\Rightarrow} & G_n \\ \downarrow{\scriptstyle k_0} & (1) & \downarrow{\scriptstyle k_n} \\ G_0' & \overset{*}{\Rightarrow} & G_n' \end{array}$$

where $k_0 : G_0 \to G_0'$ is a morphism, called extension morphism*, and $t : G_0 \overset{*}{\Rightarrow} G_n$ and $t' : G_0' \overset{*}{\Rightarrow} G_n'$ are transformations via the same productions $(p_0, ..., p_{n-1})$ and matches $(m_0, ..., m_{n-1})$ and $(k_0 \circ m_0, ..., k_{n-1} \circ m_{n-1})$ respectively, defined by the following DPO diagrams:*

$$\begin{array}{lccccc} p_i\colon & L_i & \leftarrow{\scriptstyle l_i}- & K_i & -{\scriptstyle r_i}\rightarrow & R_i \quad (i = 0, ..., n-1), n > 0 \\ & \downarrow{\scriptstyle m_i} & & \downarrow{\scriptstyle j_i} & & \downarrow{\scriptstyle n_i} \\ & G_i & \leftarrow{\scriptstyle f_i}- & D_i & -{\scriptstyle g_i}\rightarrow & G_{i+1} \\ & \downarrow{\scriptstyle k_i} & & \downarrow{\scriptstyle d_i} & & \downarrow{\scriptstyle k_i+1} \\ & G_i' & \leftarrow{\scriptstyle f_i'}- & D_i' & -{\scriptstyle g_i'}\rightarrow & G_{i+1}' \end{array}$$

In order to formulate Definition 1.5.5, we have to introduce the concept of a derived span. The derived span is intuitively a generic interface graph construction. Given a graph transformation $t : G_0 \Rightarrow G_n$ the derived span is $der(t) = (G_0 \leftarrow D \rightarrow G_n)$.

Definition 1.5.4. ***(Derived Span** [EEPT06])* *The* derived span *of an identity transformation $t : G \overset{id}{\Rightarrow} D$ is defined by $der(t) = (G \leftarrow G \rightarrow G)$ with identity morphisms.*

The derived span of a direct transformation $G \overset{p,m}{\Rightarrow} H$ is the span $(G \leftarrow D \rightarrow H)$.

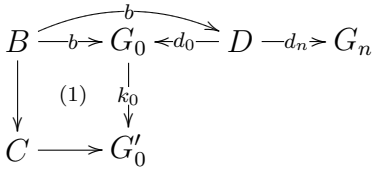
For a transformation $t : G_0 \overset{*}{\Rightarrow} G_n \Rightarrow G_{n+1}$, the derived span is the composition via pullback (PB) of the derived spans $der(G_0 \overset{*}{\Rightarrow} G_n) = (G_0 \overset{d_0}{\leftarrow} D' \overset{d_1}{\rightarrow} G_n)$ and $der(G_n \Rightarrow G_{n+1}) = (G_n \overset{f_n}{\leftarrow} D_n \overset{g_n}{\rightarrow} G_{n+1})$ this construction leads to the derived span $der(t) = (G_0 \overset{d_0 \circ d_2}{\longleftarrow} D \overset{g_n \circ d_3}{\longrightarrow} G_{n+1})$:

$$\begin{array}{ccccccc} G_0 & \leftarrow{\scriptstyle d_0}- & D' & -{\scriptstyle d_1}\rightarrow & G_n & \leftarrow{\scriptstyle f_n}- & D_n & -{\scriptstyle g_n}\rightarrow & G_{n+1} \\ & & & \nwarrow{\scriptstyle d_2} & (PB) & \nearrow{\scriptstyle d_3} & & & \\ & & & & D & & & & \end{array}$$

The construction of the extension diagram is not always possible. We need to formulate a consistency condition. The boundary graph B of an initial pushout

(Def. A.1.4) is the smallest subgraph of G_0 that contains the identification and dangling points (Def. 1.4.6) of $k_0 : G_0 \to G'_0$. If this graph is preserved by the transformation, the extension diagram exists.

Definition 1.5.5. ***(Boundary Consistency** [EEPT06])* *Given a transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ with derived span $der(t) = (G_0 \overset{d_0}{\leftarrow} D \overset{d_n}{\rightarrow} G_n)$, a morphism $k_0 : G_0 \to G'_0$ is called* boundary consistent *with respect to t if there exist an initial pushout (1) over k_0 and an injective morphism b with $d_0 \circ b = b_0$:*



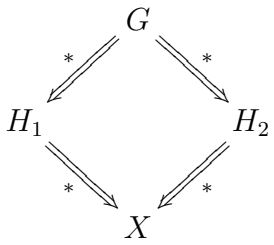
The following theorem shows that boundary consistency is sufficient for the existence of the extension diagram.

Theorem 1.5.2. ***(Embedding Theorem** [EEPT06])* *Given a transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ and a morphism $k_0 : G_0 \to G'_0$ which is consistent with respect to t, then there is an extension diagram over t and k_0.*

1.5.3 Confluence and Termination

The concept of *confluence* originates from term-rewriting systems describing that although terms can be rewritten in different ways, the result remains the same. As it is important to know whether a graph transformation system shows a functional behaviour (i.e. it terminates and provides unique result for isomorphic source models), the concept of confluence has been transferred to term graphs [Plu99], hyper graphs [Plu93] and typed graphs as well [HKT02a].

Definition 1.5.6. ***(Confluence and Local Confluence** [HKT02b])* *A* graph transformation system *is confluent if for all graph transformations $H_1 \overset{*}{\Leftarrow} G \overset{*}{\Rightarrow} H_2$ there is a graph X together with transformation sequences $H_1 \overset{*}{\Rightarrow} X$ and $H_2 \overset{*}{\Rightarrow} X$.*



A graph transformation system *is* locally confluent *if this property holds for each pair of* direct graph transformations*. The system is* confluent *if this holds for all pairs of transformations.*

Termination is a simple, yet important property of graph transformation systems. A *graph transformation* $G \overset{*}{\Rightarrow} H$ is called *terminating* if no graph production rule in the GTS is applicable to H any more.

Definition 1.5.7. ***(Termination** [EEPT06])* *A* graph transformation system *is* terminating*, if there is no infinite sequence of graph transformations* $(t_n : H \overset{n}{\Rightarrow} H_n)_{n\in\mathbb{N}}$ *with* $t_{n+1} = G \overset{t_n}{\Rightarrow} G_n \Rightarrow G_{n+1}$ *[EEPT06].*

Termination and local confluence together ensures the functional behaviour of a graph transformations system.

Theorem 1.5.3. ***(Functional Behaviour of GT Systems** [EEPT06])* *Given a terminating and locally confluent graph transformation system* GTS*, then* GTS *has a functional behaviour in the following sense:*

1. *For each graph* G*, there is a graph* H *together with a terminating graph transformation* $G \overset{*}{\Rightarrow} H$ *in* GTS*, and* H *is unique up to isomorphism.*

2. *Each pair of graph transformations* $G \overset{*}{\Rightarrow} H_1$ *and* $G \overset{*}{\Rightarrow} H_2$ *can be extended to terminating graph transformations* $G \overset{*}{\Rightarrow} H_1 * \Rightarrow H$ *and* $G \overset{*}{\Rightarrow} H_2 * \Rightarrow H$ *with the same graph* H*.*

In the following we present the necessary definitions to prove local confluence and termination.

Confluence can be shown by using the concept of critical pairs. Critical pairs which can be detected and analysed statically, represent potential conflicts in a minimal context [HKT02b]. Given a $GTS = (TG, P)$ and a graph G, there can be several rules in P applicable to G. Given two rules $p_1 : L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1$ and $p_2 : L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2$ applicable to G via matches m_1 and m_2. There is no conflict if, after applying any of them, the other one is still applicable, i.e. the direct graph transformation defined by the former does not disable the application of the latter.

Definition 1.5.8. ***(Parallel Independence and Conflict** [EEPT06])*

Two direct graph transformations $G \overset{p_1,m_1}{\Rightarrow} H_1$ *and* $G \overset{p_2,m_2}{\Rightarrow} H_2$ *are* parallel independent *of all nodes and edges in the intersection of the two matches are gluing items with respect to both transformations, i.e.*

$$m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2)).$$

Two direct graph transformations $G \overset{p_1,m_1}{\Rightarrow} H_1$ *and* $G \overset{p_2,m_2}{\Rightarrow} H_2$ *are* in conflict *if they are not parallel independent. This type of conflict is called* delete-use conflict.

A critical pair characterises the conflict situation in a minimal context.

Definition 1.5.9. ***(Critical Pair** [EEPT06]) A critical pair for the pair of rules* (p_1, p_2) *is a pair of direct graph transformations* $P_1 \overset{p_1,o_1}{\Leftarrow} K \overset{p_2,o_2}{\Rightarrow} P_2$ *in conflict, such that* o_1 *and* o_2 *are jointly surjective morphisms.*

The context is minimal, because o_1 and o_2 are required to be jointly surjective morphism. This means that each item in K has a preimage in L_1 or L_2, thus K can be considered as a suitable gluing of L_1 and L_2.

If GTS does not contain critical pairs, it is locally confluent. A working implementation for checking critical pairs can be found in the Attributed Graph Grammar System (AGG) [AGG07].

Although termination is generally undecidable, special criteria were introduced in [EEdL$^+$05, LPE07] and an implementation for termination checking is also built into AGG [BKPPT05]. Unfortunately these criteria are not applicable in our case (Sec. 6.6.4) we present the necessary definitions from [EEdL$^+$05] that helps us proving termination.

The notion of *essential match* deals with the possible re-application of a production to the similar match.

Definition 1.5.10. ***(Tracking Morphism and Essential Match** [EEdL$^+$05]) Given a (typed) graph grammar with injective matches. A production* p *given by an injective morphism* $r : L \to R$ *and injective match* $m : L \to G$ *leading to a direct transformation* $G \overset{p,m}{\Rightarrow} H$ *via* (p, m) *defined by the pushout* (1) *of* r *and* m*. The morphism* $d : G \to H$ *is called* tracking morphism *of* $G \overset{p,m}{\Rightarrow} H$*:*

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle m^*} \\ G & \xrightarrow{d} & H \end{array} \qquad \begin{array}{ccc} & & L \\ & {\scriptstyle m_0}\swarrow & \downarrow{\scriptstyle m_1} \\ G_0 & \xrightarrow{d_1} & H_0 \end{array}$$

Since both r *and* m *is injective, the pushout properties of* (1) *imply that also* d *and* m^* *are injective.*

Given a transformation $G_0 \overset{*}{\Rightarrow} H_1$*, i.e. a sequence of direct transformations with an induced injective tracking morphism* $d_1 : G_0 \to H_1$*, a match* $m_1 : L \to H_1$ *of* L *in* H_1 *has an* essential match $m_0 : L \to G_0$ *of* L *in* G_0 *if we have* $d_1 \circ m = m_1$*. Note that if the transformation is nondeleting, there is at most one essential match* m_0 *for* m_1*, because* d_1 *is injective.*

A non-deleting rule is *self-disabling* if it has a NAC that prohibits the existence of the same pattern that the rule creates.

Definition 1.5.11. ***(Self-Disabling Production)*** *Given a production rule p by $r : L \to R$ with NAC $n : L \to N$. $NAC(n)$ is* self-disabling *if there is an injective $n' : N \to R$ such that $n' \circ n = r$. A production is* self-disabling *if it is nondeleting and has a* self-disabling *NAC.*

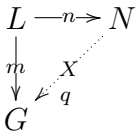
The following lemma establishes that a *self-disabling* production cannot be applied on the same match again, and extends it to graph transformations that consists of only *self-disabling* rules.

Lemma 1.5.1. ***(Essential Match Applicability** [EEdL$^+$05] **)** In every transformation starting from G_0 of a nondeleting (typed) graph grammar $GG = (TG, P, G_0)$ with injective matches and self-disabling productions, each production $p \in P$ with $r : L \to R$ can be applied at most once with the same essential match $m_0 : L \to G_0$ where $m \models NAC(n)$.*

1.6 Negative Application Conditions

This section extends the previous theory of graph transformations to facilitate Negative Application Conditions (NACs) allowing control over the applicability of rules. A NAC is connected to either the LHS or RHS of a production rule forming a pre or postcondition on the rule. If this pattern is found in the corresponding host graph, the production cannot be applied. We use NACs extensively both in our theoretical contributions and implementation.

Definition 1.6.1. ***(Negative Application Condition** [EEPT06]**)** A* negative application condition *or $NAC(n)$ on L is an* arbitrary morphism *$n : L \to N$. A morphism $g : L \to G$ satisfies $NAC(n)$ on L i.e. $g \models NAC(n)$ if and only if does not exists and injective $q : N \to G$ such that $q \circ n = g$.*



A set of NACs on L is denoted by $NAC_L = \{NAC(n_i) | i \in I\}$. A morphism $g : L \to G$ satisfies *NAC_L if and only if g satisfies all single NACs on L i.e. $g \models NAC(n_i) \forall i \in I$.*

Definition 1.6.2. ***(Production Rule with NACs)*** *A set of NACs NAC_L (resp. NAC_R) on L (resp. R) for a rule $p : L \xleftarrow{l} K \xrightarrow{r} R$ (with injective l and r) is called a* left *(resp.* right*) NAC on p. $NAC_p = (NAC_L, NAC_R)$ consisting of a set of left and a set of right NACs on p is called a* set of NACs on p. *A* rule (p, NAC_p) with NACs *is a rule with a set of NACs on p.*

To enable the creation of concurrent NACs (Def. 1.6.5, Thm. 1.6.3), we show the possibility of translating a postcondition to a precondition.

Definition 1.6.3. ***(Construction of Left From Right NACs** [EEPT06]) For each $NAC(n_i)$ on R with $n_i : R \to N_i$ of a rule $p = (L \leftarrow K \rightarrow R)$, the equivalent left application condition $L_p(NAC(n_i))$ is defined in the following way:*

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow{\scriptstyle n_i'} & (1) & \downarrow & (2) & \downarrow{\scriptstyle n_i} \\ N_i' & \longleftarrow & Z & \longrightarrow & N_i \end{array}$$

- *If the pair $(K \to R,\ R \to N_i)$ has a pushout complement, we construct $(K \to Z,\ Z \to N_i)$ as the pushout complement (1). Then we construct pushout (2) with the morphism $n_i' : L \to N_i'$. Now we define $L_p(NAC(n_i)) = NAC(n_i')$.*
- *If the pair $(K \to R,\ R \to N_i)$ does not have a pushout complement, we define $L_p(NAC(n_i)) = true$.*

For each set of NACs on R, $NAC_R = \cup_{i \in I} NAC(n_i)$ we define the following set of left NACs:

$$L_p(NAC_R) = \cup_{i \in I'} L_p(NAC(n_i'))$$

with $i \in I'$ if and only if the pair $(K \to R,\ R \to N_i)$ has a pushout complement.

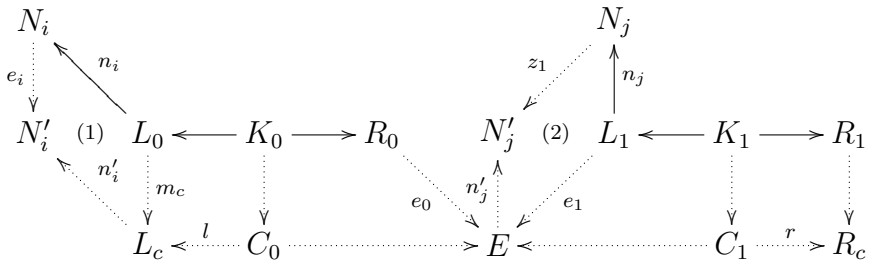
The following theorem establishes that a precondition is equivalent to the corresponding postcondition.

Lemma 1.6.1. ***(Equivalence of Left and Right NACs** [EEPT06]) For every rule p with NAC_R a set of right NACs on p, $L_p(NAC_R)$ as defined in 1.6.3 is a set of left NACs on p such that for all direct transformations $G \overset{p,g}{\Rightarrow} H$ with comatch h,*

$$g \models L_p(NAC_R) \Leftrightarrow h \models NAC_R$$

1.6.1 Concurrency with NACs

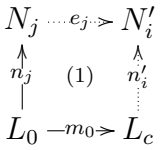
The theory of concurrency and extension introduced in Section 1.5 needs to be updated to facilitate NACs. In this section we discuss concurrency, while extension theory is treated in Section 1.6.2.



Our aim is to construct the resultant NAC of the E-concurrent production $p_1 *_E p_2$ translated from the E-related rules p_1 and p_2 with NACs $NAC(n_i)$ on L_0 and $NAC(n_j)$ on L_1. The concurrent NAC is denoted by NAC_{p_c}

First, we show the construction that translates $NAC(n_j)$ on L_0 to an equivalent NAC that is on L_c.

Definition 1.6.4. ***(Construction of NACS on*** L_c ***from NACs on*** L_0 *[LEPO08b])* *Consider the following diagram:*



For each $NAC(n_j)$ *on* L_0 *with* $n_j : L_0 \to N_j$ *and* $m_0 : L_0 \to L_c$ *let*

$$D_{m_0}(NAC(n_j)) = \{NAC(n'_i) \mid i \in I, n'_i : L_c \to N'_i\}$$

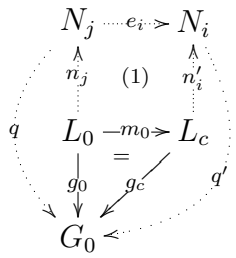
Where I *and* n'_i *are constructed as follows:* $i \in I$ *if and only if* (e_i, n'_i) *with* $e_i : N_j \to N'_i$ *jointly surjective,* $e_i \circ n_j = n'_j \circ m_0$ *and* e_i *is injective.*

For each set of NACs $NAC_{L_0} = \{NAC(n_j) \mid j \in J\}$ *on* L_0 *the downward translation of* NAC_{L_0} *is then defined as:*

$$D_{m_0}(NAC_{L_0}) = \cup_{j \in J} D_{m_0}(NAC(n_j))$$

The following theorem establishes that the translated $D_{m_0}(NAC_{L_0})$ is equivalent to the original $NAC(n_j)$.

Lemma 1.6.2. ***(Equivalence of Set of NACs on*** L_0 ***and Set of NACs on*** L_c *[LEPO08b])* *Given* $g_c : L_c \to G_0$, $m_0 : L_0 \to L_c$ *with* NAC_{L_0} *and* $g_0 = g_c \circ m_0$ *as in the following diagram:*

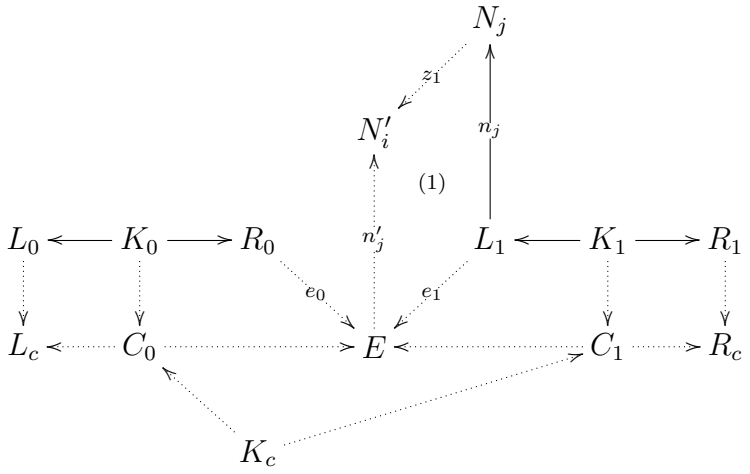


then the following holds:

$$g_0 \models NAC_{L_0} \Leftrightarrow g_c \models D_{m_0}(NAC_{L_0})$$

The second task is to define the equivalent NACs on the LHS of the concurrent rule from the set of NACs on the LHS of the second rule of the two-step transformation. This is done in two steps. First, a $NAC(n_j)$ on L_1 is translated to the $NAC_{n'_j}$ on E. Then, it can be seen that $L_c \leftarrow C_0 \rightarrow E$ is a production. Using the construction of left from right NACs (Def. 1.6.3) we create the equivalent NAC on L_c.

Definition 1.6.5. ***(Construction of NACs on*** L_c ***from NACs on*** L_1 *[LEPO08b])* *Given an E-dependency relation with jointly surjective morphism pair* (e_0, e_1) *for the rules* p_0,p_1 *and* $p_c = p_0 *_E p_1 = L_c \leftarrow K_c \rightarrow R_c$ *the E-concurrent rule of* p_0 *and* p_1 *is as depicted in the following diagram:*



For each $NAC(n_j)$ *on* L_1 *with* $n_j : L_1 \to N_j$*:*

$$DL_{p_c}(NAC(n_j)) = L_p(D_{e_1}(NAC(N_j)))$$

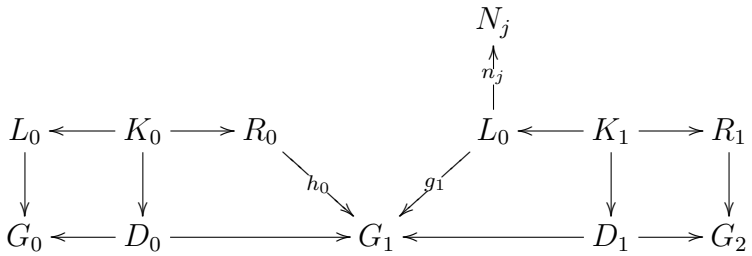
with $p : L_C \leftarrow C_0 \rightarrow E$ and D_{e_1}, L_p according to Definitions 1.6.4 and 1.6.3.

For each set of $NAC_{L_1} = \{NAC(n_j) \mid j \in J\}$ on L_1, the down and leftward translation of NAC_{L_1} is defined as:

$$DL_{p_c}(NAC_{L_1}) = \cup_{j \in J} DL_{p_c}(NAC(n_j))$$

The corresponding theorem shows that the construction is equivalent.

Lemma 1.6.3. ***(Equivalence of NACs on Rule*** p_1 ***and NACs on*** p_c *[LEPO08b])* *Given a two-step E-related transformation via $p_0 : L_0 \leftarrow K_0 \rightarrow R_0$ and $p_1 : L_1 \leftarrow K_1 \rightarrow R_1$*



with g_c being the match from the LHS of the E-concurrent rule $p_c = p_1 *_E p_2$ into G_0 (as described in the synthesis construction of Theorem 1.5.1) then the following holds:

$$g_1 \models NAC_{L_1} \Leftrightarrow g_c \models DL_{p_c}(NAC_{L_1})$$

The following theorem is the concurrency theorem extended to incorporate NACs.

Theorem 1.6.1. ***(Concurrency Theorem with NACs*** *[LEPO08b])*

1. Synthesis. *Given a transformation sequence $t : G_0 \overset{*}{\Rightarrow} G_{n+1}$ via a sequence of rules $p_0, p_1, ..., p_n$ with NACs, then there is a* synthesis construction *leading to the direct transformation $G_0 \Rightarrow G_{n+1}$ via the concurrent rule $p_c : L_c \leftarrow K_c \rightarrow R_c$ via NAC_{p_c}, match $g_c : L_c \rightarrow G_0$ and comatch $h_0 : R_c \rightarrow G_{n+1}$ induced by $t : G_0 \overset{*}{\Rightarrow} G_{n+1}$.*

2. Analysis. *Given a direct transformation $G'_0 \Rightarrow G'_{n+1}$ via the concurrent rule $p_c : L_c \leftarrow K_c \rightarrow R_c$ with NAC_{p_c} induced by $t : G_0 \overset{*}{\Rightarrow} G_{n+1}$ via a sequence of rules $p_0, p_1, ..., p_n$ then there is an* analysis construction *leading to a transformation sequence $t' : G'_0 \overset{*}{\Rightarrow} G'_{n+1}$ with NACs via $p_0, p_1, ..., p_n$.*

3. Bijective Correspondence. *The synthesis and analysis constructions are inverse to each other up to isomorphism.*

1.6.2 Extension with NACs

In this section we present the theory of extension with the presence of NACs in the participating production rules. First, we introduce the extension diagram with NACs.

Definition 1.6.6. ***(Extension Diagram with NACs** [LEPO08a])* *An* extension diagram with NACs *is a diagram as defined in Definition 1.5.3 except that the transformations t and t' use rules containing NACs. Thus, the matches $(m_0, ..., m_{n-1})$ and extended matches $(k_0 \circ m_0, ..., k_{n-1} \circ m_{n-1})$ have to satisfy the NACs of the rules $p_0, ..., p_{n-1}$*

Although the existence of the extension diagram (Thm. 1.5.2) needed only boundary consistency, we need another consistency condition when NACs are present. NAC-consistency intuitively requires that the concurrent NAC_{p_c} should be satisfied by the match of the concurrent production rule.

Definition 1.6.7. ***(NAC-Consistency** [LEPO08a])*

A morphism $k_0 : G_0 \to G'_0$ is called NAC-consistent *with respect to a transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ if $k_0 \circ g_c \models NAC_{p_c}$ with NAC_{p_c} the concurrent NAC and g_c the concurrent match induced by t.*

As the extension diagram with NACs and the NAC-consistency introduced, we can present the embedding theorem with NACs.

Theorem 1.6.2. ***(Embedding Theorem with NACs** [LEPO08a])* *Given a transformation $t : G_0 \overset{n}{\Rightarrow} G_n$ using rules with NACs. If $k_0 : G_0 \to G'_0$ is boundary consistent and NAC-consistent with respect to t then there exists an extension diagram with NACs over t and k_0 as defined in Def. 1.6.6 and depicted below.*

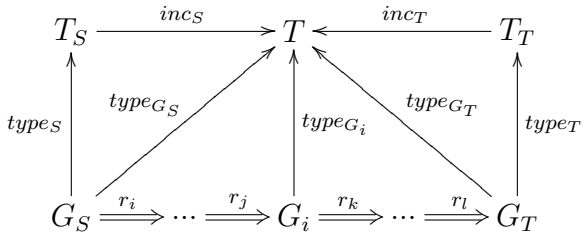
$$\begin{array}{ccc} G_0 & \overset{*}{\Rightarrow}_{t} & G_n \\ \downarrow{\scriptstyle k_0} & (1) & \downarrow{\scriptstyle k_n} \\ G'_0 & \overset{*}{\Rightarrow}_{t'} & G'_n \end{array}$$

1.7 Model Transformation Approaches

In this section we present two model transformation approaches, where the models are given by abstract syntax graphs. These two paradigms are the algebraic graph transformation (conventional) approach, and triple graph grammars. The conventional approach presents graph transformations as the imperative application of rules for transforming a source graph to a target graph. Triple graph grammars (TGGs) establishes bidirectional correspondence between participating graphs.

1.7.1 Based On Algebraic Graph Transformations

For the model transformation that is based on algebraic graph transformations, specific source and target models are given. Both models are described by their abstract syntax graphs. The abstract syntax graphs of the source models are specified by a subset of instance graphs over a type graph T_S. Similarly, the target models are specified with a type graph T_T. Both type graphs T_S and T_T are subgraphs of a common type graph T. As shown in the diagram below the containment is expressed as inclusion morphism $inc_S : T_S \to T$ and $inc_T : T_T \to T$.



The transformation is defined as a graph transformation system $GTS = (T, R)$ with the common type graph T and a set of transformation rules R (Def. 1.4.4). The start model is represented by G_S typed over T_S. The transformation proceeds by applying rules of R as described in Def. 1.4.5. The resultant graph G_T represents the target model. During the transformation process, intermediate graphs, like G_i, are typed over T containing elements from both T_S and T_T. In order to fulfill semantical correctness, the resultant graph G_T (target model) has to be typed over T_T.

The application of the production rules can be nondeterministic for two reasons. There can be several applicable rules, and one is chosen arbitrarily. Also, given a rule, it is possible to have multiple matches. Both kinds of nondeterminism can be restricted. Control flow can be introduced with negative application conditions, explicit control structures or priorities. The matches can be restricted by input parameters. And finally a rule can be selected from outside (e.g. the user).

1.7.2 Based on Triple Graph Grammars

In this section we present the basics of a different approach to graph transformations. Conventional graph transformations, introduced in Section 1.4.2 are restricted to transform one instance of a class of graphs into an instance of another class.

The concept of triple graph grammars (TGGs) [Sch94] is a purely declarative way of describing two-way graph transformations. A correspondence graph is used, that describes the connections between the other two graphs enabling the definition

of *m-to-n* relationships. These correspondence graphs and rules allow us to record additional information about the transformation process itself, which are for instance needed to propagate incremental updates of one data structure as incremental updates into its related data structures [Sch94].

Definition 1.7.1. ***(Graph Triples** [Sch94]) Let LG, RG, and CG be three graphs, and* $lr : CG \to LG$, $rr : CG \to RG$ *are those morphisms which represent* m-to-n *relationships between the left-hand side graph LG and the right-hand side graph RG via the correspondence graph CG in the following way:*

$$x \in LG \text{ is related to } y \in RG \Leftrightarrow \exists z \in CG : x = lr(z) \wedge rr(z) = y$$

The resulting graph triple *is denoted as follows:*

$$GT = (LG \xleftarrow{lr} CG \xrightarrow{rr} RG)$$

One must be aware, that the CG is not similar to the interface graph K of conventional graph transformations. The interface graph is an intersection of the left- and right-hand side while the correspondence graph expresses a common syntax and basis for expressing relations.

The advantage of triple graph grammars in comparison to conventional graph transformations is clearly visible. In the latter case we have a fixed and unintelligible graph on left- and right-hand sides of productions. A TGG, with the two sides are equivalent, replaces three different conventional graph transformation cases:

- a *left-to-right* transformation, which takes any left-hand side graph as input and returns a corresponding right-hand side graph
- a *right-to-left* transformation, which analyses a right-hand side graph and produces left-hand side if possible
- a *correspondence* analysis, which monitors the relationships between a given left-hand side and a given right-hand side by trying to establish correspondences between them.

Chapter 2

Related Work

The presentation of a problem on its own is not sufficient, its context is almost equally important. The reflection on related work is necessary as it positions the contribution. Thus, the present chapter satisfies this need. It is organised as follows: Section 2.1 examines research done on formal verification of graph and model transformations discussing the verification of temporal properties. Compositionality of bidirectional transformations is analysed in Section 2.2. Then a short taxonomy of refactoring approaches is presented in Section 2.3, succeeded by a detailed discussion of these approaches.

2.1 Verification of Graph and Model Transformations

Our contributions are on the field of model transformations and their verification. Although there are several research areas of model transformation verification, we would like to concentrate on those that verify temporal properties.

There are behavioural properties of graph transformations that can be described as temporal properties. For instance, termination — required to show the functional behaviour of the proposed semantic mapping (Sec. 6.6.4) — can be described by temporal properties. The verification of a system with respect to temporal properties is often similar to the setup of our approach: it is based on the translation of the system to a formal model that can be checked against a logic formula by an automatic tool (a model-checker). Alternatively, systems can be formalised in a logic and verified by carrying out formal proofs, either automatically or interactively, using a theorem prover.

Automated verification, for all the crucial role it may play in the formal analysis of dynamically evolving systems, appears to have been covered until recently by a comparatively small section of the otherwise very rich literature on GT systems. A few major early contributions in the static analysis of such systems are [Koc00, GHK98, Hec98a]. In [Hec98b] the author introduced an abstract theory of over-approximations (system views) based on graph transformations, aiming at the verification of reactive systems, which relies on the interpretation of a branching time logic built into models as GT systems of the double pushout approach.

Since then, several approaches based on model-checking have been introduced. Interesting comparisons between some of the main trends can be found in [BKR05] — between the unfolding and the partitioning approach, and [RSV04] — between the GROOVE-based and the SPIN-based approach. Research has largely been focused on abstraction techniques to cover large and infinite models [BK02, BCK08, KK06, Ren04a, Ren04c, BBER08, Ren08], allowing more expressive systems (e.g. with types and attributes) [KK08, Kas05, Var04, HLM06], and on making state-of-the-art generic model-checking techniques available for GTS verification [SV03, Var04, DLRdS03, DMdS05, FFR07].

The concurrent behaviour of GT systems has been studied in depth throughout the years, leading by now to a consolidated theory of concurrency which generalises the corresponding semantics of Petri nets, including process and unfolding semantics (see, e.g., [CMR96, Rib96, BCM98, BCMR07]).

Several methods have been successfully proposed for the analysis of Petri nets, ranging from the calculus of invariants [Rei85] to model checking based on finite complete prefixes [McM93, Esp94]. Some of these methods, most notably the one originally proposed by McMillan in [McM93], are based on the concurrent semantics of nets, which allows to avoid the combinatorial explosion arising when one explores all possible interleavings of concurrent events.

The unfolding approach to GTS model-checking [BCM99, BK02, BKK03, BCK04, BCK08] [KK06, KK08] relies on a translation of GT systems to models based on Petri nets and on the application of an unfolding strategy. This is also the underlying approach in a counterexample-guided abstraction refinement technique implemented in [KK06], more recently extended to support attributed graphs [KK08].

The model-checking approach presented in [Ren03, KR06, Ren08] is largely based on graph transformation techniques as they have been implemented in the GROOVE tool [Ren04b]. Given a GTS, GROOVE can generate its state space and convert it to a Kripke model to be checked against temporal logic formulas [KR06]. On top of

this approach, abstract interpretation techniques based on notions of abstract graph and abstract graph transformation have been investigated in order to deal with large systems. An abstraction approach related to shape analysis and based on structural similarity between nodes of the state graphs has been developed in [Ren04a, Ren04c]. The goal of this approach, also called *partitioning*, is to obtain tractable models based on a logically intuitive notion of abstraction. In a further development [BBER08], shape-based abstraction has been joined by a form of topological abstraction, based on adjacency relations. The combined abstractions have been shown to satisfy preservation and reflection with respect to a modal logic.

Verification of GT systems can also be carried out relying on more standard model-checking techniques [SV03, Var04]. The CheckVML tool presented in [SV03] aims at the verification of arbitrary visual modelling languages, and it can be used to generate a model-level specification of a GTS in the form of a Promela description for the SPIN model checker. SPIN can be used to check models against LTL formulae. In contrast with the GROOVE approach, the model translation is already optimised — e.g. it tends to abstract static elements away, in order to avoid state explosion [Var04]. Another line of research based on translation to Promela and model-checking with SPIN can be found in [DLRdS03, DMdS05, FFR07].

The authors of [BGMM08] rely on standard SAT model-checking in order to carry out automatically the verification of behaviour models in which graph transformation is used to represent data abstraction. AGG is used to generate a GTS model that gets translated to a linear temporal logic model, and this can be checked by a SAT-solver, relying on a bound model-checking approach.

2.2 Compositionality of Bidirectional Transformations

Although the design of our semantic mapping is inspired by Triple Graph Grammars (Sec. 5.1.2), it is not bidirectional. However, *spatial compositionality* of bidirectional transformations, discussed in [Ste07, Ste08], is similar to the notion of compositionality proposed in this thesis.

The aim of bidirectional transformations is to maintain consistency [Ste08]. The setting is truly symmetric: there are no distinct source or target models. When either of the models change, a transformation is ran to restore consistency.

Spatial composition yields that if two systems are composed of parts which them-

selves can be acted on by model transformations, consistency of individual parts implies the consistency between the systems. Given a system m composed of parts $c_i, i \in I$ and s composed of parts $t_j, j \in J$ and a consistency relation $\mathcal{R}$, the consistency $\mathcal{R}(m, s)$ can be *demonstrated* by showing that for every c_i, there is a corresponding t_j, such that $\mathcal{R}(c_i, t_j)$. Thus, we can understand the effect of a transformation on the composed system by understanding its effect on the parts composed [Ste08].

In our case, there are two models: the semantic domain N and the software artifacts N. As the semantic domain only formalises the behaviour of the software artifact, it is not modified directly. Only the software artifacts may change, thus the transformation only implements a function $sem : M \to N$. Explicit consistency checking is not necessary; the software artifact m and the formalised behaviour m of the chosen semantic domain are consistent if $n = sem(m)$. If the software artifact is changed, the transformation is re-run, and the old semantic model is replaced by the newly generated version.

2.3 Refactoring

To discuss the related work on refactorings systematically, the following list summarises the most important properties of a refactoring approach.

- *Subject language*: either programming language (procedural languages like C, object oriented like C++ and Java) or modelling language (UML diagrams or constraint languages like Object Constraint Language (OCL) [OMG06a]).
- *Representation of refactoring*: informal (based on language syntax, described as a pattern) or formal (graph transformation rules or other formal notation)
- *Representation of semantics*: informal (based on syntax), invariants (predicates that have to be true before and after the refactoring), formal (process algebras or graph-based operational semantics)
- *Verification method*: assumed (behaviour preservation is assumed or based on empirical evidence), informal (regression testing or other informal reasoning), formal (verification of bisimulation or trace refinement with tool support)

Refactoring approaches are easily classified according to the above categories. Based on empirical observations, most of the approaches can be grouped into two distinct methodologies: the *pattern-based* and the *verification-based*. All approaches

in the same methodology share a common setup of the above ingredients, the actual implementation and language refactored can differ.

The *pattern-based* approaches like [MTR07, SPTJ01, HJvE06, PC07] assume a large set of refactoring patterns. The application of the patterns are assumed (but not formally verified) to preserve behaviour, and thus the overall change is supposed to be behaviour preserving as well. A pattern-based refactoring consists of three important steps: the recognition for the need of refactoring, the application of the refactoring pattern that is shown to be behaviour preserving; and the assessment of the system if the refactoring really improved it.

In *verification-based* approaches [vKCKB05, RLK+08], the refactoring is performed by the developers. In the *pattern-based* approach, the developers merely applied a pattern; here, an actual development process is performed. When the necessary improvement is finished, the system is checked for behaviour preservation by formal means. Note that these methodologies are not complementary: one may verify the behaviour preservation after pattern application. It is rarely done however.

In the following sections, we discuss the different refactoring approaches one by one, organised by their notion of behaviour. Approaches where formal behaviour is not defined and the behaviour preservation is based on empirical evidence or various invariants are detailed in Section 2.3.1. In Section 2.3.2 approaches with formal behaviour are discussed. Finally, Section 2.3.3 summarises all the presented approaches in a concise table.

2.3.1 Invariants and Empirical Evidence

This section deals with work done on refactorings without formal behaviour. The presentation is not chronological, it goes along the subject language the refactoring is performed on: object-oriented languages, then procedural programming and finally modelling languages.

The notion of refactoring was originally introduced by Opdyke [Opd92] focusing on object-oriented languages. Refactorings are proposed to be intermediate level reorganisation plans, and as such, the representation of their semantics is based on regression testing: before and after refactoring, a program has to produce the same output for a given set of inputs. He identifies a set of *invariants* that preserves the behaviour of refactorings. The concept of invariants is inspired by the works of Banerjee and Kim [BKKK87] on database schema evolutions. Opdyke proved that the presented refactorings preserve the invariants. However, the preservation invariant

does not necessarily imply preservation of behaviour.

Instead of invariants, Roberts [Rob99] uses pre- and postconditions described as first order predicates. This way, it is possible to calculate the applicability of refactorings in a refactoring sequence. Given a composite refactoring containing a refactoring sequence, Ó Cinnéide and Nixon[OCN98] propose an algorithm to calculate the pre- and postconditions of such composite refactorings from the predicates of all their refactorings contained. In both cases, behaviour preservation is proved only by showing that a refactoring fulfills its pre- and postconditions.

Tokuda and Batory [TB01] implemented the refactorings proposed by Opdyke [Opd92] and Banerjee and Kim [BKKK87] in C++. They define refactorings as parameterised behaviour preserving program transformations and point out that formal proof of behaviour preservation is desirable but unlikely. The complexity of object oriented languages makes the definition of formal semantics challenging; semantics may vary between compilers or language versions. Their notion of semantics consists in the introduction of *enabling conditions* that are invariants defined for a specific refactoring. According to their results, these conditions help to preserve behaviour, however this may not be sufficient. Behaviour preservation is due to good implementation, not formal proofs.

The refactoring textbook by Fowler [FBB+99] presents refactorings like design patterns: each refactoring is presented with its name, short summary, motivation for use, guide on applying the refactoring and an example in Java. Although this presentation is not formal, there is a definite progress towards a standardisation. An explicit notion of behaviour or verification of its preservation is missing.

There are refactoring approaches for non object-oriented languages as well. Garrido and Johnson [GJ02] worked on refactorings in C. The problems of this different paradigm were addressed. A catalog of refactorings was introduced and implemented in a prototype tool. The notion of behaviour is regression testing, as proposed by Opdyke [Opd92]. Vittek [Vit03] created a refactoring browser for C and C++, and also addressed the problem of preprocessor directives.

There are other alternative notions of behaviour preservation as well. Tip *et al* [TKB03] introduced type constraints formally, but argue informally about preservation of type correctness. Bergstein [Ber91] uses the notion of *object-preservation*. In object-oriented database refactorings, the repopulation of a database is objectionable. Object preservation thus means that the set of objects that the class defines is not changed.

Bottoni *et al* [BPPT04, BPPT03] widen the scope of refactoring by using dis-

tributed graph transformations. Their aim is to maintain consistency between the various software artifacts before and after refactoring. The software system (including code, class diagrams, sequence diagrams and state charts as well) is represented by distributed graphs. A distributed graph is basically a network graph that has composite nodes containing object graphs. The system is represented by this network graph with the nodes being the different views. Source code is modelled as a control-flow graph. They use distributed graph transformation rules as a technique or mechanism to perform refactorings, their representation of refactorings is based on the pre- and postcondition approach of Roberts [Rob99]. The left-hand side is a precondition that has to be satisfied, and the right-hand side is the postcondition that describes the effects of the refactoring. Notions of semantics and behaviour preservation are not mentioned explicitly.

A notable refactoring approach based entirely on UML models is of Sunyé *et al* [SPTJ01]. Refactorings are defined on UML class diagrams and state charts. Although the representation of a refactoring is a text-based, informal description, the preservation of behaviour is expressed by OCL invariants. The preservation of invariants is justified by empirical evidence.

Porres [Por03] also presents a method for rule-based UML refactorings. He uses the System Modelling Workbench (SMW) [Por02] for rule representation. SMW uses a languge for UML modification, with a syntax similar to OCL. Refactoring rules consist of two parts: the precondition and the actions that define the refactoring. He also has an implementation for the Python language. The formal notion of behaviour and its preservation are not discussed however.

Another area of refactoring arises when UML class diagrams are annotated with OCL constraints. After a refactoring, the OCL expressions need to be updated as well. Also, it is possible that the OCL expressions themselves need refactoring. Correa and Werner [CW04] present OCL-based refactoring patterns. They address both problems: presenting OCL exclusive and combined refactoring patterns. A graph-grammar inspired approach is presented by Markovic and Baar [MB08], in which the formal notation is given by QVT Merge [QVT05]. They present a catalog of OCL annotated class diagram refactorings. The verification of behaviour preservation in both approaches is case by case reasoning.

Biermann *et al* [BEK+06] present refactorings on EMF models. The refactoring rules are represented as graphs, and implemented in the Eclipse Modelling Framework. They also provide consistency check for refactorings, but no formal notion of behaviour. Taentzer *et al.* [TM07] present domain-specific refactorings represented

as EMF model transformations.

The first to employ graph transformations were Mens *et al* [MDJ02]. In their opinion, refactorings should be as generic as possible, thus a graph-based representation was created, in which refactoring rules are represented by graph transformation rules. However, as refactoring tools work on source-code, the authors restrict refactorings to be performed only on the static structure of the program. Hence three structural invariants are defined formally: *access preservation*, *update preservation* and *call preservation*. The method of verification is not mentioned.

When several refactorings are applicable to a system, the works of Taentzer *et al* [MTR07] help to make an informed choice. Applying one refactoring may prohibit the application of other refactorings. Thus, by representing the refactorings as graph transformation rules, sequential dependency analysis and critical pair analysis can be used. This way, the the implicit dependencies among refactorings can be detected, and the sequence of refactorings most suitable in a given context can be uncovered.

As the use of graph transformation to express visual languages and their refactorings gained popularity, shortcomings of graph transformations were addressed by Hoffmann *et al* [HJvE06]. Although refactoring patterns tend to be generic, a graph transformation rule represents a refactoring on a specific system instance. A generic refactoring pattern may correspond to a large set of very similar graph transformation rules. The authors of [HJvE06] point out that different graph transformations are needed for each method body in the *push-down-method* [FBB+99] refactoring. Thus, in [HJvE06, vEJ04, Hof05] the expressive power of graph transformation rules is augmented. *Cloning* and *expanding* operations are added to graph variables. This way, the graph transformation rules are turned into a rule scheme. Thus, a rule scheme is a suitable formal representation of a refactoring pattern. When applied to a system, the rule schemes are instantiated. Janssens and Van Eetvelde [EJ03] address the size problem of the graph representation of programs. The graph representation of a simple class can be enormous. Thus, a hierarchical structure for the graphs that represent programs is introduced. It enables the developers to view the system at the appropriate abstraction level.

Pérez and Crespo [PC07] present a method to detect whether the evolution between two versions of a system can be expressed as a sequence of refactoring patterns. Refactorings — formally presented as graph transformations following Mens *et al* [MDJ02]— are performed on Java programs. A graph parsing algorithm based on state-space search is implemented in AGG to find the suitable refactoring sequence.

In all the previous refactoring approaches, it was assumed that the choice of the

refactoring to apply is always made by the developer. Van Gorp *et al* [vGSMD03] propose a solution for automating this choice. The refactoring is on the level of UML models using OCL for expressing pre- and postconditions. However, the pre- and postcondition approach is improved by the introduction of *refactoring contracts*: a third set of OCL expressions is added to capture the problematic model constructs that can be improved by that refactoring. Hence, by searching the model, it is possible to identify problematic parts and apply the relevant refactorings automatically. Neither the preservation of invariants nor the representation of refactorings are explicit; the approach focuses on the automation.

Massoni *et al* [MGB06] extend invariants to support automatic refactorings. The invariants are expressed in first-order logic, based on the Analysable Annotation Language (AAL) [KMJ02] and implemented as Java code annotations. The invariants are not only expressions that provide behaviour-preservation when used but also preconditions providing semantic information about the classes and their fields. This helps the refactoring tool to select the appropriate refactoring pattern automatically.

2.3.2 Formal Behaviour

After the exhaustive list of informal approaches, this section presents works with formal notion of behaviour. Since there are fewer of such approaches, they are presented in greater detail following the chronological order of release.

Refactoring of architectural models has been studied formally first in architectural description languages (ADLs) like Wright [ADG98, All97] or Darwin [MK96], using process calculi like CSP or π-calculus for expressing formal semantics.

Refinement of Information Flow Architectures Philipps and Rumpe [PR97] present a calculus for stepwise refinement of abstract information flow architectures. These architectures are represented as a hierarchical data-flow network. The authors use the suggestive box-and-arrow notation [YC79] for graphical representation.

The formal calculus proposed in the paper is based on [BDD$^+$92]. CSP is a completely generic process algebra, while this algebra incorporates properties of the software architecture. The central notion is a set $\mathbb{C}$ of channel identifiers with a given set M of messages. The calculus uses *streams* to describe communication sequences on a channel, which are similar to *traces* in CSP. The difference however is that the notion of time is used. M^* denotes a finite sequence and over a set M of of messages; M^∞ is a set of infinite sequences over M. The communication histories are *timed streams*, i.e. $M^{\mathfrak{N}} = (M^*)^\infty$. The time axis is divided into an infinite stream of time intervals,

where each interval consists a finite number of transmitted messages. Channels and timed streams are assigned by a *named stream tuple* function ($\mathbb{C} \to M^{\mathfrak{N}}$) and $\overrightarrow{C}$ is a set of named stream tuples within $C \in \mathbb{C}$.

All other objects in the calculus are derived from these basic notions. Component behaviour is modelled as a relation over input and output communication histories. Given input channels $I \in \mathbb{C}$ and output channels $O \in \mathbb{C}$ the interface behaviour of a component is a function $\beta = \overrightarrow{I} \to \mathbb{P}(\overrightarrow{O})$. Component is a tuple $c = (n, I, O, \beta)$ with n as component name. The operators *name.c*, *in.c*, *out.c* and *behav.c* yield n, I, O and β respectively.

As shown, both the modelling language and its semantics are different to the ones we use. The box-and-arrow notation is restrictive compared to CSML. At the level of semantics, the notion of streams is similar to the trace semantics of CSP, but the time axis, and the channel-based structuring makes the proposed calculus more specific.

The authors introduce three different notions for behavioural refinement: basic behavioural refinement, refinement of systems and refinement with invariants. Basic behavioural refinement is essentially a refinement relation defined on behaviours, i,e given behaviours $\beta_1, \beta_2 \in \overrightarrow{I} \to \mathbb{P}(\overrightarrow{O})$ we say β_1 is refined by β_2 if and only if $\forall i \in \overrightarrow{I} : \beta_2(i) \subseteq \beta_1(i)$. Behaviour refinement of systems uses the notion of black box behaviour, i.e. $[S] : \overrightarrow{in.S} \to \mathbb{P}(\overrightarrow{out.S})$. With the black box behaviour, systems can be treated as components. Their refinement relation is a behavioural refinement on the given interface, i.e. $S \rightsquigarrow S' \Leftrightarrow \forall i \in \overrightarrow{in.S} : [S'](i) \subseteq [S](i)$ ($in.S = in.S'$ and $out.S = out.S'$ was assumed). The idea of both refinement relations is similar to trace refinement in CSP, but tailored to the special constructs of the calculus.

The refinement with invariants is different however. An invariant over the possible message flows within a system $S = (I, O, C)$ is given as a predicate Ψ over all streams within the system: $\Psi : \overrightarrow{(I \cup out.C)} \to \mathbb{B}$. An invariant is valid within a system, if it holds for all named stream tuples l defining the streams of the system. If the invariant is valid, then the behavioural change was a refactoring.

A prototypical CASE tool, AutoFocus [HSSS96] is used for verification.

Although we compared the modelling language, its semantics and the refinement notions between the two approaches, the main difference is flexibility. In our approach the modelling language and semantic domain are examples and thus are interchangeable, the authors main contribution is the calculus for that specific modelling notation. Also, the verification is not dealing with compositionality: always the complete behaviour of the system is verified.

Behaviour Preservation of UML Models Van Kempen et al [vKCKB05] also proposed to use CSP to formally describe the behaviour of UML models. The authors use a class diagram variant to model software structure. The classes may contain attributes, but only the operations are concerned. Two classes are connected if there is a call relationship between them, i.e. class A has an association with class B if A calls a method of B. The owned behaviour of the classes are specified in terms of statecharts: every class has a statechart defining its behaviour. Statecharts model a method call by a *Call Event* and the return call by a *Signal Event*. The execution of a method in a class is either implied by the semantics of the *Call Event* or defined using a statechart.

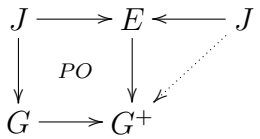
The behaviour expressed in statecharts are formalised by CSP. The mapping of statecharts to CSP is not formal, in [vKCKB05] it was described informally. The idea of the mapping is that a class A will correspond to a process P with its functions calls and state changes as events. Refactoring is tested as trace refinement between the old and the new system.

This approach uses similar concepts to ours: UML based modelling language (class diagram with statecharts compared to component diagrams with activities) and CSP as denotational semantics and trace refinement as refinement relation. However, as pointed out previously, our approach is generic; the connection between class diagrams and CSP could be formalised with graph transformations to form another example to our theoretical contributions. Also, compositionality is not concerned.

Behaviour Preservation using Borrowed Contexts The only approach that is comparable to ours is that of Rangel *et al* [RLK+08, RKE07]. Not only it is generic with respect to modelling domain, but verification is performed at the level of refactoring rules.

First, let us review the technical machinery they employ to verify refactoring rules. In the standard DPO approach (Sec. 1.4.2), the productions rewrite graphs with no interaction with any other entity then the graph itself. Hence, the authors adopted the DPO approach with borrowed contexts [EK06], where graphs have interfaces, through which the missing parts of the LHS can be borrowed from the environment. This leads to open systems which take into account interactions with the external environment [RLK+08]. A graph G with interface J is basically a morphism $J \to G$. The interface contains the elements that communicate with the envionment, i.e. they function as boundary points. The context is a pair of morphisms $J \to E \leftarrow \overline{J}$. The pushout below constructs the embedding of graph with interface $J \to G$ into a

context $J \to E \leftarrow \overline{J}$.



The authors define *rewriting rules with borrowed contexts* based on graphs with interfaces and graph contexts. A rewriting step with borrowed context is notated as $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$ over a graph context $J \to F \leftarrow K$.

While we map denotational semantics that can be represented as graphs to the modelling domain, the authors use operational semantics directly on graph-based modelling domains. Operational semantics defined for a type graph TG is a set of graph productions $OpSem^{TG}$. Then, the notion of behaviour preservation is bisimulation, i.e. a relation $\mathcal{R}$ is called a *bisimulation* if whenever there is a $(J \to G)\mathcal{R}(J \to G')$ and a transition $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$ then there exists a model $K \to H'$ and a transition $(J \to G') \xrightarrow{J \to F \leftarrow K} (K \to H')$ such that $(K \to K)\mathcal{R}(K \to H')$. The advantage of the borrowed context technique is that the derived bisimulation is a congruence and thus it is preserved by embedding into contexts. This notion corresponds to compositionality in our approach.

As can be seen, the theoretical background that achieves rule-level verification of behaviour preservation is completely different. Although both approaches are flexible with respect to the actual modelling and semantic domain, harnessing this flexibility raises different issues. In our case the semantic mapping has to be adapted on changing any of the domains. In the operational semantic approach one has to create a complete set of operational semantic rules for every modelling domain.

It is indeed true, that this approach works directly with graph transformations, and thus do not need the auxiliary encoding like ours. However, their lack of tool support gives us the practical advantage. We successfully applied our theory to practice. While their case study is used to illustrate their theoretical contributions, our implementation demonstrates the validity of ours.

As it can be seen, none of the above approaches fulfill our requirements of having formal representation of refactorings and behaviour with verifying refactorings at rule-level using proper tool support on real-world examples. Thus, we present our approach, which complies with these requirements.

Approach	Language	Representation	Semantics	Verification method
[Opd92]	programming	informal	invariants	informal
[BKKK87]	modelling	informal	invariants	informal
[Rob99]	programming	informal	invariants	formal
[TB01]	programming	informal	invariants	assumed
[FBB+99]	programming	informal	informal	assumed
[GJ02]	programming	informal	invariants	informal
[TKB03]	programming	informal	informal	informal
[Ber91]	programming	informal	informal	informal
[MDJ02]	programming	formal	informal	informal
[BPPT04]	modelling	informal	invariants	assumed
[SPTJ01]	modelling	informal	informal	informal
[Por03]	modelling	formal	informal	assumed
[CW04]	modelling	informal	informal	informal
[MB08]	modelling	formal	informal	informal
[BEK+06]	modelling	formal	informal	assumed
[MGB06]	modelling	formal	informal	assumed
[vKCKB05]	modelling	informal	formal	formal
[PR97]	modelling	informal	formal	formal
[RLK+08]	modelling	formal	formal	formal
Ours	modelling	formal	formal	formal

Table 2.1: Summary of Refactoring Approaches

2.3.3 Summary of Approaches

In Table 2.1, all the approaches presented in the previous sections are summarised. Note that [vGSMD03, EJ03, PC07, MTR07] generally follow the methodology of Mens *et al* [MDJ02], and are thus not explicitly highlighted.

Chapter 3

Modelling Architecture

The theoretical contributions of the thesis use software architecture models as an example application domain. Our choice for architecture modelling language is based on the UML logical view: it shows *how the functionality is designed inside the system, in terms of the system's static structure and dynamic behaviour* [EPLF03].

Combined Structure Modelling Language (CSML) — the architecture description language we present in this chapter — comprises three different views of the system: a type-level architectural view that defines the system as a static blueprint, a behavioural view that describes the dynamic behaviour of the system and an instance-level view that presents the actual running configuration.

For type-level representation we use the *Components* package of UML2. The central concept which forms the building block is the component, a replaceable modular unit. Activity modelling from the *FundamentalActivities* package has been chosen for its usability as a workflow modelling language in the context of service-oriented systems. Although the notion of activity partitions is introduced in [OMG06b], it violates the principle of encapsulation. Activities are associated with components (type-level) as owned behaviour. Similar to classes and objects in object-oriented programming, the behaviour of components is shared between all instances of the same type. While the type-level is a static description of the system behaviour, the instance level presents the actual running configuration. Since they show the collaborating features of the system as they combine at run time, the packages covered by the *composite structure diagram* were chosen for our work. *Combined Structure Modelling Language* merges the constructs defined in all these packages (type-level, behaviour and instance-level). To provide a precise definition, this chapter introduces the metamodel of CSML in required detail.

CSML is illustrated with the help of an example based on the *Car Accident Sce-*

nario from the SENSORIA Automotive Case Study [WCG+06]. The *Car Accident Scenario* is concerned with automatic dispatch of medical assistance to road traffic accident victims. The service assumes a car being equipped with GPS-based location tracking devices with communication capabilities and a pre-registered mobile phone of the driver. The occurrence of a road accident is detected by various sensors, and reported to an on-board safety system (*car agent*). The safety system immediately sends the current location of the car (obtained by GPS) to a pre-established accident report endpoint (*accident server*). The accident server attempts to call the registered mobile phone of the driver and analyses the sensorial data acquired from the car. If there is no answer to the call, emergency services close to the reported location of the car are contacted (*local services*) and asked for immediate dispatch (presuming that the driver has been incapacitated by injuries sustained in the accident). From the general scenario, our work elaborates on the *accident server*.

The outline of the chapter is as follows. Section 3.1 gives an overview of the metamodel. Then, a detailed presentation follows according to the original packages that the metamodel encompasses: Section 3.2 presents the components, Section 3.3 the activities, and Section 3.4 the composite structures. A diagrammatic representation for CSML with necessary examples are introduced in Section 3.5. A detailed comparison between the official UML2 metamodel and CSML metamodel is provided in Section 3.6.

3.1 Metamodel

We represent our model as instances of metamodels represented by attributed typed-graphs. A metamodel, as mentioned in Section 1.1, consists of an *explicit model* of the constructs and a set of well-formedness constraints defined on them. The *explicit model*, which is the type graph TG_{arch} is shown in Figure 3.1. It is based on the official UML2 metamodel with only the necessary model elements kept from the component, composite structure and activity diagrams [OMG06b].

We introduce the model elements in the metamodel grouped according to their original diagram affiliation (i.e. *ActivityNode* originally belongs to the Activity Diagram, consequently it will be presented in Section 3.3 that deals with the *Activities* package). As the only exception is the *ComponentContainer* class, we present it here.

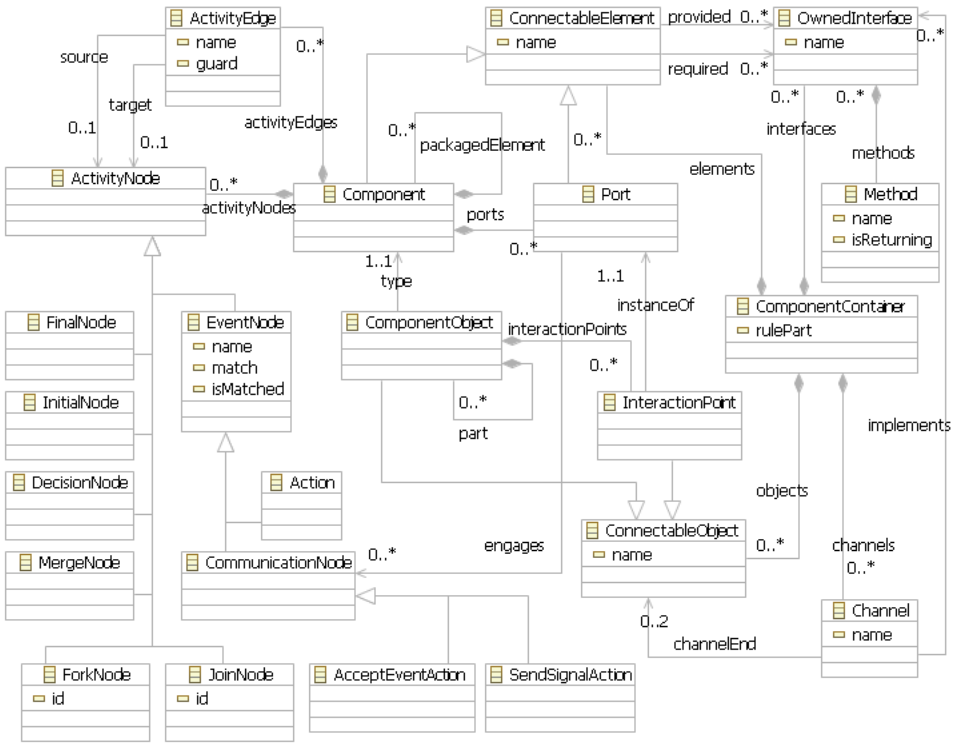


Figure 3.1: TG_{arch}: the combined structure metamodel

ComponentContainer

The *ComponentContainer* is a central element of the metamodel. The implementation uses EMF *ecore* [EMF07] which stores the model as an XML tree. A root node is necessary that contains all elements; the *ComponentContainer* is this root.

Attributes

- *rulePart: String [0..1]*: One instance of the metamodel represents one typed graph. The *rulePart* attribute is the name of this graph (e.g. L, R or G).

Associations

- *elements [0..*]*: A collection of *ConnectableElements* owned by the *ComponentContainer*.
- *interfaces [0..*]*: A collection of *OwnedInterfaces* owned by the *ComponentContainer*.
- *objects [0..*]*: A collection of *ConnectableObjects* owned by the *ComponentContainer*.

- *channels [0..*]*: A collection of *Channels* owned by the *ComponentContainer*.

Constraints

1. There can be only one *ComponentContainer*.
   self.allInstances()->size = 1

2. The *Components*, *Interfaces*, and the *ActivityEdges* and *EventNodes* contained by the *ComponentContainer* must have unique names. This is not the trivial requirement that every *Component* has to be distinguishable; but rather that a *Component* cannot bear the same name as an *ActivityEdge*, *ActivityNode* or *Interface* and vice versa.
   context ComponentContainer:
   elements->forAll(oclIsKindOf(Component)).activityNodes->
   forall(oclIsKindOf(EventNode)).name->
   intersection(elements->
   forAll(oclIsKindOf(Component)).activityEdges.name->
   intersection(interfaces.name->intersection(elements.name)))->isEmpty()

3.2 Components

The *Components* package helps to describe the types of components in the system together with their provided and required interfaces [OMG06b]. The components part of the metamodel is shown in Figure 3.2. The diagram associated with the *Components* package is the *component diagram*. The component diagram of the accident server is illustrated in Figure 3.3. The classes we use are the following, based on [OMG06b].

ConnectableElement

Used according to the *Composite* pattern [BMR+96], the *ConnectableElement* is a generic parent for the type-level classes. A *ConnectableElement* is typed by its provided and required interfaces as a specification of the services it provides to its clients and those that it requires from other *ConnectableElements*.

Attributes

- *name: String [1]*: The name of the *ConnectableElement*.

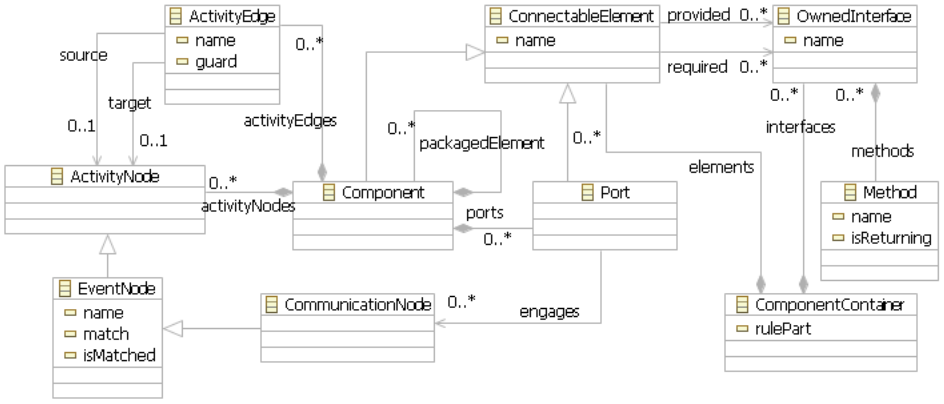


Figure 3.2: Components metamodel

Associations

- *provided [0..*]*: The interfaces that the *ConnectableElement* exposes to its environment.
- *required [0..*]*: The interfaces that the *ConnectableElement* requires from other *ConnectableElements* in its environment in order to provide its full functionality.

Constraints

1. *ConnectableElements* have unique names.
   context ConnectableElement **inv**:
   ConnectableElement.allInstances()->
   forall(p1,p2 | p1 <> p2 **implies** p1.name <> p2.name)

Component

Each *Component* is a replaceable modular unit of the system with encapsulated implementation of data and behaviour. A *Component* specifies a formal contract of the services that it provides and those that it requires in terms of its provided and required interfaces. Every *Component* type is substitutable with other *Component* types based on the compatibility of interfaces [OMG06b].

In the component diagram, *Components* are represented by rectangles with a component icon and a classifier name. *Components* may contain other components as well as ports.

Generalisations

- ConnectableElement

Attributes No additional attributes.

Associations

- *packagedElement [0..*]:* *Components* may contain other *Components*. This is the collection of the owned *Components*.
- *ports [0..*]:* A *Component* may have a set of *Ports* that formalise its interaction points. This is the collection of the owned *Ports*.
- *activityEdges [0..*]:* A collection of *ActivityEdges* that describe the dynamic behaviour of the *Component*.
- *activityNodes [0..*]:* A collection of *ActivityNodes* that describe the dynamic behaviour of the *Component*.

Constraints

1. A *Component* has to contain an initial node.
   self.activityNodes-> forall(oclIsKindOf(InitialNode))->size() = 1

Port

Ports are distinguished interaction points between *Components* and their environment. They isolate a *Component* from its environment by forming a point for conducting interactions between the internals of the *Component* and its environment. Hence, the *Component* can be defined independently of its environment making it more versatile. The interfaces associated with a *Port* specify the nature of the interactions that may occur over that *Port*. The required interfaces of a *Port* characterise the requests that may be made from the *Component* to its environment through that*Port*. The provided interfaces of a *Port* characterise requests to the *Component* that its environment may make through that *Port* [OMG06b].

In the component diagram, the squares on the *Components* represent *Ports*.

Generalisations

- ConnectableElement

Attributes No additional attributes.

Associations

- *engages [0..*]*: The collection of *CommunicationNodes* in the component behaviour the *Port* may engage in. These nodes define the requests that may be made from the *Component* to its environment or vice versa.

Constraints

1. A *Port* must be contained by one and only one *Component*.
   self.Component->size = 1

2. A *Port* must be connected to at least one *CommunicationNode* and an *OwnedInterface*.
   context Port:
   engages->size > 0 **and** (provided->size() > 0 **or** required->size() > 0)

OwnedInterface

An interface represented by the *OwnedInterface* class declares a set of features and obligations that a *Component* (or *Port*) offers. Since interfaces are declarations, they are not instantiable; that is, there are no run-time instances of interfaces. Interfaces are implemented by *ConnectableElements*. Note that a given *ConnectableElement* may implement more than one interfaces and that an interface may be implemented by a number of different *ConnectableElements*. A *ConnectableElement* that implements an interface specifies instances that are conforming to the interface and to any of its ancestors.

An interface realisation means that the *Component* supports the set of features owned by the interface, and any of its parent interfaces. The set of interfaces realised by a *Component* is its set of provided interfaces. They describe the services that the instances of that *Component* offer to their clients. Interfaces may also be used to specify required interfaces. Required interfaces specify services that a *Component* needs in order to perform its function and fulfill its own obligations to its clients [OMG06b].

In the component diagram, circles represent *provided interfaces*; the socket shaped elements represent *required interfaces*.

Attributes

- *name: String [1]:* The name of the *OwnedInterface*.

Associations

- *methods [0..*]:* References all the operations owned by the *OwnedInterface*.

Constraints

1. An interface must contain at least one method.
   self.methods->size() >= 1

2. *OwnedInterfaces* have unique names.
   context OwnedInterface **inv**:
   OwnedInterface.allInstances()->
   forall(p1,p2 | p1 <> p2 **implies** p1.name <> p2.name)

Method

A *Method* is a behavioural feature of a classifier that specifies the name, type, parameters, and constraints for invoking the associated behaviour [OMG06b]. *Methods* are described by their names. The *isReturning* value designates if the *Method* has a return value.

Attributes

- *name: String [1]:* The name of the *Method*.

- *isReturning: boolean [0..1]:* Specifies if the *Method* has a return value or not. If *true*, the *Method* returns a value to the caller. The actual value is not part of the model.

Associations No additional associations

Constraints

1. *Methods* have unique names.
   context Method **inv**:
   Method.allInstances()->
   forall(p1,p2 | p1 <> **implies** p1.name <> p2.name)

2. *AcceptEventActions* represent the reception of method calls in case of provided interfaces. Consequently, if the interface containing the *Method* is provided by a *Port*, the *Port* must engage in an *AcceptEventAction* of the same name (as the *Method*).
   context Method:
   let receiveEvents =
   OwnedInterface.provided-> forall(oclIsKindOf(Port)).engages->
   forall(oclIsKindOf(AcceptEventAction)) **in**
   if OwnedInterface.provided->size() >= 1 **then**
   receiveEvents->size() >= 1 **and** *receiveEvents*.name->intersection(name) = 1

3. *SendSignalActions* represent the method calls through required interfaces. Hence, if the interface containing the *Method* is required by a *Port*, the *Port* must engage in a *SendSignalAction* of the same name (as the *Method*).
   context Method:
   let sendEvents =
   OwnedInterface.required-> forall(oclIsKindOf(Port)).engages->
   forall(oclIsKindOf(SendSignalAction)) **in**
   if OwnedInterface.required->size() >= 1 **then**
   sendEvents->size() >= 1 **and**
   sendEvents.name->intersection(name) = 1

4. If the method is *returning* and the interface containing the *Method* is required by a *Port*, then the *Port* must engage in an *AcceptEventAction* of the same name as well. This represents that the *Component* responsible for the method calling receives the return value.
   context Method:
   let returnEvents =
   OwnedInterface.required-> forall(oclIsKindOf(Port)).engages->
   forall(oclIsKindOf(AcceptEventAction)) **in**
   if isReturning **then** OwnedInterface.required->size() >= 1 **and**
   returnEvents->size() >= 1 **and**
   returnEvents.name->intersection(name) = 1

5. If the method is *returning* and the interface containing the method is provided by a *Port*, then the *Port* must engage in a *SendSignalAction* of the same name as well. This represents that the *Component* providing the functionality behind the method sends the return value.

context Method:
let replyEvents =
OwnedInterface.provided-> forall(oclIsKindOf(Port)).engages->
forall(oclIsKindOf(SendSignalAction)) **in**
if isReturning **then** OwnedInterface.provided->size() >= 1 **and**
replyEvents->size() >= 1 **and** *replyEvents*.name->intersection(name) = 1

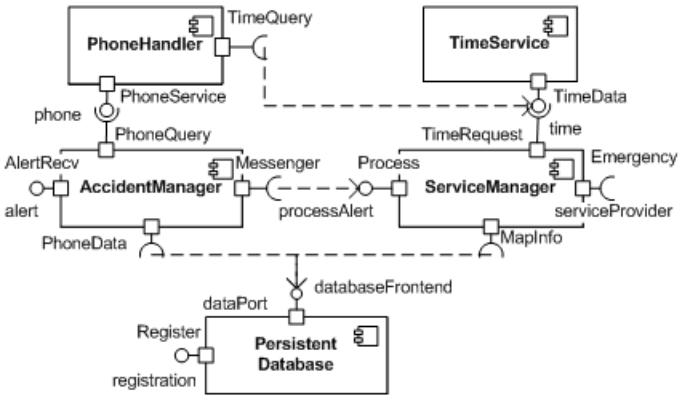


Figure 3.3: Component diagram of the accident server

3.3 Activities

The *Activities* package defines the workflow modelling language of UML. Although the *Activities* package of UML2 is not a simple control-flow language (it also contains object flow, signalling and even exception handling), CSML uses only the basic control-flow elements. The activities part of the metamodel is demonstrated in Figure 3.4. The diagram associated to the *Activities* package is the *activity diagram*. The owned behaviour of the *ServiceManager* component expressed as an activity diagram is shown in Figure 3.5.

The semantics of activities are based on *token flows*. A token traverses an edge: it may be taken from the source node and moved to the target node. A node may initiate the execution when specified conditions on its input tokens are satisfied. The classes we use are the following, based on [OMG06b].

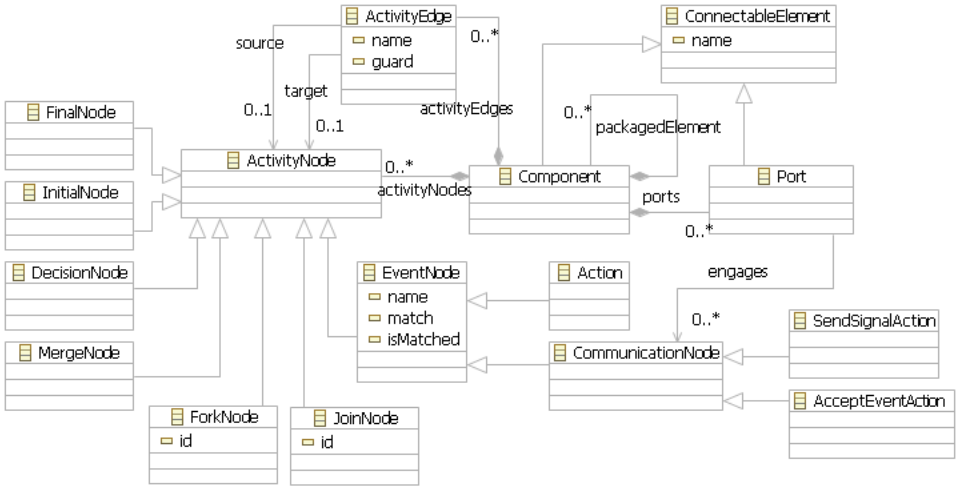


Figure 3.4: Activities metamodel

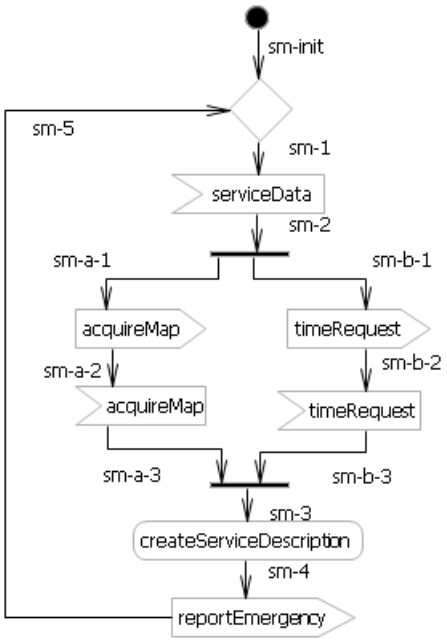


Figure 3.5: Owned behaviour of *ServiceManager* as an activity diagram

ActivityNode

An *ActivityNode* is an abstract class for points in the flow of an activity connected by edges. It covers executable nodes and control nodes [OMG06b].

Attributes No additional attributes.

Associations No additional associations.

Constraints

1. Every *ActivityNode* must be contained within one and only one *Component*.
   self.Component->size() = 1

ActivityEdge

An *ActivityEdge* is a directed connection that has a source *ActivityNode* and a target *ActivityNode*, along which tokens may flow. In CSML *ActivityEdges* represent control flow that only passes control tokens. An *ActivityEdge* is notated by an open arrowhead line connecting two *ActivityNodes*. If the edge has a name, it is notated near the arrow.

Attributes

- *name: String [1]:* The name of the *ActivityEdge*.
- *guard: String [0..1]:* Specification evaluated at runtime to determine if the edge can be traversed.

Associations

- *source [0..1]:* Node from which tokens are taken when they traverse the edge.
- *target [0..1]:* Node to which tokens are put when they traverse the edge.

Constraints

1. *ActivityEdges* have unique names.
   context ActivityEdge **inv**:
   ActivityEdge.allInstances()->
   forall(p1,p2 | p1 <> p2 **implies** p1.name <> p2.name)
2. The source and the target of an edge must be in the same *Component* as the edge.
   self.source.Component = self.Component
   and self.target.Component = self.Component

EventNode

EventNode is an abstract class for all named *ActivityNodes* that represent an event in the system. *EventNode* encompasses the simple *Action* and the communication nodes as well.

Generalisations

- ActivityNode

Attributes

- *name: String [1]:* The name of the *EventNode*.
- *match: integer [0..1]:* The *match* value is used for expressing object connections in graph production rules. It is described in detail in Section 5.5.
- *isMatched: boolean [1]:* The *isMatched* attribute expresses if the point is a gluing point in a graph production rule. It is described in detail in Section 5.5.

Attributes No additional associations.

Constraints

1. *EventNodes* have unique names.
   context EventNode **inv**:
   EventNode.allInstances()->
   forall(p1,p2 | p1 <> p2 **implies** p1.name <> p2.name)
2. *EventNodes* must have one and only one incoming edge.
   self.target->size() = 1
3. *EventNodes* must have one and only one outgoing edge.
   self.source->size() = 1

Action

An *Action* represents a single step within an activity. It begins execution by taking tokens from its incoming control edges. When the execution of an *Action* is complete, it offers tokens on its outgoing *ActivityEdges*. *Actions* are notated as round-cornered rectangles with their names in their centre.

In official UML2, every action is allowed to have multiple incoming and outgoing edges. We restrict them to have one incoming and one outgoing edge. The semantic expressiveness is not changed: a substitute merge and decision node can be included before and after the action.

Generalisations

- EventNode, ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints No additional constraints.

InitialNode

An *InitialNode* is a control node that starts the flow. A token is placed on the *InitialNode* when the activity starts. *InitialNodes* are notated as solid circles.

Generalisations

- ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints

1. There can be only one *InitialNode* in a *Component*.
   context Component:
   activityNodes->forall(oclIsKindOf(InitialNode))->size() = 1

FinalNode

An *activity final node* stops all flows in the activity. More precisely, it stops all executing actions in the activity, and destroys all tokens on object nodes. As we use only one type of final node, we use the *FinalNode* name. A *FinalNode* is notated as a solid circle with a hollow circle inside.

Generalisations

- ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints

1. A *FinalNode* must have one and only one incoming edge.
   self.target->size() = 1

CommunicationNode

CommunicationNode is an abstract class for those special nodes that engage in communication through a *Port*: *AcceptEventAction* receives and *SendSignalAction* sends messages through a *Port*.

Generalisations

- EventNode, ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints

1. A *CommunicationNode* must be connected to precisely one *Port* that provides or requires an interface with a *Method* of the same name as the *CommunicationNode*.
   context CommunicationNode: Port->size() = 1 **and**
   (Port.provided.methods.name->intersection(name)->size() = 1 **or**
   Port.required.methods.name->intersection(name)->size() = 1)

AcceptEventAction

An *AcceptEventAction* is an action that waits for the occurrence of an event meeting specified conditions. An *AcceptEventAction* may represent two types of communication events, depending on the port it engages in. It either means the reception of function calls through provided interfaces or the reception of function return values through required interfaces. An *AcceptEventAction* is notated with a concave pentagon.

Generalisations

- CommunicationNode, EventNode, ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints No additional constraints.

SendSignalAction

A *SendSignalAction* is an action that creates a signal instance from its inputs, and transmits it to the target object, where it causes the execution of an activity. A *SendSignalAction* may represent two types of communication events, depending on the *Port* it engages in. It either means a function call sent through a required interface or the sending of a return value through a provided interface. A *SendSignalAction* is notated with a convex pentagon.

Generalisations

- CommunicationNode, EventNode, ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints No additional constraints.

DecisionNode

A *DecisionNode* is a control node that chooses between outgoing flows. Each token arriving at a decision node can traverse only one outgoing edge. Tokens are not duplicated. Each token offered by the incoming edge is offered to the outgoing edges. Then, guards of the outgoing edges are evaluated to determine which edge should be traversed. The notation of a *DecisionNode* is a diamond-shaped symbol.

Generalisations

- ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints

1. A *DecisionNode* has exactly one incoming edge.
   self.target->size() = 1
2. A *DecisionNode* has at least two outgoing edges.
   self.source->size() >= 2
3. Exactly one of the outgoing edges must have the *else* guard condition.
   self.target->forall(guard = 'else')->size() = 1

MergeNode

A *MergeNode* is a control node that brings together multiple alternate flows. All tokens offered on incoming edges are offered to the outgoing edge. There is no synchronisation of flows or joining of tokens. The notation for a *MergeNode* is a diamond-shaped symbol.

Generalisations

- ActivityNode

Attributes No additional attributes.

Associations No additional associations.

Constraints

1. A *MergeNode* has exactly one outgoing edge.
   self.source->size() = 1
2. A *MergeNode* has at least one incoming edge.
   self.target->size() >= 1

ForkNode

A *ForkNode* is a control node that splits a flow into multiple concurrent flows. Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token. The notation for a *ForkNode* is a thick line segment.

A *ForkNode* has an *id* element that represents a pairing between a *ForkNode* and a *JoinNode*. In our system, we deal with well-structured activity diagrams. Allowing generic branching as in Figure 3.6(b) imposes theoretical problems in the mapping to CSP as discussed in Section 5.3.5.

Generalisations

- ActivityNode

Attributes

- *id: integer [1]:* The element that represents pairing between a *ForkNode* and a *JoinNode*.

Associations No additional associations.

Constraints

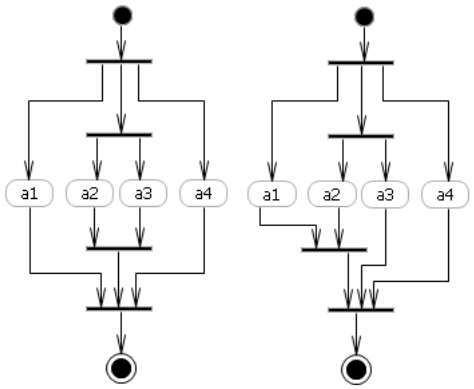
1. A *ForkNode* has exactly one incoming edge.
   self.target->size() = 1
2. A *ForkNode* has at least two outgoing edges.
   self.source->size() >= 2

JoinNode

A *JoinNode* is a control node that synchronises multiple flows. If there are tokens offered on all incoming edges, then a token is offered on the outgoing edge. The notation for a *JoinNode* is a thick line segment.

Generalisations

- ActivityNode



(a) Well-Formed (b) Not Well-Formed

Figure 3.6: Parallelism in activity diagrams

Attributes

- *id: integer [1]:* The element that represents pairing between a *ForkNode* and a *JoinNode*.

Associations No additional associations.

Constraints

1. A *JoinNode* has exactly one outgoing edge.
   self.source->size() = 1

2. A *JoinNode* has at least one incoming edge.
   self.target->size() >= 1

3. The *id* of the *JoinNode* must be defined.
   self.id >= 0

4. There must be always a paired *ForkNode* in owner component, that has the same unique *id*.
   context Component:
   activityNode->forall(oclIsKindOf(JoinNode)).id->
   intersection(activityNode->forall(oclIsKindOf(ForkNode)).id)->size() = 1

Restricting possible configurations of the *ForkNode* and *JoinNode* changes semantic expressiveness compared to the official UML2 definition in [OMG06b]. Only

activity diagrams that are well-formed with respect to parallelism are allowed. Figure 3.6(b) depicts a reasonably complicated not well-formed case: there is no obvious correspondence between the *ForkNodes* and *JoinNodes*. Figure 3.6(a) shows a well-formed activity diagram: every *ForkNode* has a corresponding *JoinNode*. We assume that every spanned outgoing process will be synchronised at one corresponding *JoinNode* (if present).

3.4 Composite Structures

Composite structure diagrams represent the system as a composition of interconnected run-time component instances collaborating over communications links [Amb04]. While the *Components* and *Activities* packages had corresponding diagrams, the *Composite Structures* incorporates several packages: these are the *InternalStructures, Ports, Collaborations, StructuredClasses* and *Actions*. The composite structures part of the metamodel is illustrated in Figure 3.7. A possible configuration of the accident server is shown in Figure 3.8. The classes we use are the following, based on [OMG06b].

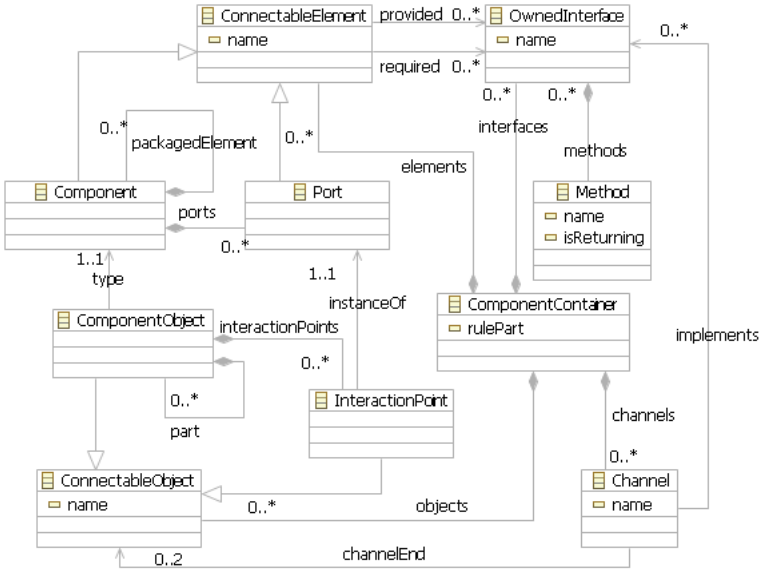


Figure 3.7: Composite structures metamodel

ConnectableObject

Used according to the *Composite* pattern [BMR+96], the *ConnectableObject* is a generic parent for the instance-level objects. The instance level objects are connected to the type level elements with instantiation connections.

Attributes

- *name: String [1]*: The name of the *ConnectableObject*.

Associations No additional associations.

Constraints

1. *ConnectableObjects* have unique names.
   context ConnectableObject **inv**:
   ConnectableObject.allInstances()->
   forall(p1,p2 | p1 <> p2 **implies** p1.name <> p2.name)

ComponentObject

A *ComponentObject* is an instance of its typing *Component*. Object instantiation is denoted by the *type* relation. *ComponentObjects* are graphically represented as boxes with the usual *instance : type* title pattern.

Generalisations

- ConnectableObject

Attributes No further attributes.

Associations

- *part [0..*]:* As *Components* may contain other *Components*, *ComponentObjects* may contain other *ComponentObjects* as well. This is the collection of the owned *ComponentObjects*.
- *type [1]:* The typing *Component* of the represented instance.
- *interactionPoints [0..*]:* The collection of the owned *InteractionPoints*.

Constraints No additional constraints.

InteractionPoint

An *InteractionPoint* is a model element that represents an instance of a *Port* in the system. Port instantiation is denoted by the *instanceOf* relation. The graphical notation is a square on the owner *ComponentObject*.

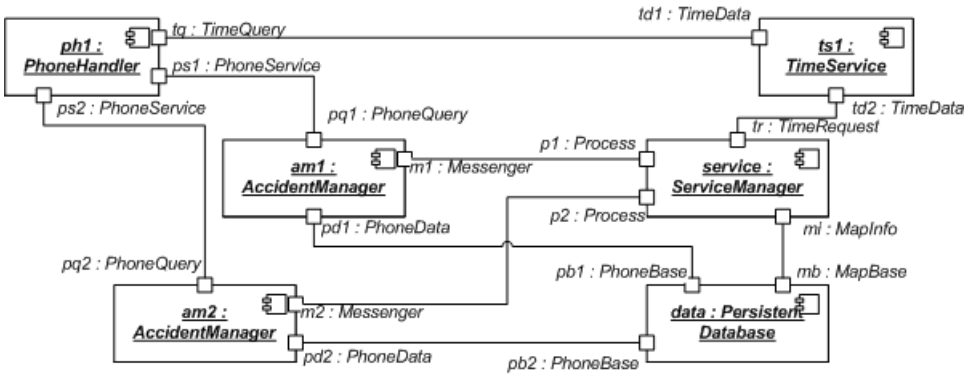


Figure 3.8: Composite structure diagram of the accident server

Generalisations

- ConnectableObject

Attributes No additional attributes.

Associations

- *type [1]:* The typing *Port* of the represented instance.

Constraints

1. An *InteractionPoint* must be contained by one and only one *ComponentObject*.
   self.ComponentObject->size = 1

Channel

A *Channel* specifies a link that enables communication between two component instances [OMG06b]. The possible communication events through a *Channel* are determined by the type of the *InteractionPoints* on the endpoints of the channels. *InteractionPoints*, being instances of *Ports* and thus typed by interfaces, define the possible actions that can happen through the relevant *Channel*. *Channels* are not instances of *OwnedInterfaces*; they are rather realisations of the communication behaviour the interfaces declare, expressed through the *implements* relation.

Attributes

- *name: String [1]*: The name of the *Channel*.

Associations

- *channelEnd [0..2]:* A *Channel* has two channel ends, each representing the participation of instances of the *Ports* typing the *ConnectableObjects*.
- *implements [0..*]:* The *OwnedInterface* of the represented realisation.

Constraints

1. *Channels* have unique names.
   context Channel **inv**:
   Channel.allInstances()->
   forall(p1,p2 | p1 <> p2 **implies** p1.name <> p2.name)
2. Both ends of a *channelEnd* must point to a *ConnectableObject*.
   self.channelEnd->size() = 2
3. The two ends of the *channelEnd* must point to different *ConnectableObjects*.
   context Channel **inv**:
   channelEnd->size() = 2 **implies** channelEnd->forall(p1,p2 | p1 <> p2)

3.5 Diagrammatic Representation

As introduced in the previous sections, CSML encompasses the *component*, *composite structure* and *activity* views of the system. The diagrammatic representation we are about the present merges all these views and visualises them in one diagram called the *combined structure diagram*. The complete combined structure diagram of the accident server is shown in Figure 3.9. The diagram consists of two parts separated by a dashed line: the upper part is the type-level, and the lower part is the instance-level.

- The type-level part of the combined structure diagram is based on the component diagram. The activity diagrams describing component behaviour are embedded inside the components. The communication events fit into the communication framework by representing function calls from the corresponding *OwnedInterfaces* through the relevant *Ports*. For instance, the *phoneData SendSignalAction* in *AccidentManager* represents the function call from the *databaseFrontEnd* interface through the *PhoneData* port. The corresponding *AcceptEventAction* in *PersistentDatabase* receives the function call.

- The instance-level part of the combined structure diagram adopts the notation of the composite structure diagram. The configuration of the accident server shown in Figure 3.9 is similar to the one depicted in Figure 3.8.

In the following we present the behaviour of components in the accident server that will be further used for describing refactorings.

AccidentManager

The *AccidentManager* is the core component, responsible for receiving incoming alerts through the *AlertRecv* port. In order to initiate a phone call it acquires the phone number of the driver from the *PersistentDatabase* and passes it to the *PhoneService*, which calls the driver. The alert is cancelled in case the driver denies the need for assistance. Otherwise, the call is returned to the *AccidentManager* to assess the available data (including sensorial and positioning data from the car) and decide if the situation is a real emergency. On an ascertained emergency, the necessary data is passed to the *ServiceManager*. The *ServiceManager* matches the GPS location of the car with the *MapHandler* and creates a service description. Then, the local emergency services are contacted through the *serviceConnector* interface .

PhoneHandler

The *PhoneHandler* is a dedicated interface to the cellular network. A *callNumber* function call is received with a phone number. The call process is started by initiating the connection. On a successful call, the car driver is questioned on his situation. On a failed call, the recent timestamp is acquired and the various data concerning the call attempt are collected. When receiving the *callInfo* function call, the call data is transferred to the *AccidentManager*.

PersistentDatabase

The *PersistentDatabase* is the simplified database component of the accident server. It contains the phone numbers of the registered car drivers as well as the map of the area within jurisdiction. Components can query the phone numbers of users based on their identification or ask for a route plan from the nearest emergency service station to the location of the crashed car.

ServiceManager

The *ServiceManager* provides the connection to the emergency services. An ascertained alert is passed through the *serviceData* function call. Based on the detailed information, the precise location and timestamp is obtained. With all necessary details at hand, a service request is compiled and dispatched to the local emergency services through the *reportEmergency* call.

TimeService

The *TimeService* is a simple time server. It provides a precise timestamp to all requests.

3.6 Differences to UML2

This section elaborately details the differences between the UML 2.1.1 [OMG06b] and the *combined structure modelling language* (CSML) metamodel (Fig. 3.1).

The CSML metamodel is close to a subset of the UML metamodel with several semantic variation points. UML is a general purpose modelling language with a vast amount of classes and properties. As CSML is specific to the architecture domain, it is radically simplified compared to UML. Most of the UML classes that were introduced for flexibility and universality pose an unnecessary complication for CSML. Excessive amount of unused classes are a problem for two main reasons:

- The performance of the transformation decreases,
- The graphical representation of a production rule becomes unintelligible.

We refrain from listing all the classes and features that are not present in CSML. Generally, the semantical and important structural differences between the two languages are introduced.

3.6.1 Component and Behaviour

A component is a self contained unit encapsulating the state and behaviour of a number of classifiers. As shown in Figure 3.10, these classifiers realise (or implement) the behaviour of a component in UML and thus are connected to the *ComponentRealization* class. Also, UML allows multiple different sets of realisations for a single specification. Thus, a component can contain multiple *ComponentRealization*

classes. A classifier incorporates behaviour specifications in its own namespace, the one connected through the *classifierBehaviour* association specifies the behaviour of the classifier itself. This behaviour specification can be an activity diagram.

In CSML, the notion of component realisation is not concerned. Also, the behaviour specification is not flexible: the *ActivityEdge* and *ActivityNode* classes are directly contained by the *Components*. The component behaviour is not assumed to be specified in different ways or by multiple activity diagrams.

3.6.2 Semantics of *SendSignalAction* and *AcceptEventAction*

The semantics of *SendSignalAction* and *AcceptEventAction* is generally similar in UML and CSML: *SendSignalAction* creates a signal instance and transmits it to the target object; *AcceptEventAction* waits for the occurrence of an event meeting its conditions. Their coupling is different however.

The classes involved in UML signalling are shown in Figure 3.11. *SendSignalAction* is a child of *InvocationAction* that allows signals to be sent through ports. When sending a signal, a *Signal* class is instantiated. On the receiving side, the *Trigger* object of the *AcceptEventAction* specifies the type of events accepted by the action. Any *Event*, including *SignalEvents* can be accepted as well. A *SignalEvent* occurs when a signal message, originally caused by a *SendSignalAction*, is received by another object. When the trigger is activated, *SignalEvent* results in the execution of the *AcceptEventAction*.

In CSML the inheritance structure and mechanics is different: *AcceptEventAction* and *SendSignalAction* are children of *CommunicationNode*. As shown in Figure 3.4, *CommunicationNode* is directly connected to *Port*. The main difference is the absence of explicit signal classes: the event is implicitly determined by the interface typing the port and the name of the relevant communication node (Sec. 3.3).

3.6.3 Composite Structures

As mentioned in Section 3.6.1, there are no explicit *Port* or *Component* instances in UML. Components are realised through classifiers connected to the corresponding *ComponentRealization* class. When an instance of a classifier is created, instances corresponding to each of its ports are created as well and held in the slots specified by the ports. These instances are referred to as *interaction points*. The *interaction point object* must be an instance of a classifier that realises the provided interfaces of

the port.

CSML takes a direct view on the case: both *Components* and *Ports* have explicit instances. The instance of a *Component* is a *ComponentObject* and the instance of a *Port* is an *InteractionPoint*.

Although the idea of the composite structure view is the same in UML and CSML, the contents is substantially different because of the above differences. Since the instantiation relation of components and ports in UML is not direct, the composite structures view contains the instances of the realising classifiers. In CSML, the object and port instances are concerned.

3.6.4 Inheritance Structure of Activities

After presenting the major semantic differences, we elaborate on the metamodel of the activities view. The coupling of a component and its activity diagram is discussed in Section 3.6.1. This section details the inheritance structure. The UML metamodel of the activities view is shown in Figure 3.12.

The edges that represent the control flow between the activities are represented by the *ActivityEdge* class in CSML. In UML however, not only control flow, but object flow edges are allowed. Hence, the *ActivityEdge* is an abstract class with *ObjectFlow* and *ControlFlow* children.

The activities are represented by the *ActivityNode* abstract class in both languages. In UML, there is a *ControlNode* intermediate class, that serves as a parent for *InitialNode*, *FinalNode*, *ForkNode*, *JoinNode*, *DecisionNode* and *MergeNode*. In the CSML, these classes are direct children of *ActivityNode*. In UML, there are two different final nodes: *ActivityFinalNode* shuts down the execution of the activity diagram while *FlowFinalNode* only stops that particular flow, where the token was consumed. CSML uses only the latter, and thus, instead of an inheritance hierarchy, it has only one *FinalNode* class.

The *Action* is a direct child of *ActivityNode* in UML, but in CSML there is an *EventNode* that generalises *Action* and *CommunicationNode*. The reasons for the existence of *CommunicationNode* is detailed in Section 3.6.2.

3.6.5 OwnedInterface and Methods

The connections between the *Component*, *Interface* and *Operation* classes are similar in both models. They are named differently however: *Interface* is called *OwnedInterface* and *Operation* is called *Method* in CSML. *Interface* was renamed because of

implementation reasons; Tiger EMF Transformer executes generated Java code, and *interface* is a reserved word in Java.

However, when Figure 3.13 is compared to the CSML metamodel (Fig. 3.2), the difference between the contained attributes is apparent. While there is a vast amount of attributes in UML, CSML is intentionally simplified: a method has only a name, and a boolean variable *isReturning*. As introduced in Section 3.2, *isReturning* denotes if the method has a return value, but the value itself is not considered.

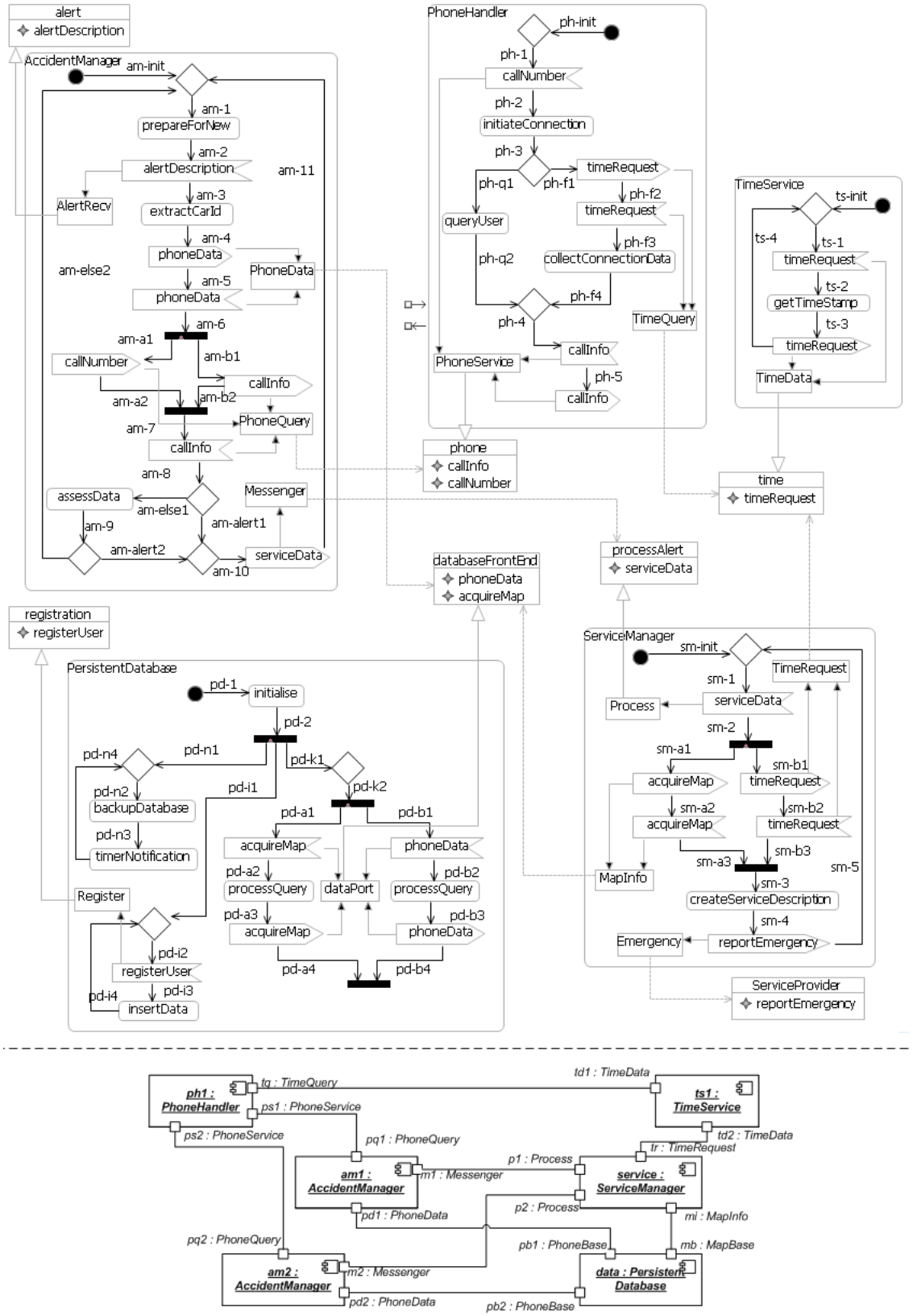


Figure 3.9: Combined structure diagram of the accident server

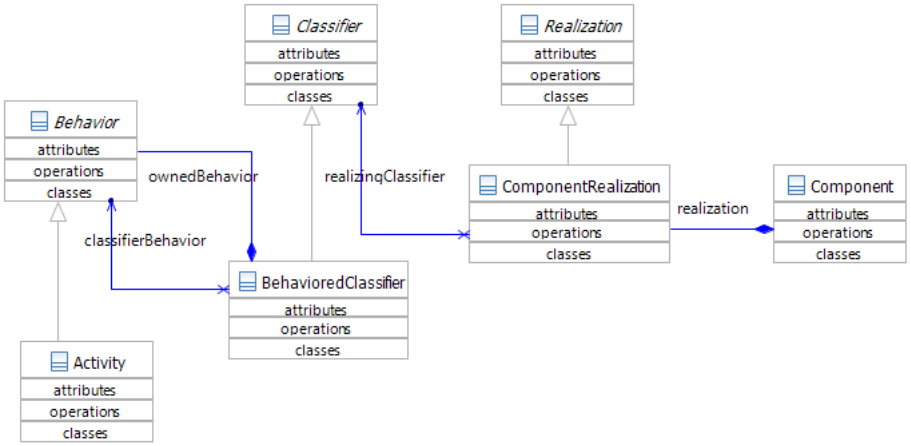


Figure 3.10: Connecting the behaviour to the Component

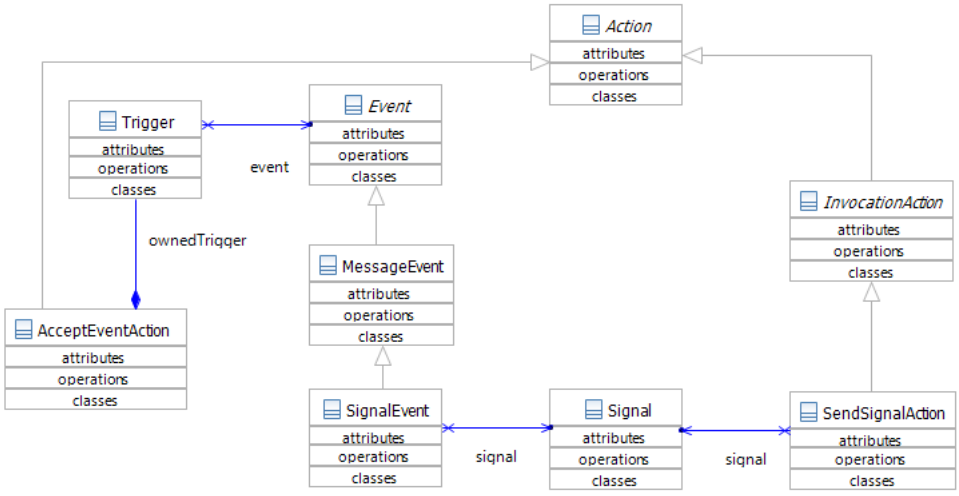


Figure 3.11: Connecting *SendSignalAction* and *AcceptEventAction*

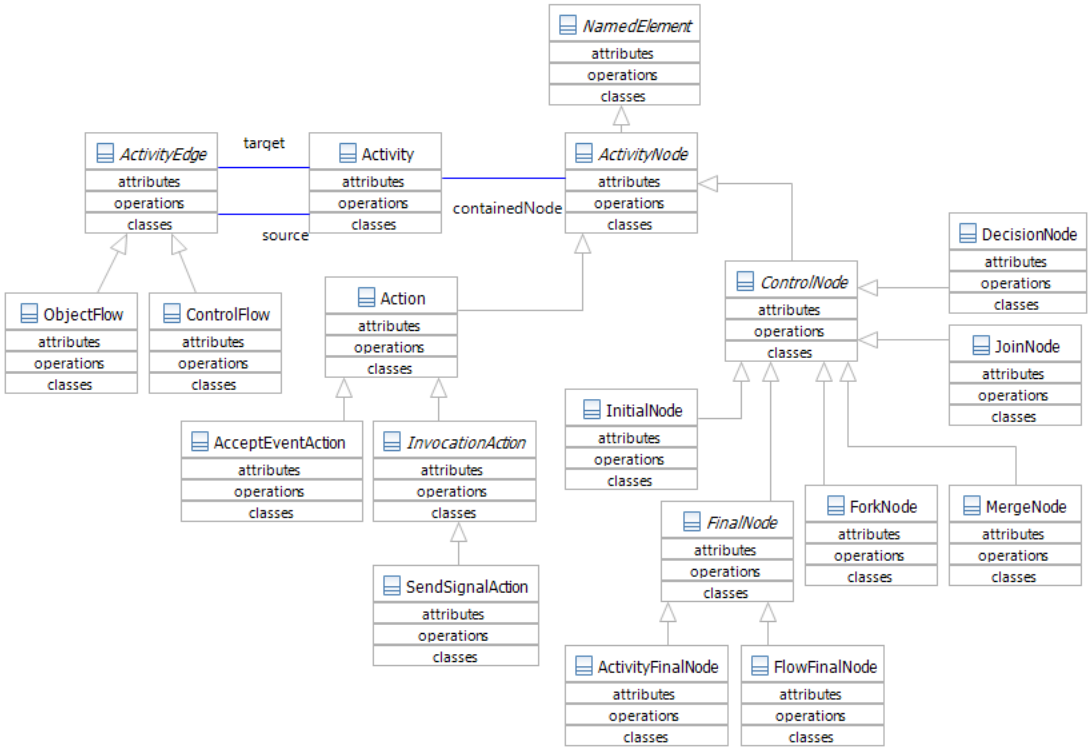


Figure 3.12: Metamodel of Activities

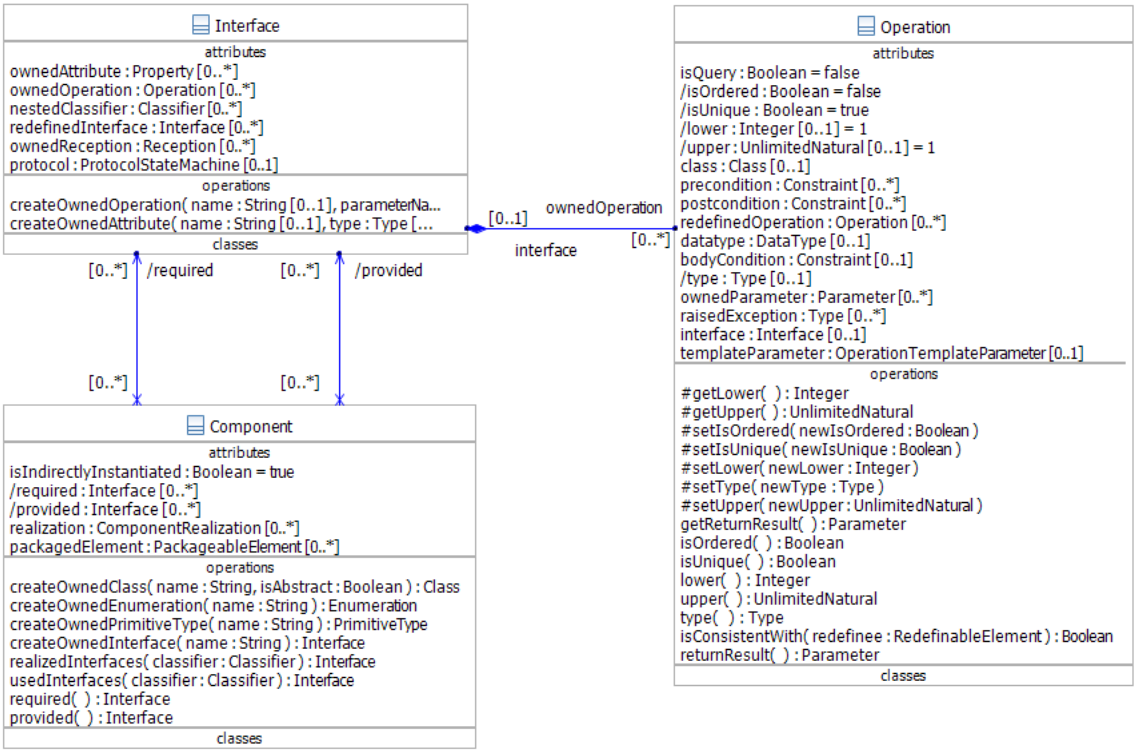


Figure 3.13: Connecting Component, Interface and Operation

Chapter 4

Semantic Domain

As an example semantic domain for the theoretical contributions, Communicating Sequential Processes (CSP)[Hoa85] were chosen. CSP is a process algebra providing formal means of modelling concurrent systems by describing the interactions, communications, and synchronisation between a collection of independent agents [Hen88].

CSP as a semantic domain was chosen with implementation in mind. We use the FDR2 [FSEL05] refinement checker for establishing properties of models expressed in CSP. There are differences between the 'official' CSP and the dialect interpreted by FRD2. We adopted a CSP dialect based on FDR2; its syntax is presented in Section 4.1.

The semantic mapping that maps the semantic domain to a model is described by graph transformations. Thus, CSP need to be represented as instances of a typed graph. This graph based representation and the correspondence between the graph instances and the term-based form is introduced in Section 4.2.

The various notions of semantics in CSP that can be used for verification are elaborated in Section 4.3.

4.1 Syntax

FDR2 stores the systems expressed in CSP in files. An FDR2 compliant CSP file consists of three major parts: channel definitions, system specifications, and system equations. The definition of the syntax of a CSP file is presented using EBNF [Int96].

```
fdr2 file =
   [channel definitions, new line],
   [system specification, new line],
```

```
   [system equation, new line];

channel definitions = set of events;
system specification = process assignment*;
system equation = process assignment*;
```

The *channel definitions* form a collective alphabet of the described systems: it lists all possible events. A *system specification* is the actual set of CSP expressions that defines the behaviour of a system. The *system equation* is the root process of a system. As a file may contain multiple systems, the difference between the *system specifications* and *system equations* is only semantic.

The basic elements of CSP are processes. A *process* is the behaviour pattern of an agent with an alphabet of a limited set of events. Processes are defined using recursive process equations with guarded expressions. For clarity, we use the terminology shown in the expression below.

$$\overbrace{\underbrace{P =}_{declaration}\ \underbrace{(a \rightarrow Q)||(b \rightarrow R)}_{definition}}^{assignment}$$

A CSP expression that defines the behaviour of a process is called process *assignment*. The *declaration* specifies the name of the process, while the *definition* is a *process expression* that describes the behaviour assigned to that process. The syntax of the process assignments is the following.

```
process assignment = process identifier, '=', process expression;
process identifier = process name;
process expression = process name
 | event, "->", process expression
 | process expression [] process expression
 | process expression, "[|", [set of events], "|]", process expression
 | process expression, "|||", process expression
 | process expression \ set of events
 | process expression, ";", process expression
 | renaming
 | "SKIP";
set of events = "{|", event name*, "|}";
```

The interpretation of these process expressions is as follows. The prefix operator `a -> P` performs event `a` and then behaves like `P`. The process `P [] Q` represents

an external choice between processes `P` and `Q`. In this case the choice is observable and controllable by the environment. The hiding operator in case of `P \ a` means a process that behaves like `P` except that all occurrences of event `a` are hidden. The split operator `P ; Q` means that upon the successful termination of `P` it behaves like `Q`. The process `SKIP` represents successful termination.

FDR2 treats the process alphabets and the parallel composition operator in a significantly different way than the official CSP. According to [Hoa85], every process has its own, intrinsic alphabet αP, and a simple parallel composition operator $(P \parallel Q)$ is used. Hence, the synchronised events are not defined explicitly, they are the intersection of the respective alphabets, i.e. $\alpha P \cap \alpha Q$. In FDR2 the processes lack the intrinsic alphabet definition. As mentioned, the *channel definition* contains the list of all possible events, but they are not explicitly bound to a particular process. Consequently, a parameterised concurrency operator `P [| X |] Q` is used where `X` is the set of synchronised events. The participating processes are engaged in a lock-step synchronisation, the events outside the set `X` are interleaved. An interleaving process `P ||| Q` is truly concurrent: there is no synchronisation at all.

In our application it is important to define a group of processes with similar behaviour. To this end, we use renaming. When renaming, we label each process by a different name, also each event of the labelled process is labelled by that name. A labelled event is a pair `l_x` where `l` is a label, and `x` is the symbol for the event. A process `P` labelled by `l` is also denoted by `l_P`. It engages in the event `l_x` whenever `P` would have engaged in `x`. In the 'official" CSP, a renaming function is used to define `l_P`. The function is `f_l(x) = l_x` for all $x \in \alpha P$ and the definition of the renaming is `l_P = f_l(P)` [Hoa85]. In FDR2 however, renaming bears a different notation; assuming the renaming $r(event_1) = renamed_1$, $r(event_2) = renamed_2$ and $r(P) = R$ in CSP, the syntax of the respective FDR2 representation is shown below.

```
renaming = process name, "[[", event renaming, "]]";
event renaming = [event name, "<-", event name]*;

process name = [side, '_'], [label, '_'], name, ['_', subscript];
event name = [label, '_'], name, ['_', subscript];

side = terminal string;
label = terminal string;
name = terminal string;
subscript = terminal string;
```

4.2 CSP Metamodel

The metamodel TG_{CSP} of the graph-based representation of CSP is illustrated in Figure 4.1. It is the abstract syntax tree of its EBNF form. The mapping between the graph-based representation and the term-based representation is formalised with the mapping $sem_{g2t} : \mathbf{Graph_{TG_{CSP}}} \rightarrow CSP$. In this section we introduce both the graph-based representation and the mapping sem_{g2t}. The classes and corresponding terms of the metamodel are elaborated in the following. It is important to note that there are no additional constraints defined on the various classes. The only constraint is based on the containment: a contained element must have precisely one container. This also helps to avoid unwanted connections.

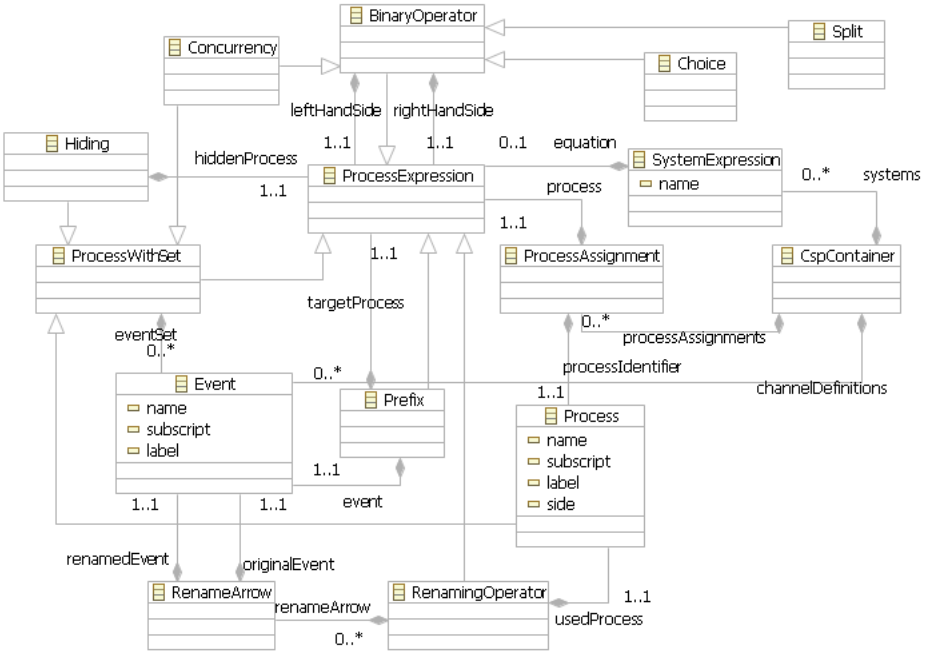


Figure 4.1: TG_{CSP}: the metamodel for CSP

CspContainer

The *CspContainer* is a similar element to the *ComponentContainer* in the architecture metamodel. As the implementation uses the EMF *ecore* [EMF07], the model is represented as an XML tree. The *CspContainer* is the root node that contains all elements.

Associations

- *systems [0..*]*: A collection of *system equations* owned by the *CspContainer*.
- *channelDefinitions [0..*]*: A collection of *events* that forms the channel definitions.
- *processAssignments [0..*]*: A collection of *ProcessAssignments* used to specify the behaviour of the contained systems.

ProcessAssignment

The *ProcessAssignment* assigns behaviour defined as a *ProcessExpression* to a process identifier. As shown in Figure 4.2, the *processIdentifier* edge contains the declared process while the *process* edge holds the definition of the declared process.

Associations

- *processIdentifier [1]:* the declaration of the process name.
- *process [1]*: the *ProcessExpression* that defines the behaviour assigned to the process.

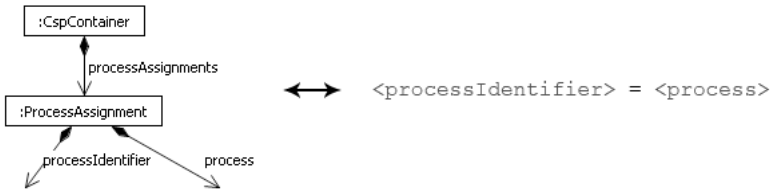


Figure 4.2: Graph to terms: *ProcessAssignment*

ProcessExpression

ProcessExpression is the abstract composite class of process expressions. As CSP revolves around processes and expressions that define them, nearly all classes are children of this class in the CSP metamodel.

Process

The instances of class *Process* represent processes, the basic building blocks of CSP. In Figure 4.3 a complete process assignment is illustrated: process P is defined to behave as process Q. The attributes *name*, *subscript* and *label* create together the

$label.name_{subscript}$ pattern. In FDR2 this pattern is flattened to `label_name_subscript`. As CSP is used for refinement checking, identical process names are not allowed.

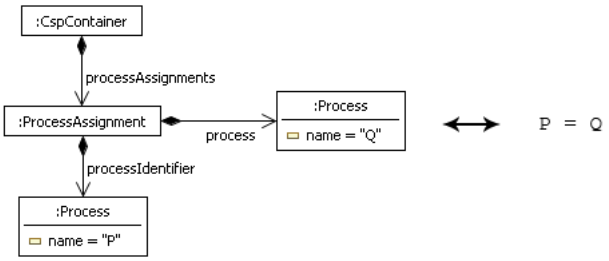


Figure 4.3: Graph to terms: *Process*

Attributes

- *name: String [1]:* The name of the *Process*.
- *subscript: String [0..1]:* The subscript of the process name.
- *label: String [0..1]:* The label of a renamed *Process*.
- *side: String [0..1]:* Indicates the particular side of the refactoring rule (LHS or RHS) where the process is.

Event

Event represents an event class in CSP with its instances representing occurrences of that event. The attributes *name*, *subscript* and *label* create together the $label.name_{subscript}$ pattern. In FDR2 this pattern is flattened to `label_name_subscript`.

Attributes

- *name: String [1]:* The name of the *Event*.
- *subscript: String [0..1]:* The subscript of the event name.
- *label: String [0..1]:* The label of a renamed *Event*.

Prefix

Prefix represents the prefix operator in CSP. An example is shown in Figure 4.4: process P is initially willing to communicate a and will wait indefinitely for this a to happen. After a it behaves like Q.

Associations

- *event [1]*: the *Event* that the process engages.
- *targetProcess [1]*: after communicating a, the system behaves like this *Process*.

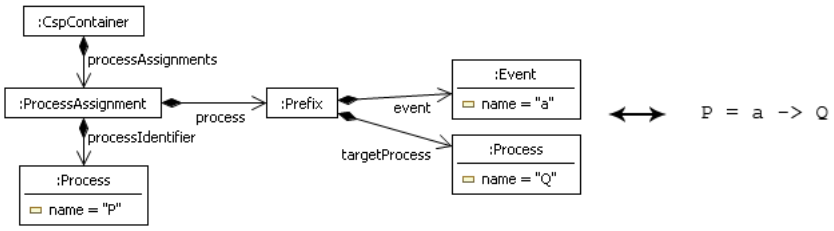


Figure 4.4: Graph to terms: *Prefix*

BinaryOperator

The *BinaryOperator* is an abstract class that represents all process operators taking two operands.

Generalisations: ProcessExpression

Associations

- *leftHandSide [1]*: the *ProcessExpression* on the left-hand side of the operator.
- *rightHandSide [1]*: the *ProcessExpression* on the right-hand side of the operator.

ProcessWithSet

Another abstract superclass is the *ProcessWithSet*. It generalises such process expressions that may have a set of events associated with them.

Generalisations: ProcessExpression

Associations

- *eventSet [0..*]*: the set of *events* associated with the process expression.

Concurrency

The *Concurrency* class represents the parallel composition of processes. Figure 4.5 shows an example configuration. Processes E and F are in parallel composition. The two processes engage in a lock-step synchronisation with events a and b. The *leftHandSide* and *rightHandSide* operands are inherited from the *BinaryOperator*, while the *eventSets* are from the *ProcessWithSet*.

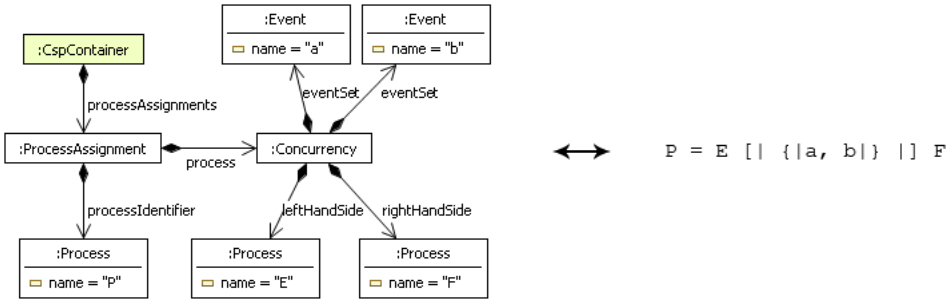


Figure 4.5: Graph to terms: *Concurrency*

Generalisations: ProcessExpression, BinaryOperator, ProcessWithSet

Choice

In our CSP dialect we use only external choice which is represented by the *Choice* class. In Figure 4.6 process P behaves either like E or like F.

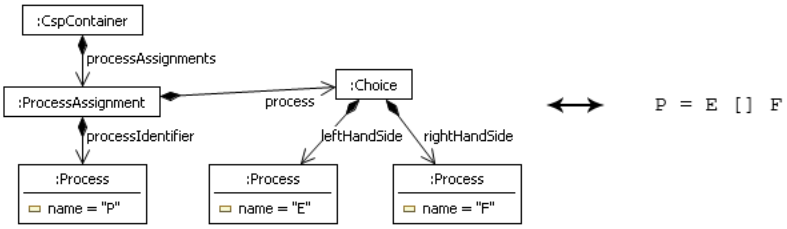


Figure 4.6: Graph to terms: *Choice*

Generalisations: ProcessExpression, BinaryOperator

Hiding

The class *Hiding* is the graph based representation of the hiding operator. Process P in Figure 4.7 is defined to behave as process Q, except that events a and b have been

internalised, i.e hidden from the outside environment.

Generalisations: ProcessExpression, ProcessWithSet

Associations

- *hiddenProcess [1]*: the process that has a set of events hidden from the environment.

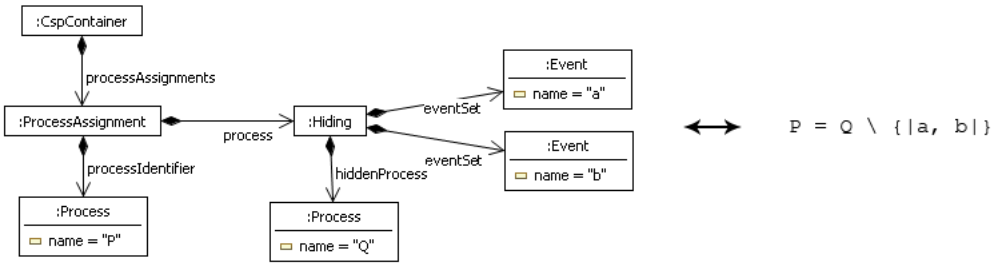


Figure 4.7: Graph to terms: *Hiding*

RenameArrow

When a process is renamed, it is necessary to transform all the events that it may engage in as well. *RenameArrow* defines the renaming of a single event explicitly. In Figure 4.8, the event class *commit* connected through *originalEvent* is renamed by the label *p1*. The renamed event used in the renamed process is *p1_commit*.

Associations

- *originalEvent [1]*: the event that is being renamed.
- *renamedEvent [1]*: the renamed name of the event.

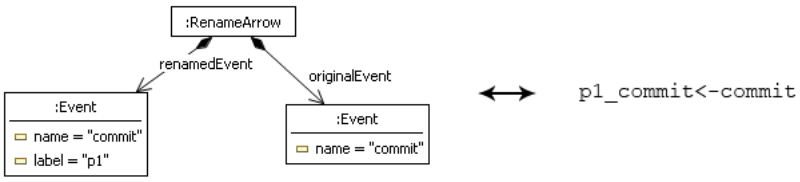


Figure 4.8: Graph to terms: *RenameArrow*

RenamingOperator

RenamingOperator implements the FDR2 syntax of renaming in the graph based representation. The correspondence between the term- and graph-based representation is shown in Figure 4.9. In the example, process P is renamed to process R. Although there are no explicit alphabets of processes in FDR2, still, the events a process engaged in need to be renamed. Assuming that P is engaged in *event1* and *event2*; with the help of *RenameArrow* they are renamed to *renamed1* and *renamed2*.

Associations

- *usedProcess [1]*: the process being renamed.
- *renameArrow [0..*]*: the set of rename arrows that rename all events the process engages in.

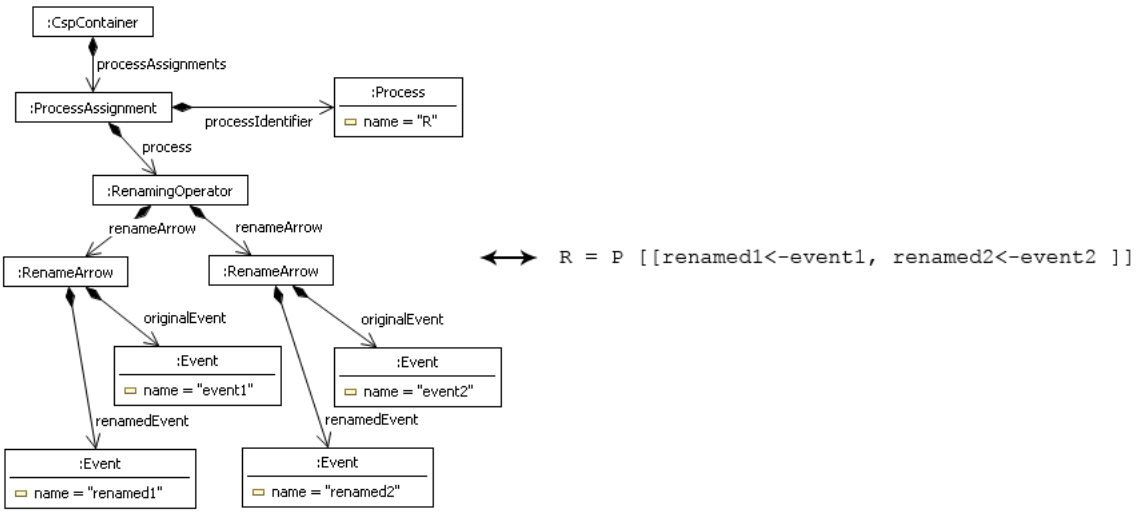


Figure 4.9: Graph to terms: *RenameOperator*

SystemExpression

SystemExpression is a special process used in FDR2 only: it is the root process of a system.

Associations

- *equation [1]*: the behaviour assigned to the root process of a system.

4.3 Semantics

The semantics of CSP is usually defined in terms of traces, failures, or divergences [Hoa85].

4.3.1 Traces

A *trace* of a process is a finite sequence of events in which the process has engaged up to some moment in time. Two events cannot occur simultaneously, one is always succeeds the other. Traces are denoted as a sequence of symbols, separated by commas enclosed in angular brackets. Let us give some examples on traces [Hoa85]:

- $\langle a, b \rangle$ consists of two events: a followed by b.
- $\langle a \rangle$ is a trace containing only event a
- $\langle\rangle$ is the empty trace containing no events
- The process `P = a -> SKIP [] b -> SKIP` can generate traces $\langle\rangle, \langle a \rangle, \langle b \rangle$.

The complete set of all possible traces of process P is denoted by the function $traces(P)$.

There are various operations available on traces, the ones used later are the following:

- *Catenation*: construct the trace from a pair of operands s and t by putting them together in this order.
- *Afters*: if $s \in traces(P)$, then P/s (*'P after s'*) represents the behaviour of P after the trace s is complete. This operator is only used as a notation for discussing behaviour of processes in abstract contexts. It represents the behaviour of P on the *assumption* that s has occurred [Ros97].

4.3.2 Divergences

With the introduction of the hiding operator, it became possible for a process to perform an infinite sequence of internal actions. An internal action is notated by τ. The state when a process enters into an infinite sequence of internal actions is called *divergence*. The process that does nothing but diverge is notated as **div** or $CHAOS$. Consider the following recursions:

```
P = a -> (P \ a)
Q = a -> SKIP
```

The behaviour shown to the environment is similar, since the traces in both cases are $\langle\rangle, \langle a\rangle$, i.e. $traces(P) = traces(Q)$. However, the internal behaviour is different: after trace $\langle a\rangle$ process P diverges.

The divergences of a process, represented by the function $divergences(P)$, are the set of traces after which the process diverges, i.e.

$$divergences(P) = \{s | s \in traces(P) \wedge (P/s) = \mathbf{div}\}$$

4.3.3 Failures

The failures model uses the concept of *refusals* to deal with the nondeterminism introduced by the hiding operator. A *refusal set* is a set of events that a process can fail to accept however long it is offered. The function $refusals(P)$ is the set of initial refusals of P. We also need to know not only what P can refuse to do after the empty trace, but also what it can refuse after any of its traces [Ros97]. A failure is a pair (s, X), where $s \in traces(P)$ and $X \in refusals(P/s)$. The possible failures of a process are defined by the function

$$failures(P) = \{(s, X) | s \in traces(P) \; and \; X \in refusals(P/s)\}$$

To demonstrate failures, consider the following processes with alphabet $\{a, b, c\}$:

```
P1 = (c -> a -> SKIP) [] b -> c -> SKIP)
P2 = ((c -> a -> SKIP) [] (b -> c -> SKIP)) \ {| c |}
```

The complete set of failures for the deterministic `P1` is

$$\{(\langle\rangle, \{\}), (\langle\rangle, \{a\}), (\langle c\rangle, \{c, b\}), (\langle b\rangle, \{a, b\}), (\langle c, a\rangle\{a, b, c\}), (\langle b, c\rangle\{a, b, c\})\}$$

The process `P2` shows how failures express nondeterminism introduced by hiding. The complete set of failures are:

$$\{(\langle\rangle, X) | X \subseteq \{b, c\}\} \cup \{(\langle a\rangle, X), (\langle b\rangle, X) | X \subseteq \{a, b, c\}\}$$

4.3.4 Refinement Relations

From these semantic notions the corresponding equivalence and refinement relations can be deduced.

- If every trace of Q is also trace of P, we say P *trace-refines* Q, written $P \sqsubseteq_T Q$ if and only if $traces(Q) \subseteq traces(P)$.
- If every trace s of Q is possible for P and every refusal after this trace is possible for P, then Q can neither accept and event or refuse one unless P does. P *failure-refines* Q: $P \sqsubseteq_F Q$ if and only if $traces(Q) \subseteq traces(P)$ and $failures(Q) \subseteq failures(P)$.
- One process *failure/divergence-refines* another, written $P \sqsubseteq_{FD} Q$ if and only if

$$failures(Q) \subseteq failures(P) \wedge divergences(Q) \subseteq divergences(P)$$

The equivalence relations are constructed as a symmetric closure on the refinement relations: $P \equiv_X Q$ if and only if $P \sqsubseteq_X Q \wedge Q \sqsubseteq_X P$ where $X = \{T, F, FD\}$.

All these relations can be used to express behaviour preservation of refactoring rules and compatibility of system components in CSP. It is important to note that the traces model is not unsuitable for systems with hiding or internal choice. It simply does not take the internal actions into account. When the observable behaviour is concerned, the traces model is the perfect choice.

Chapter 5

Semantic Mapping

The modelling language has to be provided with formal behaviour. In our case, a CSP-based semantics formalise the behaviour of the CSML. Since both domains are represented as instances of their respective metamodels, the correspondence between them is defined as a graph transformation system [DB08, BHE08]. This chapter presents this graph transformation system: Section 5.1 outlines the underlying mechanics while Sections 5.2, 5.3, 5.4 and 5.5 details the transformation.

5.1 Transformation Overview

The graph transformation system that implements the semantic mapping between the *combined structure modeling language* (CSML) and CSP is denoted by $GTS_{smc} = (P_{smc}, TG_{root})$ with the common type graph TG_{root} and a set of transformation rules P_{smc}. The CSP metamodel TG_{CSP} (Fig. 4.1) and the CSML metamodel TG_{arch} (Fig. 3.1) are subgraphs of TG_{root}. This containment is expressed as inclusion morphisms $inc_S : TG_{arch} \to TG_{root}$ and $inc_T : TG_{CSP} \to TG_{root}$.

This section is divided into two parts: Section 5.1.1 introduces the mechanics of the transformation, Section 5.1.2 presents the general rule design.

5.1.1 Transformation Mechanics

The task of GTS_{smc} is to read the software architecture model expressed in CSML and generate the corresponding CSP graph. Since the CSP graph is essentially the abstract syntax tree of the term-based CSP expression set, let us elaborate on its structure. As it was introduced in Section 4.1, a CSP process assignment consists of a declaration and a definition. The declaration provides a unique name that identifies

the process; the definition specifies its behaviour. Figure 5.1 shows a generic outline of the CSP graph: the root node is a *CspContainer* instance connected to *process declarations*. A process declaration is a *ProcessAssignment* instance that is identified by a *Process* instance contained via the *processAssignment* aggregation. Further depth is given to the tree by the connected behaviour. We call these connected behaviour-trees *ProcessExpression* subtrees.

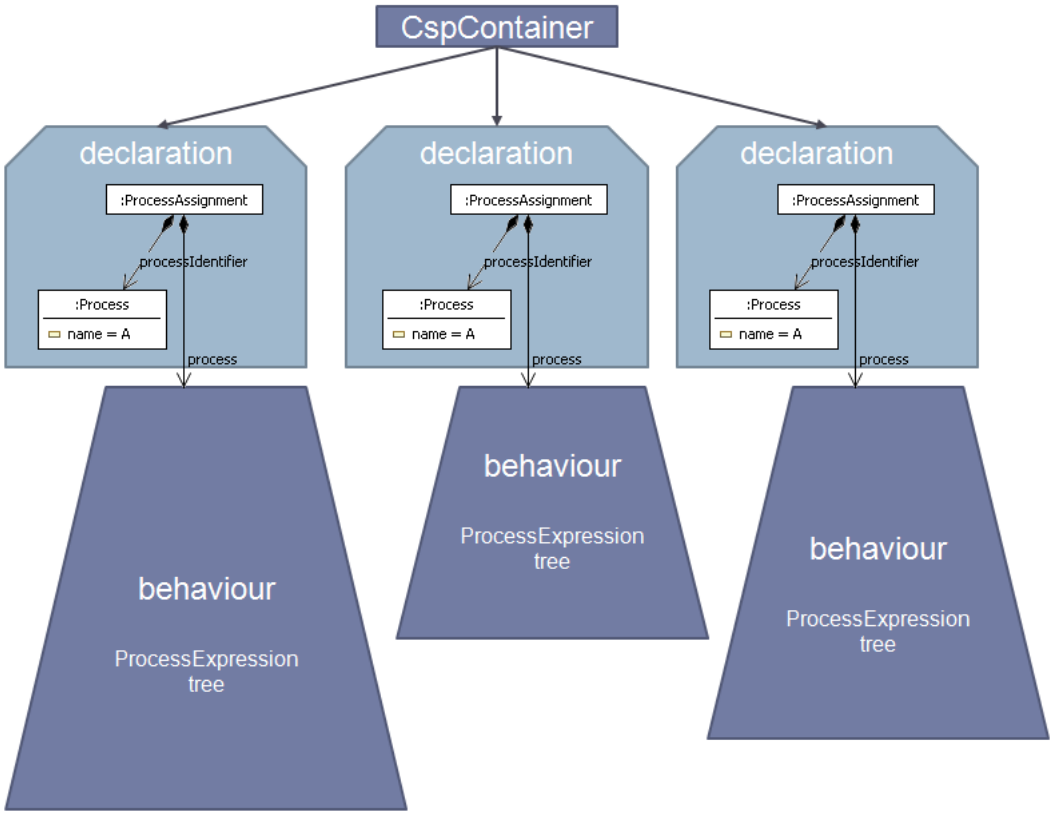


Figure 5.1: Overview of the transformation

Definition 5.1.1. ***(ProcessExpression Subtrees)*** *Given a graph* $G \in \mathbf{Graphs_{TG_{CSP}}}$ *with node* $v \in V_G$ *such that* $type(v) = ProcessAssignment$. *A subgraph* $A_v \subseteq G$ *is called a* ProcessExpression-subtree *(or PE-subtree) if there is a node* $w \in V_A$ *with edge* $e \in E_G$ *such that* $s(e) = v, t(e) = w$ *and* $type(e) = process$ *exists.*

The transformation builds the CSP-graph top-down. The process declarations are created first from components, ports, interfaces and - as shown in Section 5.1.2 - activity edges. Then, these empty declarations are provided with behaviour specifications. Since the CSP-graph is a tree, these behaviour specifications are *PE-subtrees*. One PE-subtree corresponds to one CSML element. The rule-design reflects this: P_{smc}

is sorted into named subsets $P_i \subset P_{smc}$, each responsible for transforming a certain element of CSML. It is important to observe that one rule group does not interfere with the *PE-tree* of another group. For instance, the rules responsible for building the subtree associated with a *DecisionNode* will not modify or create a subtree that describes the behaviour of a *Port*.

At the type-level, as presented in Section 5.2, the processes are derived from the components, ports and interfaces to form the framework of the semantic model. The behaviour of the system in Section 5.3 is generated from the corresponding activity diagrams. In Section 5.4, the instance level declarations are created using both the component and composite structure diagrams. The behaviour is identical in every component instance of the same type; thus, in Section 5.5 the behaviour of the component instances and channels are renamed from the type-level behaviour.

5.1.2 Rule Design

In this section we sketch the design of our transformation rules, concentrating on a single rule for a detailed presentation.

We consider the simple example rule that transforms an *ActivityEdge*. The production rule is depicted in Figure 5.2 using the concrete syntax of the participating languages. The idea behind the mapping of component behaviour is to relate an *ActivityEdge* of an activity diagram to a *Process* in CSP. The behaviour transformation follows a delocated design. First, all the edges are transformed to the corresponding process declarations by the *BhEdge* rule shown in Figure 5.3. Then, the various nodes fill the empty process definitions.

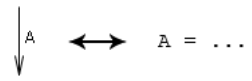


Figure 5.2: Mapping *ActivityEdge* with concrete syntax

The mapping, shown in Figure 5.2 is expressed intuitively in concrete syntax. This format will be used to give a precise overview on the outcome of the transformation.

The individual rule design of GTS_{smc} was inspired by triple graph grammars. It is important to stress that TGGs were never used for implementation; only the idea of using correspondence model for controlling the progress of transformation was harnessed. The creation of target elements, in combination with negative application conditions on the target model, allows us to retain the input model and restrict ourselves to nondeleting rules with respect to the input model. These two

properties will be important later. Earlier versions of the transformation did use a correspondence model. The mapping of behaviour with the use of a correspondence model was presented in [BH07]. However, because of complexity and performance issues, the correspondence model was abandoned, the progress of the transformation is controlled by the target model.

According to the graph transformation theory (Section 1.4.2), the production rules are defined by rule graphs, namely a left-hand side (LHS), a right-hand side (RHS) and negative application conditions (NACs). These rule graphs are object-structures that contain objects typed over EMF metamodels of UML diagrams (Fig. 3.1) as well as CSP expressions (Fig. 4.1). Object-structures are essentially attributed typed graphs.

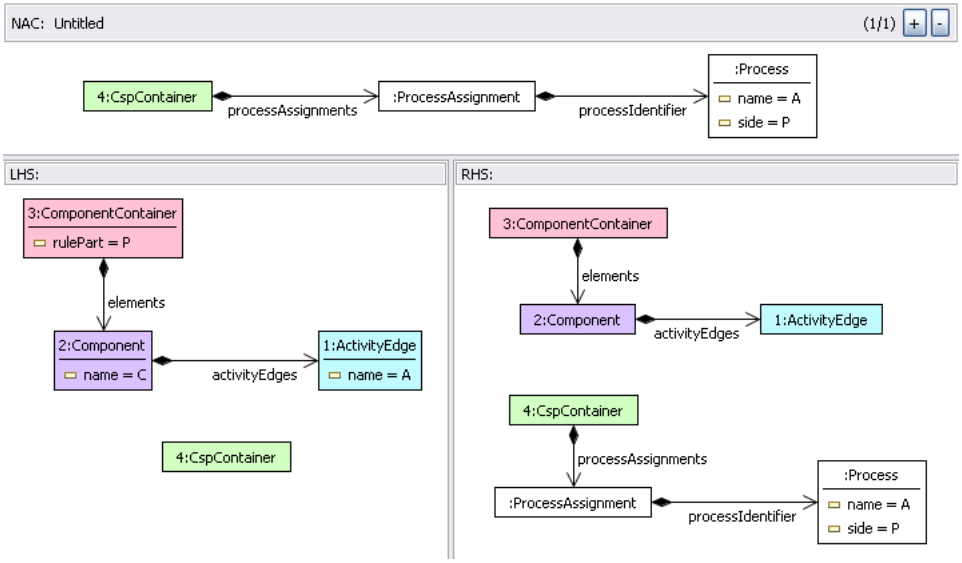


Figure 5.3: The *BhActivityEdge* rule

5.2 Type-Level Mapping

The type-level mapping realises the generation of process declarations associated with the components.

5.2.1 Components and Port Declarations

The mapping of a component and its ports is shown in Fig. 5.4. A component is mapped to a process definition with its owned behaviour (obtained from the activity diagram) and port processes in parallel. The shared event sets X and Y between

the behaviour and the port processes are generated from the engaged accept event actions and send signal actions.

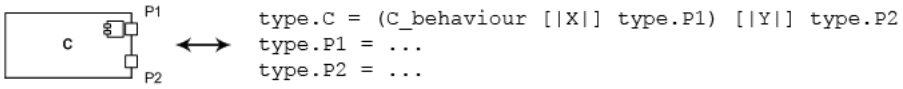


Figure 5.4: Mapping of a *Component* and its *Ports*

The rules in Figure 5.5 and 5.6 are responsible for creating the component and port processes and weaving the port processes into the component declarations.

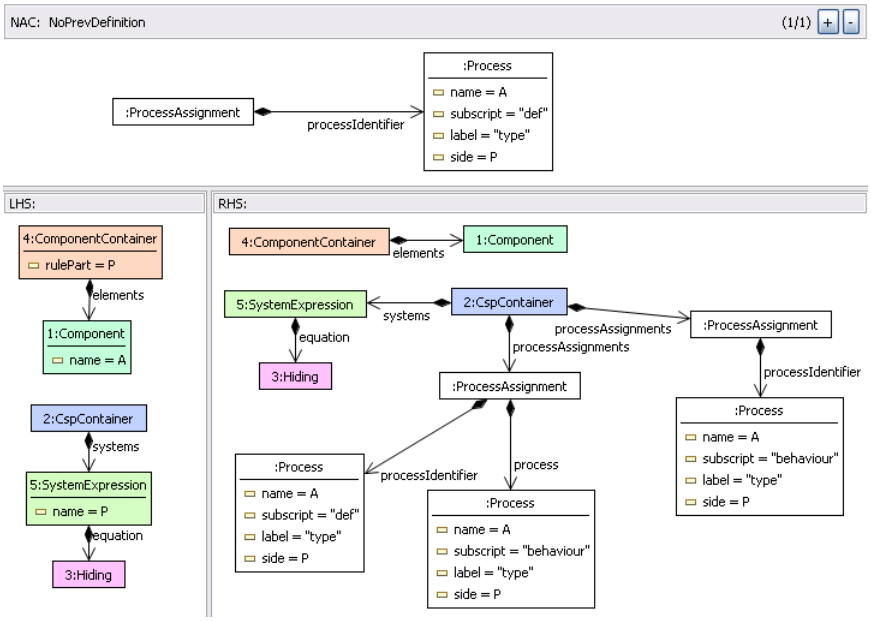


Figure 5.5: The *Component* rule

The *Component* rule in Figure 5.5 creates the process declaration and definition for the corresponding component. The NAC, defined on CSP expressions, is *self-disabling* (Def. 1.5.11). It checks the existence of similar process declarations to allow the rule application on an *essential match* only once. If no similar process assignments exist, the matched component has not been transformed yet. Thus it creates the expression `type.C_def = type.C_behaviour` and the `type.C_behaviour` process declaration. As the absence of an attribute is not matchable in EMT, the `type` label is used in both the component and port processes, to indicate a type-level process.

The *Port* rule in Figure 5.6 creates the process declarations for the corresponding ports and inserts them into the definition of the parent component. The NAC works

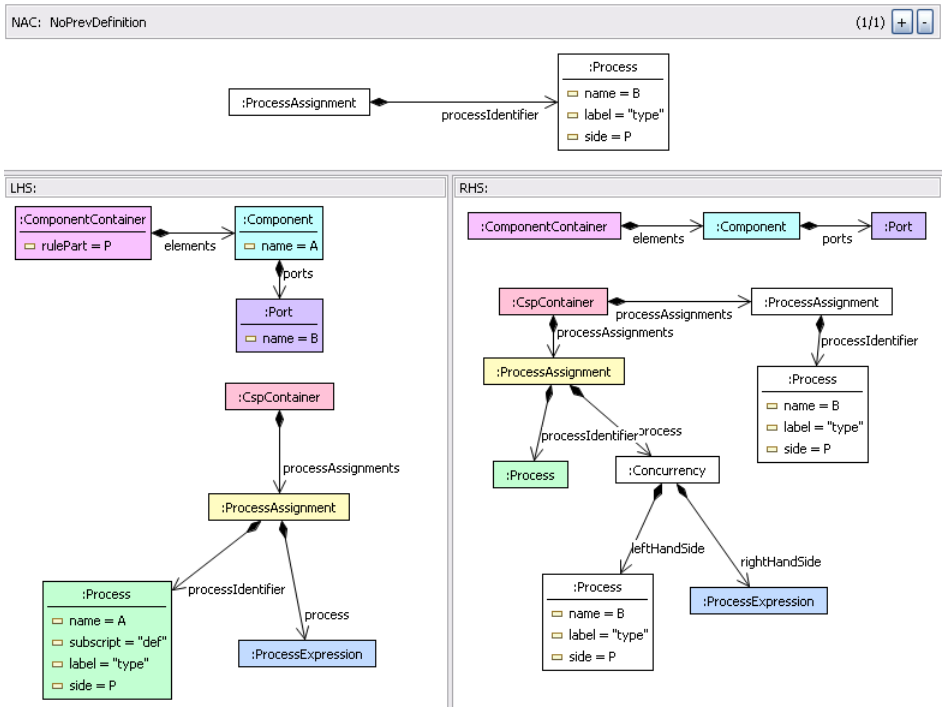


Figure 5.6: The *Port* rule

the same way as the one in the *Component* rule. Assuming the configuration in Figure 5.6, the rule transforms `P1` first with matching `A = "C"` and `B = "P1"`. `P2` is transformed in the second place with `A = "C"` and `B = "P2"`. As the component definition is not empty, the root element can be matched as a *process expression*, hence the port processes are added to the component definition in the following way:

1. `type.C = type.P1 || C_behaviour`
2. `type.C = type.P2 || (type.P1 || C_behaviour)`.

5.2.2 Ports Connected to Interfaces

Ports are mapped to processes engaging with events corresponding to their interfaces. As shown in Figure 5.7, port `type.P1`, implementing a provided interface, engages with receive and reply events. In case of a required interface, like port `type.P2` and interface `I`, the definition includes the initial send and possible return events.

The process of generating the provided interface definition of `type.P1` is shown in Figure 5.8.

- *Step 0 (Fig. 5.8(a))*: The initial setup, with the empty declaration of the `type.P1` process derived from the port.

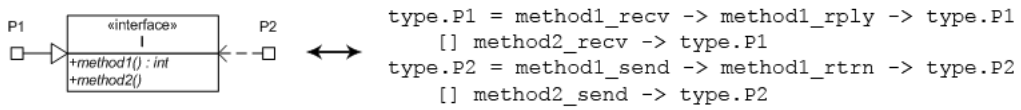


Figure 5.7: Mapping of *Port* behaviour

- *Step 1 (Fig. 5.8(b))*: As shown in Figure 5.4, port *P1* provides an interface with methods *method1* and *method2*. When an *AcceptEventAction* with *method1* name is engaged with the port, it means the reception of a function call. Thus, the definition with expression `method1_recv -> type.P1` is created.

- *Step 2 (Fig. 5.8(c))*: Similarly, an *AcceptEventAction* for *method2* can be found as well. A similar prefix is introduced through a choice: an idle port may engage in any of its communication events sequentially. Note that the graph transformation rules in *Step 1* and *2* are different. In *Step 1*, the rule needed an empty process declaration; while in *Step 2*, it searched for a declaration with a generic *ProcessExpression* as its definition, and inserted a *Choice* as a parent node.

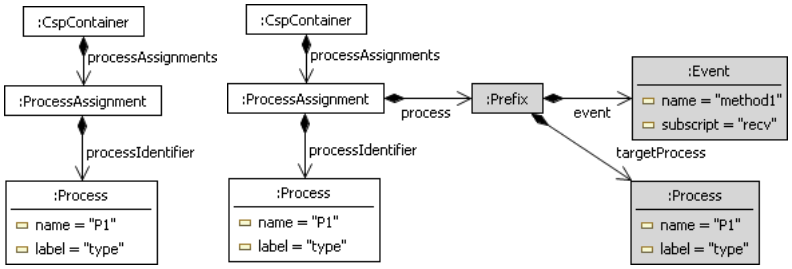
- *Step 3 (Fig. 5.8(d))*: Figure 5.4 shows that *method1* has a return value. Thus, a *SendSignalAction* engaged with the port represents the reply of *method1* (with return value) through the port. Thus, the prefix is expanded to `method1_recv -> method1_rply -> type.P1`.

Although it is not shown in Figure 5.8, the created events (`method1_recv`, `method1_rply`, `method2_recv`) are added to the corresponding event set of the concurrency operator shown in Figure 5.6 and to the channel definitions of the CSP file.

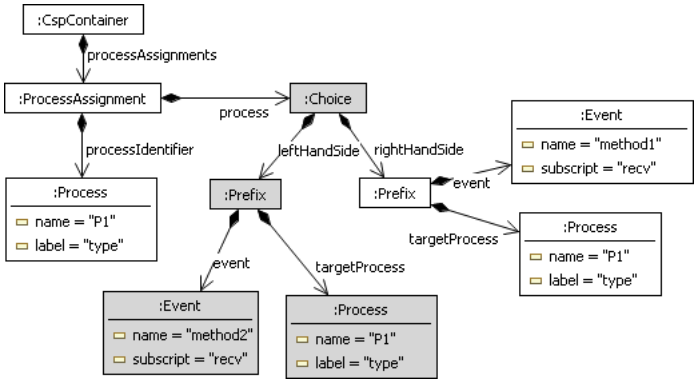
5.2.3 Interfaces

In the CSP representation, ports are façades synchronising the events between communication channels and the owned behaviour of the component. Interfaces are themselves the type-level communication behaviour. Their process counterparts contain all the possible communication events, and through event synchronisation they define the allowed order of communication.

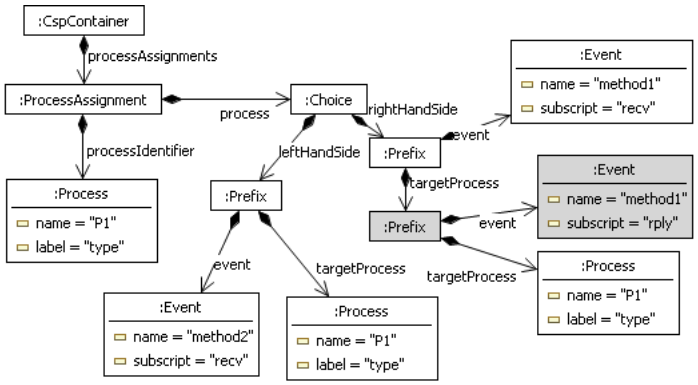
The mechanics of building the interface definitions are similar to ports. The only difference, as shown in Figure 5.9, is that the events are created in $send - receive$ and $reply - return$ pairs.



(a) Step 0 (b) Step 1



(c) Step 2



(d) Step 3

Figure 5.8: Mechanics of the provided interface transformation

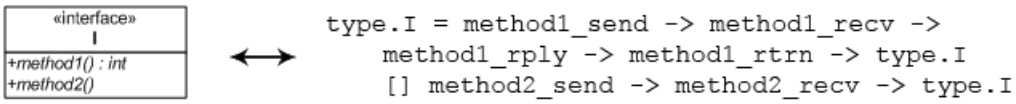


Figure 5.9: Mapping of *Interface* behaviour

5.3 Behavioural Mapping

Every component contains exactly one activity diagram as its owned behaviour, which is transformed to CSP to define the behaviour of the component.

5.3.1 Basic Behavioural Elements

As mentioned in Section 5.1.2 and shown in Figure 5.2, *ActivityEdges* are mapped to *Processes* in CSP. In this section we present the fundamentals of the mapping of behaviour.

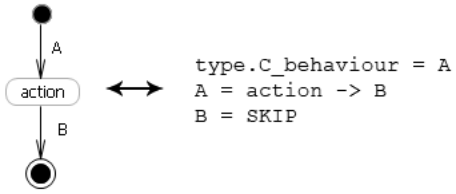
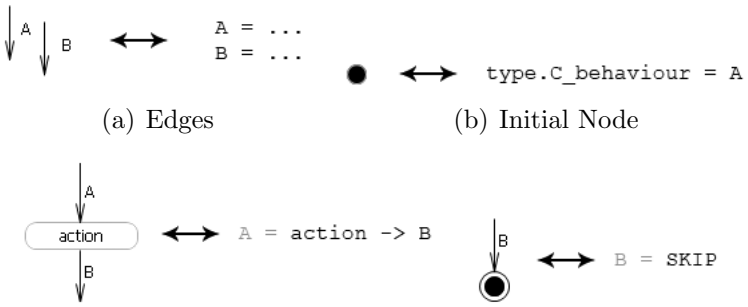


Figure 5.10: Simple behaviour transformation

A simple transformation is depicted in Figure 5.10 with its mechanics revealed in Figure 5.11. First, the edges are transformed to empty process declarations, as shown in Figure 5.11(a). Although the *InitialNode* is not mapped to anything directly, Figure 5.11(b) shows that the process of its outgoing edge fills the definition of the behaviour of the parenting component. The transformation of an *Action* according to Figure 5.11(c) is defined in terms of a new prefix expression. The definition of the process corresponding to an *ActivityEdge* ending in a *FinalNode* is the `SKIP` process as shown in Figure 5.11(d).

As the most important rule from this group, *BhAction*, presented in Figure 5.12, implements the mapping of an *Action*.

In the LHS, the *Action* and the connected edges are matched along with the process declaration corresponding to the incoming edge. In the RHS, a prefix is created. The created event is added to the channel definition. The rule is nondeleting and has a *self-disabling* NAC.



(a) Edges (b) Initial Node

(c) Action Node (d) Final Node

Figure 5.11: Basic mechanics of behaviour transformation

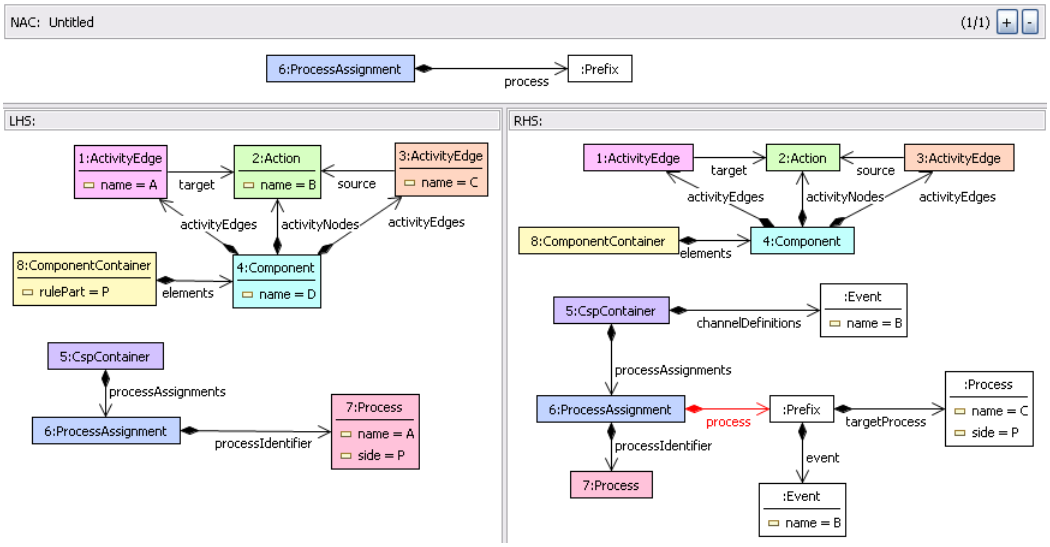


Figure 5.12: The *BhAction* rule

5.3.2 Communication Events

Both the *SendSignalActions* and *AcceptEventActions* map to similar prefixes as action nodes. As mentioned in Section 3.3, these nodes represent the various events related to function calls through the ports they engage. A concise summary of the possible communication primitives is depicted in Figure 5.13.

A *SendSignalAction* sends the function call through its engaged port typed by a required interface. A corresponding *AcceptEventAction* receives the function call in the providing component. If the function has a return value, it is replied using a *SendSignalAction* from the providing component. This return value is received in the original calling component with an *AcceptEventAction*. As presented, both actions have two possible meanings dependent on the typing of their engaged ports. Thus the transformation uses two rules for each.

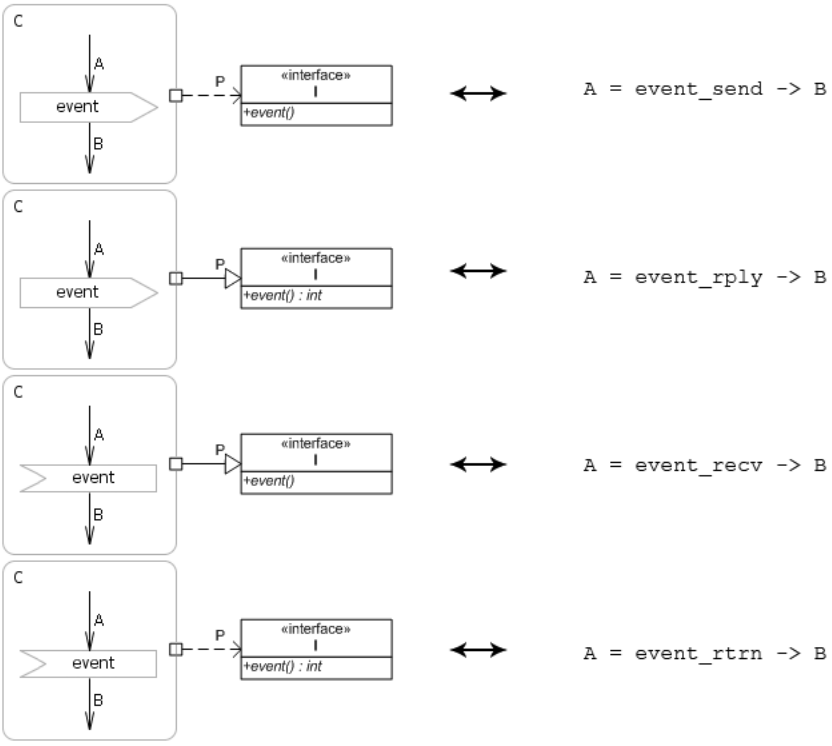


Figure 5.13: Mapping of communication events

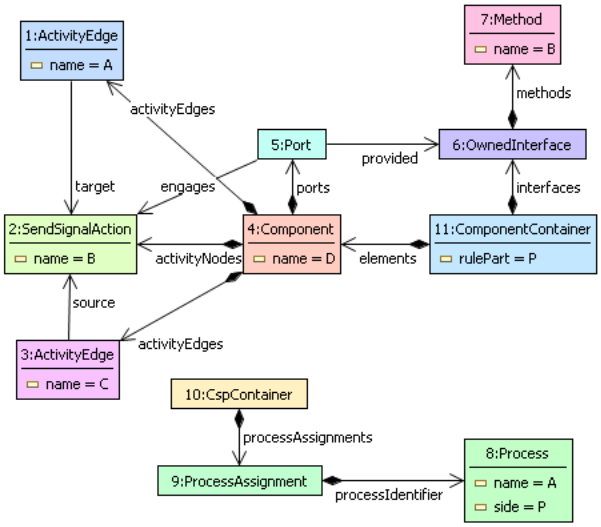


Figure 5.14: LHS of the *BhSendSignalAction1* rule

The LHS of the first of the rules transforming a *SendSignalAction* is shown in Figure 5.14. The pattern matched is similar to the one in Figure 5.12: the *SendSignalAction*, along with its incoming and outgoing edges. However, the corresponding port is also necessary here; it has to be connected to an interface with a method of

a similar name as the *SendSignalAction*. If the interface is *required* (as the one in the figure), an `eventname_send` event is created. In case of a *provided* interface, the event is a *reply* event. The rules for the accept event action are very similar to the one described.

5.3.3 Decision Node and Merge Node

The transformation of a *DecisionNode*, as depicted in Figure 5.16, is a more complicated case, although its concrete syntax is obvious. *Choice* is a binary operator, hence we have to build a binary tree bottom-up as depicted in Figure 5.15. First, the *else* branch is matched with an arbitrary edge creating the lowest element of the tree (in a dark grey shade). Then, the tree is built by adding the elements one-by-one (in a light grey shade).

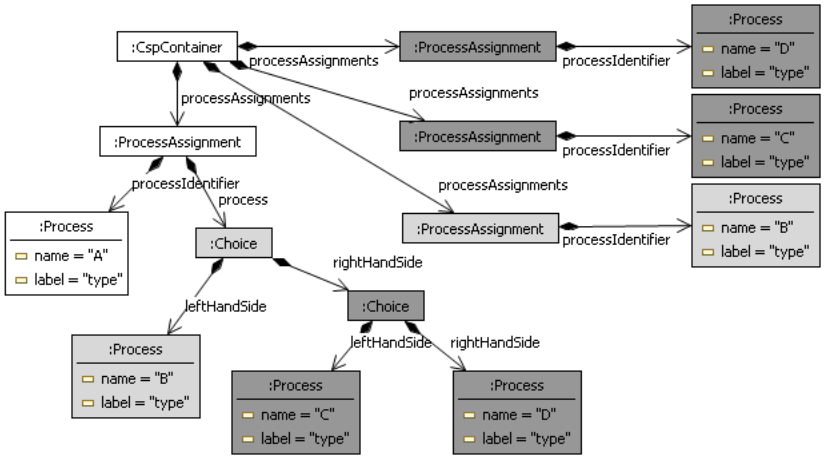


Figure 5.15: Abstract syntax tree for the result of *DecisionNode* transformation

Note that this transformation, which creates non-determinism at the syntactic level, leads to semantically equivalent processes. According to [OMG06b], *the order in which guards are evaluated is undefined* and *the modeler should arrange that each token only be chosen to traverse one outgoing edge, otherwise there will be race conditions among the outgoing edges*. Hence, the if guard conditions are disjoint, syntactically different nestings are semantically equivalent.

As mentioned, the choice of the outgoing route in the activity diagram is based on the evaluation of the guard conditions. In CSP, the external choice models this case: the environment is offered to make the decision. However, the various notions of semantics introduced in Section 4.3 deal only with the fact that the choice is external, the actual conditions are not concerned. Thus, the guard conditions themselves are

not transformed to CSP.

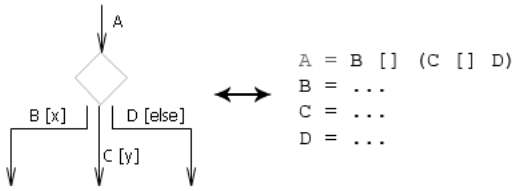


Figure 5.16: Mapping of the *DecisionNode*

The *MergeNode* is a simpler case, as illustrated in Figure 5.17. It is mapped to an equation identifying the processes corresponding to the two incoming edges.

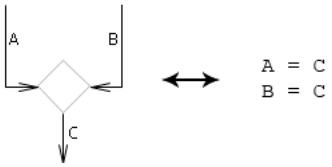


Figure 5.17: Mapping of the *MergeNode*

5.3.4 Fork Node and Join Node

ForkNode and *JoinNode* are similar in implementation mechanics to the case of the *MergeNode* and *DecisionNode*. As mentioned in Section 5.3.5, the activity diagram is well-structured in the sense of parallel nodes. The *ForkNodes* and *JoinNodes* are paired by a unique *id*, thus the transformation searches for a matching pair. The *ForkNode* in Figure 5.18 bears the same *id* as the *JoinNode* in Figure 5.19.

The mapping for the *ForkNode* is demonstrated in Figure 5.18. The concurrency operator is binary, hence by processing the outgoing edges one-by-one we create a binary abstract syntax tree of concurrency nodes the same way we did in Figure 5.15 for the *DecisionNode*. Since $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$, the different trees are semantically equivalent. As we expect the processes to terminate, we simply use the split operator and start a new process for the outgoing edge of the corresponding *JoinNode*.

The transformation of a *JoinNode* is depicted in Figure 5.19: all inbound processes terminate with success.

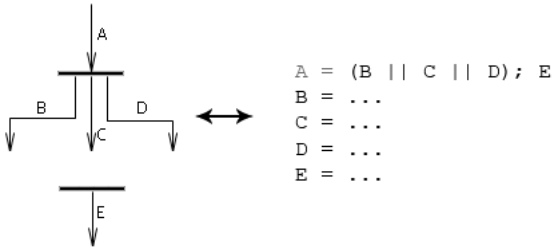


Figure 5.18: Mapping of the *ForkNode*

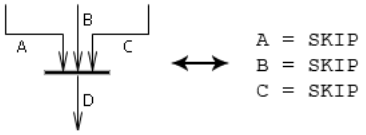


Figure 5.19: Mapping of the *JoinNode*

5.3.5 Well-Structured Activity Diagrams

After introducing the transformation of *ForkNode* and *JoinNode*, we would like to discuss the necessity of using well-structured activity diagrams with respect to parallelism. It is important to note that the difference between expressing parallelism in FDR2 and "official" CSP is only syntactic. In both cases, the parallel processes are synchronised through shared events. The difference is that in official CSP, the synchronisation is implicit, while FDR2 expresses the synchronised events explicitly.

As demonstrated in Figure 3.6(b), it is possible to have very complicated parallel configurations in an activity diagram. In such cases, there is no correspondence between fork and join nodes. However, in CSP there are two ways to synchronise previously forked processes.

1. Sequential processes can be used. If there is a parallel process $P \,||\, Q$, then both end successfully, and another process continues the control flow. It has to be stated in advance which processes participate in the parallel process. Hence, it needs to be stated as well, which processes will terminate successfully upon synchronisation. Consequently, the model of sequential processes cannot be used in the not well-formed case.

2. An explicit synchronisation event can be used as in [BH07]. In this case, there is a dedicated synchronisation event that is artificially planted into a process at the point when the activity diagram synchronises. As shown in Figure 5.20, all participating processes synchronise and one proceeds. This solution seems to be usable in not well-formed cases as well.

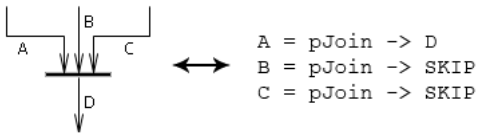


Figure 5.20: Non-structured *JoinNode* transformation

However, there is a problem with the dedicated event solution. There can be either one general synchronisation event in the system or one synchronisation event for every corresponding *JoinNode*. As the implementation is in FDR2, we have to take into account that synchronisation events need to be explicitly stated in the parallel operator. Thus, using a synchronisation event for every *JoinNode* leads back to the same problem the sequential process solution faces: the *ForkNodes* have to be paired with the *JoinNodes* and thus the activity diagrams need to be well-formed.

Using only one general synchronisation event is not allowable either. As there is no obvious connection between the *ForkNodes* and *JoinNodes* in the system, the relation between them can get very complicated. Since the synchronisation event is general, it is possible to have multiple join nodes in parallel system modules both waiting for the synchronisation event. It is not unlikely that, in this way, the system may get into deadlock or produce unwanted behaviour. Thus, the well-structured activity diagram paradigm was chosen with the semantically-feasible sequential process solution.

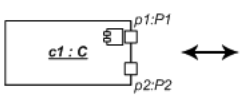
5.4 Instance-Level Mapping

The composite structure diagram models the dynamic behaviour of a software system at the architecture level. Hence, it needs to be used for checking the preservation of behaviour. The instance-level part of the semantic mapping first creates the process declarations; the behaviour is renamed from the type-level as described in Section 5.5.

5.4.1 Component Objects

To deal with multiple instances, component and port instance processes are renamed according to their instance names as shown in Figure 5.21.

Aside from the generated renaming definitions and instance name labels, the rules creating the process declarations are similar to component and port rules shown in Figures 5.5 and 5.6.




```
c1.C = (c1.C_behaviour [|X|] p1.P1) [|Y|] p2.P2
c1.C_behaviour = type.C [[ ... ]]
p1.P1 = type.P1 [[ ... ]]
p2.P2 = type.P2 [[ ... ]]
```

Figure 5.21: Mapping of a *ComponentInstance*

5.4.2 Channels

Channels are implementations of interface definitions. The channel object maps to a process declaration as shown in Figure 5.22, since its behaviour is renamed from the corresponding interface.

Figure 5.22: Mapping of a *Channel*

5.5 Renaming Rules

The only missing piece of the semantic mapping is the behaviour of the instance level objects. This behaviour is acquired by *"instantiating"* the behaviour of the components. The instantiation is basically renaming the events. The structural elements are mapped to processes, hence the distinction between the RHS and LHS of the refactoring rule is important. Events are used for checking trace refinement (as the chosen notion of behaviour preservation), thus similar event instances on both rule sides have to bear the same name.

As shown in Section 5.4, instances of structural artifacts are renamed using their instance name. Structure-based renaming, i.e. the label is the instance name of the owning object, can be used for events not present in both sides of the refactoring rule as they are considered to be deleted or created. However, because mapped event instances have to be similar on both rule sides, they need a different labelling method. Events can be relocated in the hierarchy through the refactoring, hence their renaming cannot be based on structural notions. These event instances are renamed by a unique *match* value overriding the structure-based renaming.

The three elements from CSML that map to events in CSP are the action, send signal action and accept event action. The possible event renamings are summarised in Figure 5.23. The *Action* and the two communication actions that can form two different communication primitives resolve to five different events for the non-mapped

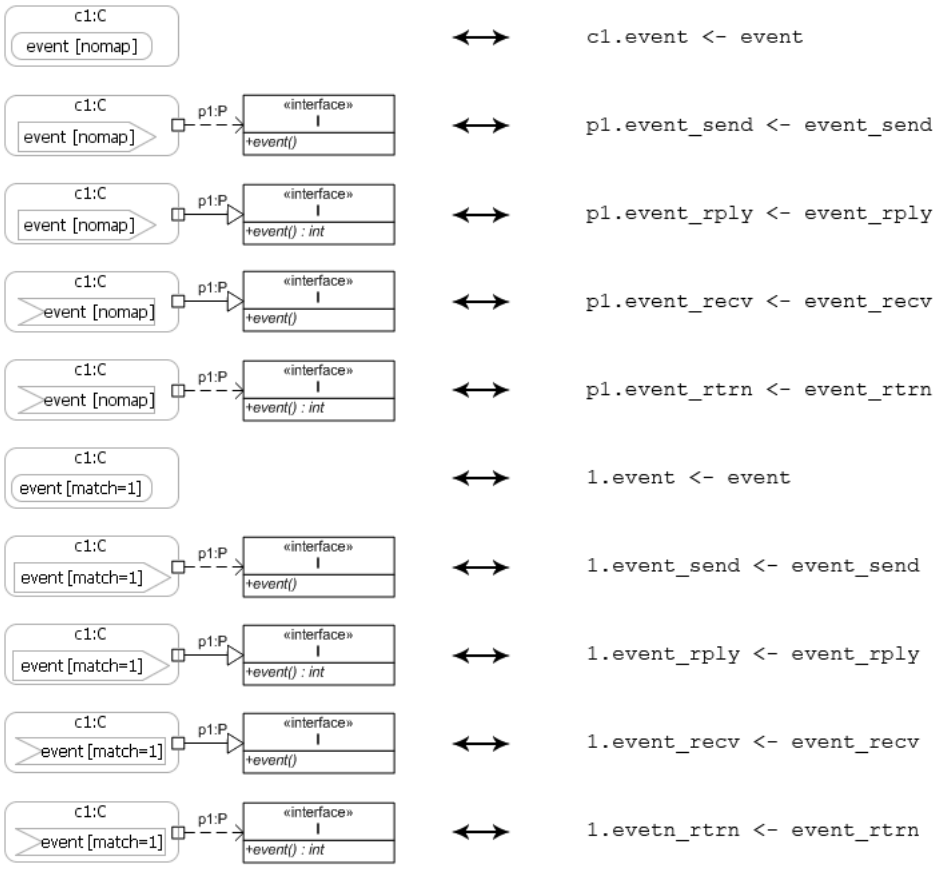


Figure 5.23: Renaming of events

case. The mapped case yields similar combinations.

As the simplest of all, the renaming of a non-mapped action is shown in Figure 5.24. The action is matched with its container and the relevant component instance. The $false$ $isMapped$ attribute ensures that the matched action is not mapped. A renaming arrow is inserted to the relevant *Renaming* operator, where the original event is renamed with a label bearing the instance name of the matched component object. The renamed event is added to the channel definition, and to the list of hidden events in the system equation. The reason for hiding is explained in Section 7.1.3. The renaming of the communication nodes work the same way, they only encompass a more complicated pattern in the LHS.

The rules for the mapped elements are slightly different. The $isMapped$ attribute in their case is $true$ with an integer *match* present holding their system-wide unique mapping identification. All mapped elements, regardless of the structural status, are labelled by the value of *match*.

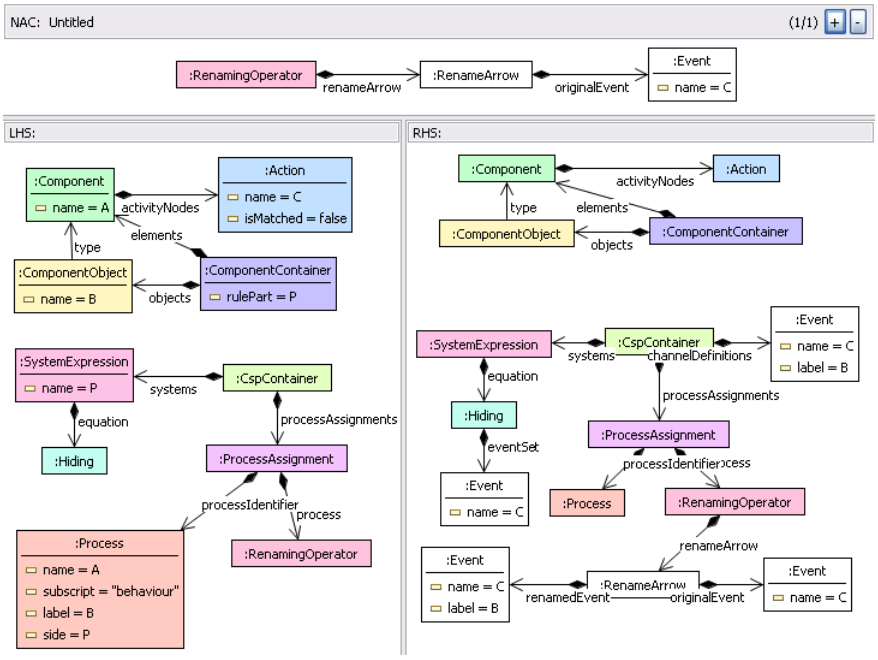


Figure 5.24: The *RnActionNoMap* rule

Chapter 6

Verification of Refactoring Rules

This chapter presents the main theoretical contributions of the thesis. A formal definition of compositionality for any total function between sets of graphs (representing software models and the semantic domain) defined by graph transformations is proposed. The formal notion of rule-level verification is defined and its correctness is proven [BHE08, BHE09b, BHE09a]. To establish compositionality, a syntactic criterion has been established for the implementation of the mappings by graph transformations with negative application conditions. Finally, the compositionality of the semantic mapping from Chapter 5 is proven.

Before presenting the technical content, a running example is introduced. This sample refactoring is demonstrated in Figure 6.1 as a graph transformation following the $H_0 \leftarrow D \rightarrow H_0'$ format of the DPO approach (Sec. 1.4.2). The idea behind the refactoring is similar to the parallelization pattern depicted in Figure 7.11. Component C calls function $a()$ through port P and then performs action b. Component D receives this call through port Q. The refactoring parallelises the actions in component C: the function call $a()$ and action b are interleaved. The behaviour of the system expressed in CSP is shown in Table 6.1 according to the transformation presented in Chapter 5 along with its trace semantics.

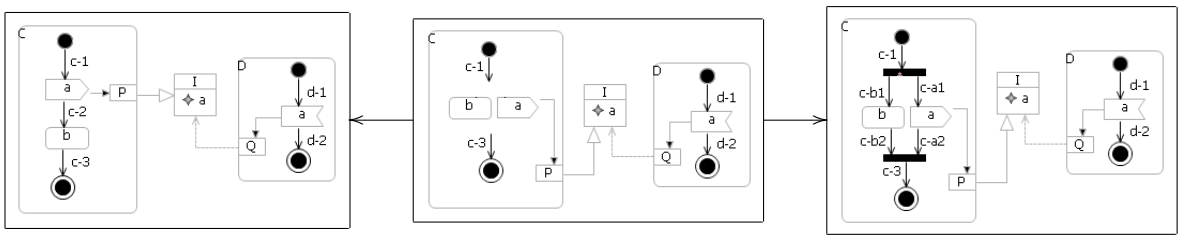


Figure 6.1: Sample Refactoring

system_L =
  type.C_L [| [a_send] |]
  type.I [| [a_recv]|] type.D
 	
type.C_L = C_behaviour
  [| [a_send]|] type.P
C_behaviour = c-1
c-1 = a_send -> c-2
c-2 = b -> c-3
c-3 = SKIP
type.P = a_send -> type.P
 	
type.D = D_behaviour
  [| [a_recv] |] type.Q
D_behaviour = d-1
d-1 = a_recv = d-2
d-2 = SKIP
type.Q = a_recv -> type.Q
 	
type.I = a_send -> a_recv
  -> type.I
	system_R =
	  type.C_R [| [a_send] |]
	  type.I [| [a_recv]|] type.D
	
	type.C_R = C_behaviour
	  [| [a_send]|] type.P
	C_behaviour = c-1
	c-1 = (c-b1 ||| c-a1); c-3
	c-b1 = b -> c-b2
	c-b2 = SKIP
	c-a1 = a_send -> c-a2
	c-a2 = SKIP
	c-3 = SKIP
	type.P = a_send -> type.P
	
	type.D = D_behaviour
	  [| [a_recv] |] type.Q
	D_behaviour = d-1
	d-1 = a_recv = d-2
	d-2 = SKIP
	type.Q = a_recv -> type.Q
	
	type.I = a_send -> a_recv
	  -> type.I

traces(type.C_L) = < a_send, b > traces(system_L) = < a_send, b, a_recv >, < a_send, a_recv, b >	traces(type.C_R) = < a_send, b >, < b, a_send > traces(system_R) = < b, a_semd, a_recv >, < a_send, b, a_recv >, < a_send, a_recv, b >

Table 6.1: Semantics of the system in Figure 6.1

6.1 Formalising Compositionality

In this section, compositionality is introduced formally. As the results proposed are generic with respect to the semantic domain, we provide a general, axiomatic definition.

Definition 6.1.1. ***(Semantic Domain)*** *A* semantic domain *is a triple* $(D, \sqsubseteq, \mathcal{C})$ *where* D *is a set,* $\sqsubseteq$ *is a partial order on* D*,* $\mathcal{C}$ *is a set of total functions* $\mathcal{C}[\,] : D \to D$*, called contexts, such that* $d \sqsubseteq e \implies \mathcal{C}[d] \sqsubseteq \mathcal{C}[e]$ *(*$\sqsubseteq$ *is closed under contexts).*

The equivalence relation $\equiv$ *is the symmetric closure of* $\sqsubseteq$*.*

Let us illustrate first, what is a context in CSP using the example in Table 6.1. The context `E[X] = System_L` of process `type.C` (i.e. `X = type.C`) is:

```
X [| [a_send] |] type.I [| [a_recv]|] type.D
```

Thus, CSP is a semantic domain $(D, \sqsubseteq, \mathcal{C})$, where D is the set of CSP process expressions and $\sqsubseteq$ can be trace, failure or divergence refinement as they are closed under context [Hoa85]. A context is a process expression `E[X]` with a single occurrence of a distinguished process variable `X`.

To motivate the definition of compositionality, we formulate the context `F(Y)` of process `type.I` and context `G(Z)` of process `type.D` as well (i.e. `Y = type.I` and `Z = type.D`).

```
F(Y)  =  type.C_L [| [a_send] |] Y [| [a_recv] |] type.D
G(Z)  =  type.C_L [| [a_send] |] type.I [| [a_recv] |] Z
```

It is important to see that the parallel operator is associative, i.e.

```
F(Y)  =  (type.C_L [| [a_send] |] Y ) [| [a_recv] |] type.D
      =  type.C_L [| [a_send] |] ( Y [| [a_recv] |] type.D)
```

Thus, we can conclude that the system is the composition of processes `type.C`, `type.I` and `type.D`. Figure 6.2 visualises the composition of the above contexts in the style of Figure 2.

However, the fact that the semantic domain is composed of different parts is a piece of trivia. The special about compositionality is that these parts of the semantic

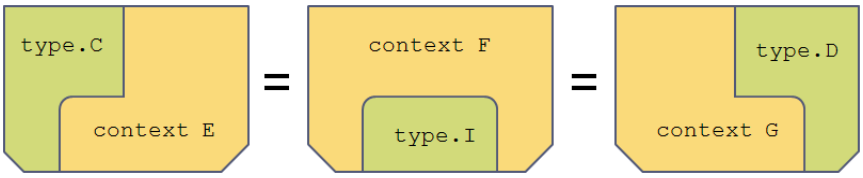


Figure 6.2: The composition of contexts

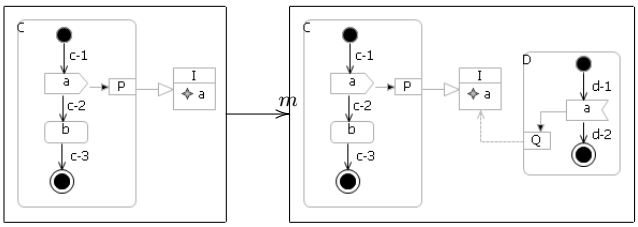


Figure 6.3: Inclusion of component C

domain are directly related to parts of the software model. For instance the behaviour of component C expressed in CSP is the process `type.C_L`, i.e. $sem(C) = \texttt{type.C\_L}$. Moreover, the inclusion $m : L \to G$ in Figure 6.3 is reflected in the semantic domain as $sem(G) \equiv E[sem(L)]$. This means that the context is uniquely determined in the semantic domain by the part of G that is not in the image of L, i.e. $G \setminus m(L)$.

Definition 6.1.2. ***(Compositionality)*** *A semantic mapping* $sem : \mathbf{Graphs_{TG}} \to (D, \sqsubseteq, \mathcal{C})$ *is* compositional *if, for any injective* $m_0 : G_0 \to H_0$ *and pushout* (1)*, there is a context* E *with* $sem(H_0) \equiv E[sem(G_0)]$ *and* $sem(H_0') \equiv E[sem(G_0')]$

$$\begin{array}{ccc} G_0 & \longrightarrow & G_0' \\ {\scriptstyle m_0}\downarrow & (1) & \downarrow{\scriptstyle m_0'} \\ H_0 & \longrightarrow & H_0' \end{array}$$

Intuitively the concept of compositionality is depicted in Figure 6.4. The semantic expression generated from H_0 and H_0' differ by only those parts that form the difference between G_0 and G_0'. The subgraph of H_0 that is not part of the image of G_0 is unchanged, so is context E.

Although concept of context was defined only for the semantic domain, it is clear, that the *part of* H_0 *that is not part of the image of* G_0 refers to a similar concept. Context in graphs (and thus in semantic domains represented as graphs) is understood via the initial pushout construction.

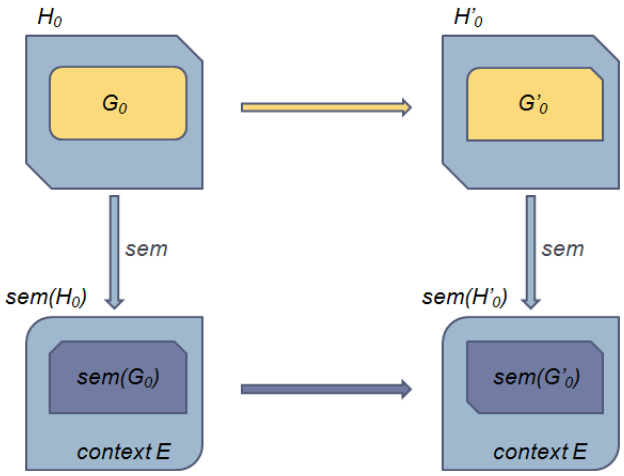


Figure 6.4: Intuitive approach to compositionality

6.2 Semantic Mapping

After the definition of semantic domain, and compositionality, we give a concise definition for semantic mapping. It consists of two steps, where the second step is an injection to the original term-based representation of the semantic domain.

Definition 6.2.1. ***(Semantic Mapping)*** *A* graph to graph semantic transformation *is a function* $sem_{g2g} : \mathbf{Graphs_{TG_{mdl}}} \rightarrow \mathbf{Graphs_{TG_{SD}}}$ *specified by a* locally confluent and terminating *graph transformation system* $GTS_{sem} = (TG_{sem}, P_{sem})$. TG_{sem} *consists of two distinguished subgraphs* TG_{mdl} *and* TG_{SD} *with* $TG_{mdl} \cap TG_{SD} = \varnothing$. *The mapping* sem_{g2g} *is defined for all* $G_0 \in \mathbf{Graphs_{TG_{mdl}}}$ *by* $sem_{g2g}(G_0) = G_n$ *if there is a transformation* $G_0 \overset{p_1}{\Rightarrow} G_1 ... G_{n-1} \overset{p_n}{\Rightarrow} G_n$ *with rules* $p_1, ..., p_n \in P_{sem}$ *which is terminating.*

A graph to term semantic transformation *is an injective function* $sem_{g2t} : \mathbf{Graphs_{TG_{SD}}} \rightarrow (D, \sqsubseteq, \mathcal{C})$.

A semantic mapping $sem : \mathbf{Graphs_{TG_{mdl}}} \rightarrow (D, \sqsubseteq, \mathcal{C})$ *is a composition of a graph to graph* sem_{g2g} *and a matching graph to term* sem_{g2t} *semantic transformations, i.e.* $sem = sem_{g2t} \circ sem_{g2g}$.

The source model is typed over TG_{mdl} while the expressions of a semantic domain $(D, \sqsubseteq, \mathcal{C})$ are represented as a typed graph typed over TG_{SD}. The graph to term transformation maps the graph representation of the semantic domain into terms (i.e. its original form).

In our case, the type graph TG_{SD} of the semantic domain is the CSP metamodel TG_{CSP} shown in Figure 4.1. The correspondence between the abstract syntax tree of

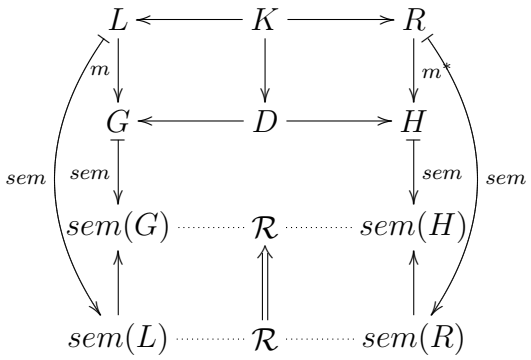


Figure 6.5: CSP correspondence for behaviour verification

CSP and the graph representation is injective. The rationale for injectivity is based on the similarity of the abstract syntax tree of CSP terms and the graph instances of the CSP metamodel.

6.3 Correctness of Rule-level Verification

In this section we prove that the method of verifying a refactoring by verifying the refactoring rule is indeed correct. The crucial condition is the compositionality of the semantic mapping, which guarantees that the semantic relation $\mathcal{R}$ (refinement or equivalence from Def. 6.1.1) is preserved under embedding of models. We will formulate the principle and prove that, assuming this property, our verification method is sound.

The overall structure is illustrated in Fig. 6.5. The original model (expressed in CSML) is given by graph G. The refactoring results in graph H by the application of rule $p : L \leftarrow K \rightarrow R$ at match m. By applying the semantic mapping sem to the LHS and RHS of the rule, we obtain the set of semantic expressions $sem(L)$ and $sem(R)$ given as typed graphs. Whenever the relation $sem(L)\ \mathcal{R}\ sem(R)$ (say $\mathcal{R}\ = \sqsubseteq$ is trace refinement of CSP, so all traces of the left processes are also traces of the right), we would like to be sure that also $sem(G)\ \mathcal{R}\ sem(H)$ (traces of $sem(G)$ are preserved in $sem(H)$).

Using our running example, the double-pushout diagram is as illustrated in Figure 6.6. The refactoring rule contains only *Component C*: the changes were introduced only to *Component C* and the refactoring rule consists of the changed parts only. Thus, we map only L and R to the semantic domain instead of G and H. The results are the processes `type.C_L` and `type.C_R`. The context `E[X]` introduced in Section 6.1 remains unchanged; hence `system_L = E(type.C_L)` and `system_R =`

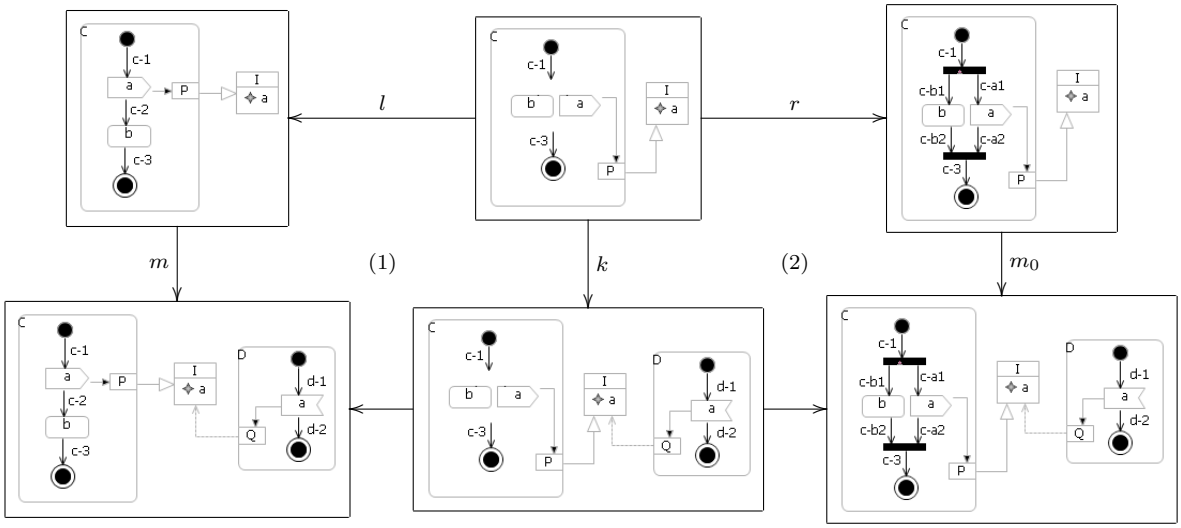


Figure 6.6: The refactoring as a DPO diagram

`E(type.C_R)`. According to the traces presented in Table 6.1 we can see that:

$$\texttt{traces(type.C\_L)} \sqsubseteq_T \texttt{traces(type.C\_R)}$$

According to the forthcoming Theorem 6.3.1, we can conclude that

$$\texttt{traces(E(type.C\_L))} \sqsubseteq \texttt{traces(E(type.C\_R))} \implies \texttt{System\_L} \sqsubseteq_T \texttt{System\_R}$$

Compositionality (Def. 6.1.2) applies where G_0 is the interface graph of a rule and G_0' is either the left or the right-hand side. In this case, the set of semantic expressions generated from G_0' contains the one derived from H_0' up to equivalence, while the context E is uniquely determined by the context graph C. The proof also relies on the fact that semantic relation $\mathcal{R}$ is closed under context.

Theorem 6.3.1. ***(Correctness of Rule-Based Verification Based on Compositionality)*** *Given a compositional semantic mapping* $sem : \mathbf{Graphs}_{\mathbf{TG_{mdl}}} \to (D, \sqsubseteq, \mathcal{C})$. *Then, for all transformations* $G \overset{p,m}{\Longrightarrow} H$, *it holds that* $sem(L) \sqsubseteq sem(R)$ *implies* $sem(G) \sqsubseteq sem(H)$.

Proof. As m is injective, pushout (1) implies that k is injective as well. Thus $sem(D) \equiv E[sem(K)]$ and $sem(G) \equiv E[sem(L)]$. Since m is injective, k and m_0 are injective as well because of pushouts (1) and (2). Similarly, the injectivity of k and pushout (2) implies that $sem(H) \equiv E[sem(R)]$.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle k} & (2) & \downarrow{\scriptstyle m_0} \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Now, $E[sem(L)] \sqsubseteq E[sem(R)]$ since $sem(L) \sqsubseteq sem(R)$ and $\sqsubseteq$ is closed under context. Hence $sem(G) \equiv E[sem(L)] \sqsubseteq E[sem(R)] \equiv sem(H)$ □

The statements in Theorem 6.3.1 also hold for the relation $\equiv$, obtained as the symmetric closure of $\sqsubseteq$.

6.4 Basic Graph Transformations

After giving the definition of compositionality in Section 6.1, we prove a condition of compositionality of semantic mappings specified by graph transformations without negative application conditions in this section. We assume that the semantic mapping *sem* is of Definition 6.2.1.

Definition 6.4.1. ***(Nontermination)*** *A graph G is* nonterminating *with respect to a typed graph transformation system $GTS = (TG, P)$ if $\exists p \in P$ that is applicable to G. The notation for a non-terminating graph is $G_{GTS} \gg$.*

Note that the *termination* as defined in Section 1.5.3 is of GT systems. The *nontermination* above deals with a single graph. The last graph H_n of a terminating graph transformation $(t_n : H \overset{n}{\Rightarrow} H_n)_{n\in\mathbb{N}}$ with no applicable rules left, is the exact opposite of a *nonterminating* graph, it is terminating.

For a transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ we create a boundary graph B and a context graph C through an initial pushout. The boundary graph is the smallest subgraph of G_0 which contains the identification points and dangling points of m_0.

Given a graph H, the gluing of the context C and graph G. The *separability* of semantics intuitively means that, if H is *nonterminating* with respect to the semantic mapping (i.e. $\exists p \in P_{sem}$ applicable to H), then the reason for the nontermination of H is that either C or G is *nonterminating* (i.e. this p is applicable to either C or G). In a *nonseparable* case, it is possible to have H *nonterminating* with $r \in P$, but both C and G is terminating, r is not applicable to either of them.

Definition 6.4.2. ***(Separable Graph Transformation System)*** *A typed graph transformation system $GTS = (TG, P)$ is* separable *with respect to a set of constraints*

$\mathcal{X}_\mathcal{S}$ *on* $\mathbf{Graphs_{TG}}$ *if for all pushouts* (1) *with* $G \models \mathcal{X}_\mathcal{S}$ *and* $C \models \mathcal{X}_\mathcal{S}$ *it holds that if* $H_{GTS} \gg$ *then either* $G_{GTS} \gg$ *or* $C_{GTS} \gg$.

$$\begin{array}{ccc} B & \longrightarrow & G \\ \downarrow & (1) & \downarrow \\ C & \longrightarrow & H \end{array}$$

As mentioned, the notion of context in graphs is expressed through the initial pushout construction. It is not trivial however, that in the term-based representation of this context will remain the same. Thus, we require this property from the sem_{g2t} transformation.

Definition 6.4.3. ***(Context Preservation of*** $\mathbf{sem_{g2t}}$***)*** *The graph to term transformation* $sem_{g2t} : \mathbf{Graph_{TG_{SD}}} \rightarrow (D, \sqsubseteq, \mathcal{C})$ *is context preserving, if for each initial pushout* (1) *with* $B, L, C, G \in \mathbf{Graphs_{TG_{SD}}}$ *it holds that* $sem_{t2g}(G) \equiv sem_{t2g}(C)[sem_{t2g}(L)]$.

$$\begin{array}{ccc} B & \longrightarrow & L \\ \downarrow & (1) & \downarrow k_0 \\ C & \longrightarrow & G \end{array}$$

The next definition is *IPO compatibility* of semantic mappings. While *compositionality* was defined through an unknown context E, *IPO compatiblity* defines it through the semantics of the context graph.

Definition 6.4.4. ***(IPO Compatibility)*** *A semantic mapping* $sem : \mathbf{Graphs_{TG_{mdl}}} \rightarrow (D, \sqsubseteq, \mathcal{C})$ *is* initial pushout compatible *(IPO compatible) if for any injective* $m_0 : G_0 \rightarrow H_0$ *and initial pushout* (1) *over* m_0 *we have* $sem(H_0) \equiv sem(C)[sem(G_0)]$.

$$\begin{array}{ccc} B & \longrightarrow & G_0 \\ \downarrow & (1) & \downarrow m_0 \\ C & \longrightarrow & H_0 \end{array}$$

Lemma 6.4.1. ***(Compositionality of IPO Compatibility)*** *Given a double pushout diagram* $(2a) - (2b)$ *with injective morphism* m_0. *If a semantic mapping* $sem : \mathbf{Graphs_{TG_{mdl}}} \rightarrow (D, \sqsubseteq, \mathcal{C})$ *over* m_0 *is IPO compatible, then it is also compositional.*

$$\begin{array}{ccccc} G_0 & \xleftarrow{l} & K & \xrightarrow{r} & G_0' \\ {\scriptstyle m_0}\downarrow & (2a) & \downarrow{\scriptstyle k} & (2b) & \downarrow{\scriptstyle m_0'} \\ H_0 & \longleftarrow & D & \longrightarrow & H_0' \end{array}$$

Proof. Given pushouts $(2a) - (2b)$ with injective morphism $m_0 : G_0 \to H_0$. The existence of initial pushout (1) over m_0 (Def. A.1.5) follows from the existence of contexts (Thm. A.1.2). The *closure property of initial pushouts* (Lemma. A.1.1) implies that the composition of pushout (1) + $(2a)$ defined as pushout (3) is also initial over m_0. The composition of initial pushout (3) with pushout $(2b)$ is an initial pushout over m'_0

$$\begin{array}{ccc} B & \longrightarrow & G_0 \\ \downarrow & (1) & \downarrow m_0 \\ C & \longrightarrow & H_0 \end{array} \qquad \begin{array}{ccccc} B & \xrightarrow{b^*} & K & \xrightarrow{r} & G'_0 \\ \downarrow & (3) & \downarrow k & (2b) & \downarrow m'_0 \\ C & \xrightarrow[c^*]{} & D & \longrightarrow & H'_0 \end{array}$$

Since *sem* is IPO-compatible with (1), $sem(H_0) \equiv sem(C)[sem(G_0)]$. As (3) + $(2b)$ is also an initial pushout, *sem* is compatible with it, and thus $sem(H'_0) \equiv sem(C)[sem(G'_0)]$. Hence *sem* is compositional with $E = sem(C)$. □

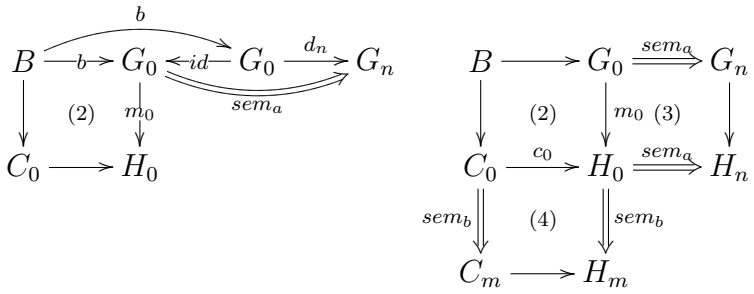
The definition of *initial-preserving* graph transformations is inspired by the world of Triple Graph Grammars [Sch94]. We assume an implicit source model left untouched by the transformation, while the transformation constructs the target model.

Definition 6.4.5. ***(Initial-Preserving)*** *A (typed) graph transformation* $t : G_0 \stackrel{*}{\Longrightarrow} G_n$ *is* initial-preserving *if it is nondeleting with respect to its initial graph* G_0*.*

Note that *initial-preservation* is not identical to *nondeletion*, as elements of the target model may be deleted or modified through the transformation process.

Theorem 6.4.1. ***(Basic Compositionality Theorem)*** *A semantic mapping* $sem = sem_{g2t} \circ sem_{g2g} : \mathbf{Graphs_{TG_{mdl}}} \to (D, \sqsubseteq, \mathcal{C})$ *is compositional if* GTS_{sem} *is* separable *with constraints* $\mathcal{X}_{\mathcal{S}}$*,* sem_{g2t} *is* context preserving *and for all* $G_0 \in \mathbf{Graphs_{TG_{mdl}}}$ *the transformation* $sem_{g2g} : G_0 \stackrel{*}{\Rightarrow} G_n$ *with* $sem_{g2g}(G_0) = G_n$ *is initial preserving.*

Proof. The main argument is based on the *Embedding Theorem* (Def. 1.5.2). For the transformation $sem_{g2g} : G_0 \stackrel{*}{\Rightarrow} G_n$ we create a boundary graph B and context graph C. The boundary graph is the smallest subgraph of G_0 which contains the identification points and dangling points of m_0. Pushout (2) is the initial pushout of m_0.

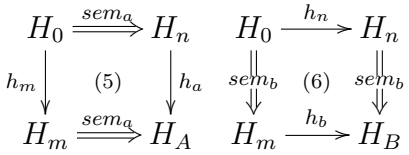


Since sem_{g2g} is *initial-preserving* the consistency diagram (Def. 1.5.5) above can be used with initial pushout (2). G_0 replaces D as it is preserved throughout the transformation. Hence, m_0 is consistent with respect to sem_{g2g} and there is an *extension diagram* over sem_{g2g} and m_0 (Def. 1.5.3). Transformations sem_a and sem_b only denote particular rule application orders of transformation sem_{g2g}.

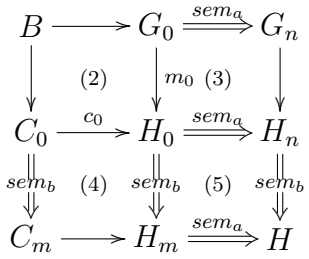
This essentially means that (3) is a pushout and H_n is the pushout object of sem_a and m_0, thus can be determined without applying the transformation sem_{g2g} on H_0.

While $sem_a(G_0) = G_n$ and thus $G_0 \overset{*}{\Rightarrow} G_n$ is terminating, H_n is possibly not terminating with respect to GTS_{sem}. The parts of H_0 not present in G_0 were not transformed to the semantic domain by the rule applications of sem_a, and the reasoning above holds for C_0 as well. The extension diagram over C_0 is pushout (4) and $sem_b(C_0) = C_m$. The termination of H_m with respect to GTS_{sem} is also unknown.

According to the Concurrency Theorem (Def. 1.5.1) the concurrent production can be created for both $H_0 \overset{*}{\Rightarrow} H_m$ and $H_0 \overset{*}{\Rightarrow} H_n$. Since the transformation is *initial-preserving*, the resulting morphisms h_n and h_m are inclusions (or identities) and the extension diagrams (5) and (6) exist. Since pushouts are unique, (5) = (6) and thus $H_A = H_B = H$.

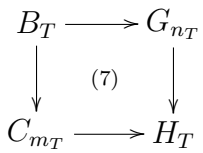


This leads to the diagram below. Since GTS_{sem} is *separable*, $G_n \models \mathcal{X}_S$ and $C_m \models \mathcal{X}_S$ and they are terminating (i.e. no semantic rule applicable), H must be also terminating. If H is terminating, that means $sem_{g2g}(H_0) = H$.



According to the composition property of pushouts (Def. A.1.1), (2) + (3) and (4)+(5) are pushouts and thus the big (2)+(3)+(4)+(5) = (6a) square is a pushout as well. Since H is a pushout object, $H \cong G_n +_B C_m$.

However, following the algebraic graph transformation approach (Sec. 1.7.1) in general, sem_{g2g} produces a graph that contains both the architectural model ($TG_{mdl} \subset TG$) and the graph representation of semantic expressions ($TG_{SD} \subset TG$). Pushout (6a) has to be restricted by type graph $TG_{SD} \subset TG$ that contains only semantic elements.



This leads, according to the *van-Kampen* square property [EEPT06], to pushout (7) typed over TG_{SD}. $B_T = \emptyset$ because $TG_{mdl} \cap TG_{SD} = \varnothing$ and $type(B) \subseteq TG_{mdl}$. Thus pushout (7) is a coproduct in $\mathbf{Graphs_{TG_{SD}}}$ and thus H_0 is the initial pushout object of C_0 and G_0.

The results of the sem_{g2g} transformations are graphs. The only task left is to apply the sem_{g2t} mapping. As sem_{g2t} is *context preserving*, we can start with equation (6.1). In step (6.2), the definition of $sem = sem_{g2t} \circ sem_{g2g}$ is applied.

$$sem_{g2t}(sem_{g2g}(H_0)) \equiv sem_{g2t}(sem_{g2g}(C_0))[sem_{g2t}(sem_{g2g}(G_0))] \quad (6.1)$$

$$sem(H_0) \equiv sem(C_0)[sem(G_0)] \quad (6.2)$$

Consequently, *sem* is *IPO compatible* and according to Lemma 6.4.1 it is also compositional. □

6.5 Graph Transformations with NACs

In Section 6.4 we were concerned with a semantic mapping defined by graph transformations without NACs. However in real-life cases, to control the transformation, negative application conditions need to be used as well. In this section we assume that the semantic mapping *sem* is of Definition 6.2.1 but the set of typed graph productions P_{sem} contains $NAC(n_j)$ on each $p_j \in P_{sem}$ with $j \in J$.

The definition of *separable* semantics (Definition 6.4.2) carries over to this section with the presence of negative application conditions allowed. Before the establishment of Theorem 6.5.1, the necessary definitions are presented.

Definition 6.5.1. ***(Created Points)*** *Given a (typed) graph production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$. *The created points* CP *are those nodes and edges in* R *that are created by* p, *i.e.* $CP = V_R \backslash r_V(V_K) \cup E_R \backslash r_E(E_K)$.

Definition 6.5.2. ***(Deleted Points)*** *Given a (typed) graph production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$. *The deleted points* DP *are those nodes and edges in* R *that are deleted by* p, *i.e.* $CP = V_L \backslash l_V(V_K) \cup E_L \backslash l_E(E_K)$.

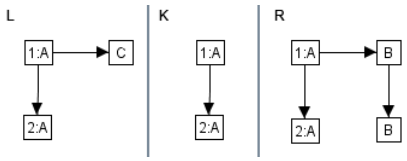


Figure 6.7: Production rule with created points

The concept of created points is demonstrated in Figure 6.7. Since the C node is deleted, it is the only deleted point of the production rule. The gluing points (Def. 1.4.6) are the two A nodes as they are preserved by the rule in Fig. 6.7. The created points are the B nodes on the right hand side of the rule.

It is possible to have a situation, where the B nodes are always - if present - gluing points in every production rule of a graph transformation system. This means that the node type B is such a special type that its instances are never deleted. This observation leads to the definition of *constant types* whose instances are already present in the start graph and are not deleted throughout the transformation.

Definition 6.5.3. ***(Constant Types)*** *Given a typed graph transformation system* $GTS = (TG, P)$. Constant types $CT \subseteq TG = (V_{CT}, E_{CT})$ *are those nodes and edges in the type graph* TG, *whose instances are not deleted or modified by any production*

$p \in P$. *i.e.* $CT = \{v \in V_{TG} \mid \forall p \in P, \forall w \in LHS(p_V) : v = type_V(w) \implies w \in GP_p\} \cup \{e \in E_{TG} \mid \forall p \in P, \forall f \in LHS(p_E) : e = type_E(f) \implies f \in GP_p\}$.

In an instance graph, constant points are those nodes and edges that are of a constant type.

A graph G typed over TG with its constant types $CT \subseteq TG$ is non-constant*, if G does not contain constant points.*

The definition of *constructive transformations* are inspired by TGGs. While the NACs contain only non-constant elements, the initial graph consists of constant points exclusively. This way the NACs concentrate on the target elements of the transformation.

Definition 6.5.4. ***(Constructive Transformation)*** *A graph transformation system $GTS = (TG, P)$ with constant types $CT \subseteq TG$, NACs, a start graph G_0 and embedding $m_0 : G_0 \to H_0$ is* constructive *if*

1. *G_0 and H_0 contains only constant points, i.e. $type(G_0), type(H_0) \subseteq CT$.*
2. *all NACs are injective and non-constant, i.e. for all $(p, NAC_p) \in P$, $NAC_p = NAC(n)$ with $n : L \to N$ we have $\exists x \in N \setminus n(L)$ with $type(x) \notin CT$*

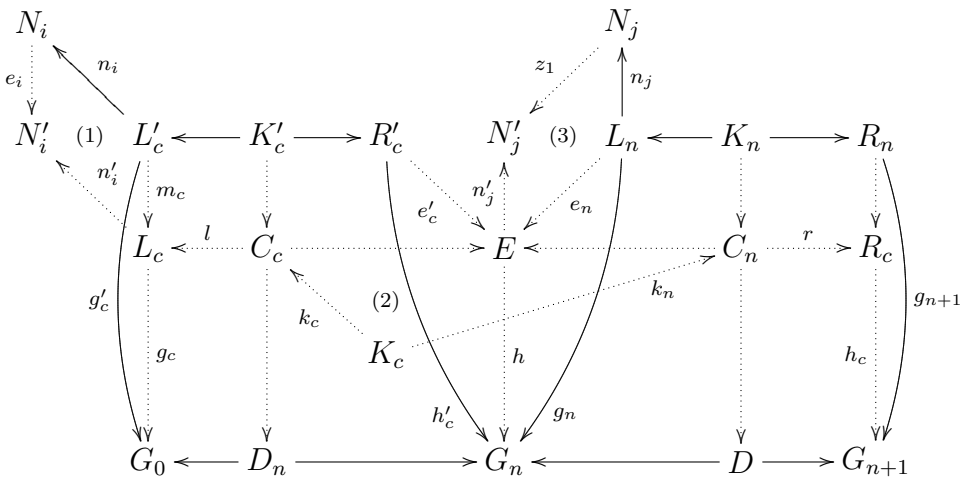
Corollary 6.5.1. *A* constructive *graph transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ with NACs is also* initial-preserving *because G consists of constant points that are not deleted through the transformation.*

Lemma 6.5.1. ***(The Concurrent NAC of Non-Constant NACs is Non-Constant)*** *Given a graph transformation system $GTS = (TG, P)$ with constant types $CT \subseteq TG$ and a graph transformation $t : G_0 \implies G_n$. The concurrent rule of t according to Thm. 1.6.1 is p_c. The concurrent NAC NAC_{p_c} of p_c is non-constant provided that $\forall (p, NAC_p) \in P$, NAC_p is non-constant.*

Proof. The proof is by mathematical induction over the length of $t : G_0 \overset{*}{\Rightarrow} G_n$.

Basis. $n = 1$. We have $p_c = p_0$ which has the property by assumption.

Induction Step. Consider $t_n : G_0 \overset{n}{\Rightarrow} G_n \Rightarrow G_{n+1}$ via the rules $p_0, p_1, ..., p_n$. We can assume by induction that $p'_c = N_i \overset{n_i}{\leftarrow} L'_c \leftarrow K'_c \rightarrow R'_c$ (the concurrent rule for $p_0, p_1, ..., p_{n-1}$) and $p_n : N_j \overset{n_j}{\leftarrow} L_n \leftarrow K_n \rightarrow R_n$ have non-constant NACs. We have to show that all NACs for p_c are non-constant.

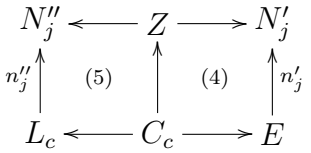


According to the synthesis construction of *Concurrency Theorem with NACs* the concurrent rule p_c with NACs induced by $G_0 \overset{n+1}{\Longrightarrow} G_{n+1}$ is $p_c = L_c \xleftarrow{l \circ k_c} K_c \xrightarrow{r \circ k_n} R_c$ (with match $g_c : L_c \to G_0$, comatch $h_c : R_c \to G_{n+1}$). The concurrent NAC_{p_c} consists of two parts.

Case 1 $n'_i : L_C \to N'_i$ defined by $n_i : L'_C \to N_i$ from p'_c.

By *Assumption 2* of *constructiveness* we have $\exists x_i \in N_i \setminus n_i(L'_C)$ with $type(x_i) \notin CT$. Let $x'_i = e_i(x_i)$ such that $type(x'_i) = type(x_i) \notin CT$. Moreover $x'_i \in N'_i \setminus n'_i(L_C)$ because otherwise pushout and pullback (1) implies that $\exists y_i \in L'_C$ with $n_i(y_i) = x_i$ and hence $x_i \in n_i(L'_C)$ which is a contradiction. Thus n'_i is non-constant.

Case 2 $n''_j : L_C \to N''_j$ defined by $n_j : L_n \to N_j$ with $n'_j : E \to N_j$ through pushouts (3) − (5).



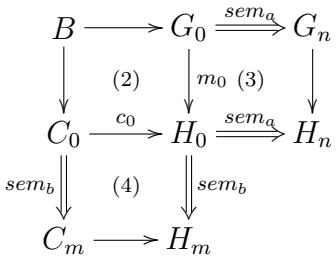
If the pushout complement C_c of (4) does not exists, the induced NAC is always true.

By assumption on p_n we have $x_j \in N_j \setminus n_j(L_n)$ with $type(x_j) \notin CT$. Because (3) is a pushout and pullback, $\exists x'_j = z_1(x_j) \in N_j \setminus n'_j(E)$ and $type(x'_j) = type(x_j) \notin CT$. Also $\exists y_j \in Z \setminus C_c$ with $(Z \to N'_j)(y_j) = x'_j$ using pushout (4) with $type(y_j) = type(x'_j) \notin CT$. And finally $\exists x''_j = (Z \to N''_j)(y_j) \in N''_j \setminus n''_j(L_C)$ because (5) is

a pushout and pullback with type $type(x''_j) = type(y_j) \notin CT$. Thus n''_j is non-constant. □

Theorem 6.5.1. ***(Compositionality Theorem with NACs)*** *Given a semantic mapping* $sem = sem_{g2t} \circ sem_{g2g} : Graphs_{TG_{mdl}} \to (D, \sqsubseteq, \mathcal{C})$ *with constant types* $CT \subseteq TG_{sem}$ *and* sem_{g2g} *containing NACs. Then, it is compositional if* GTS_{sem} *is* separable *with constraints* $\mathcal{X}_\mathcal{S}$, sem_{g2t} *is* context preserving *and* sem_{g2g} *is constructive.*

Proof. The proof is based on the *Basic Compositionality Theorem* (Thm. 6.4.1). In order to apply the *Embedding Theorem* in the proof of Theorem 6.4.1, we have to show, that the extension diagrams over m_0 and c_0 exist in the presence of NACs.



As the equivalent left NACs can be constructed from the right NACs (Lemma 1.6.1), the NACs throughout this proof are assumed to be left NACs, if not explicitly stated on the contrary.

The extension diagram exists in case of NACs, if the transformation not only boundary-consistent (Def. 1.5.5), but also NAC-consistent (Def. 1.6.7). According to the synthesis construction of *Concurrency Theorem* a concurrent rule p_c with a concurrent match g_c exists (Thm. 1.6.1). The concurrent rule p_c is basically the merge of all rules of a specific rule application order in $sem : G_0 \overset{*}{\Rightarrow} G_n$ such that the target graph G_n is produced by the application of p_c on the source graph G_0. In graph transformations containing NACs, a concurrent NAC_{p_c} exists for the concurrent rule p_c. To achieve NAC-consistency, we have to show, that $k_0 \circ g_c \models NAC_{p_c}$ with NAC_{p_c} being the concurrent NAC, g_c the concurrent match induced by t and $k_0 : G_0 \to H_0$ the inclusion morphism (Thm. 1.6.1).

Since $type(H_0) \subseteq CT$, this follows from Lemma 6.5.1, because the existence of a morphism $q : N_c \to H_0$ that violates an arbitrary NAC N_c of p_c would imply for $x \in N_C \setminus n(L_C)$ with $type(x) \notin CT$ also $type(q(x)) = type(x) \notin CT$ which is contradiction to $type(H_0) \subseteq CT$. □

6.6 Compositionality of the Semantic Mapping

In this section, we prove, that the graph to graph transformation $sem_{g2g} : \mathbf{Graphs_{TG_{arch}}} \rightarrow \mathbf{Graphs_{TG_{CSP}}}$ presented in Section 5 combined with the graph to term transformation $sem_{g2t} : \mathbf{Graphs_{TG_{CSP}}} \rightarrow CSP$ presented in Section 4 form a *compositional* semantic mapping.

The semantic mapping with *terminating* and *locally confluent* graph to graph transformation and *context-preserving* graph to term transformation is compositional, if it is *separable* and *constructive*.

6.6.1 Local Confluence

The *local confluence* of $sem_{g2g} : \mathbf{Graphs_{TG_{arch}}} \rightarrow \mathbf{Graphs_{TG_{CSP}}}$ can be proven through a critical pair analysis in AGG [AGG07]. The critical pair analysis shown in Figure 6.8 ran successfully using AGG 1.6.4. It found no essential critical pairs [LEO08], thus *local confluence* is proven.

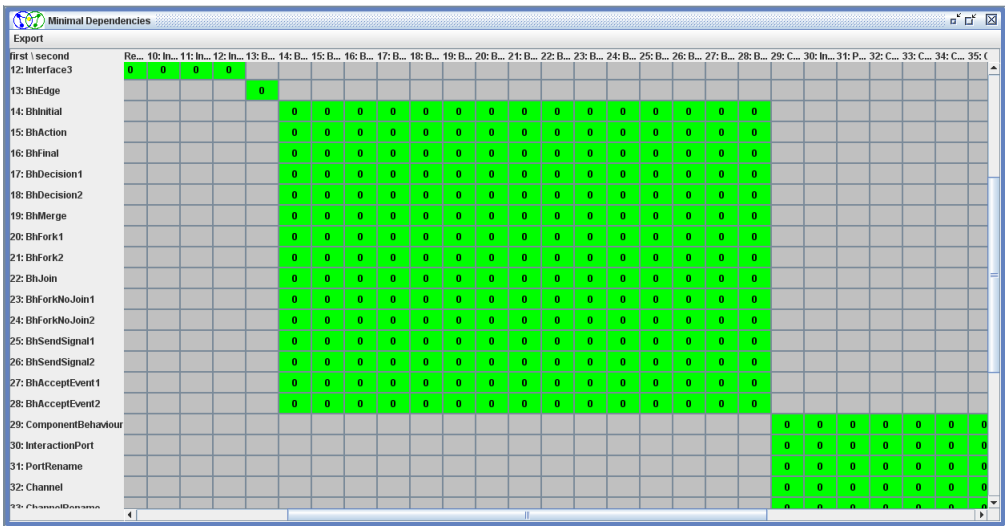


Figure 6.8: Essential Critical Pair Analysis in AGG

6.6.2 Constructiveness

The graph transformation system GTS_{smc} basically reads the architectural model and creates the corresponding set of CSP expressions. None of the rules delete elements from the architectural metamodel.

Observation 6.6.1. *GTS_{smc} with initial graph $G \in \mathbf{Graphs_{TG_{arch}}}$ is nondeleting with respect to G*

Also, the NACs are defined on CSP expressions only.

Observation 6.6.2. *Given* $GTS_{smc} = (TG_{root}, P_{smc})$*, for all* $(p, NAC_p) \in P$*,* $NAC_p \in$ $\mathbf{Graphs_{TG_{CSP}}}$.

Lemma 6.6.1. ***(Constructiveness of** sem**)*** *The typed graph transformation system* $GTS_{smc} = (TG_{arch}, P_{smc})$ *is* constructive.

Proof. The elements of the architectural metamodel are constant types, because of Observation 6.6.1 their instances are constant points in the transformation. Thus, the instances of TG_{CSP} are non-constant. Also, according to Observation 6.6.2 the NACs consist of elements that are instances of TG_{CSP}. Consequently, the NACs are non-constant as well.

These observations correspond to the assumptions of Definition 6.5.4, hence the transformation is *constructive*. □

6.6.3 Context Preservation

Context preservation in this case means that the gluing of graphs through an initial pushout is equivalent to the substitution of terms. The intuitive meaning is shown in Figure 6.9: initial pushout (1) where A is well-formed CSP graph (satisfying the well-formedness constraints) corresponds to the substitution `A = C[T]`. Since the graph representation of CSP is the abstract syntax tree, the gluing of context corresponds to substituting a tree T for a variable represented by a leaf in C.

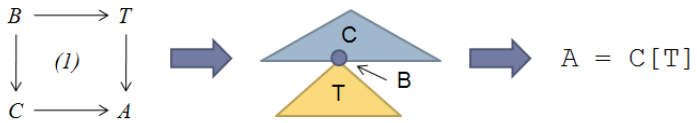
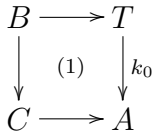


Figure 6.9: Idea of context preservation

As the initial pushout conforms to the CSP metamodel (including its well formedness criteria), all graphs of CSP are trees and their valid gluings will remain trees as well.

Theorem 6.6.1. *The transformation* $sem_{g2t} : \mathbf{Graphs_{TG_{CSP}}} \rightarrow CSP$ *is* context preserving*, ie. for each initial pushout* (1) *with* $B, T, C, A \in \mathbf{Graphs_{TG_{CSP}}}$ *it holds that* $sem_{t2g}(A) \equiv sem_{t2g}(C)[sem_{t2g}(T)]$.



Proof. The only elements in CSP that can have a context (and thus can be substituted) are the process expressions. When set `T` of CSP expressions are merged with set `C`, the following cases may occur:

1. Process assignments of `T` are added to the ones in `C` and vice versa.
2. Empty process declarations in `T` are merged with their definition found in `C` and vice versa.

Gluing various process expressions inside a *PE-tree* is not possible, there are neither stand alone process expressions nor placeholders in process definitions to fill. As it can be seen in the CSP metamodel (Fig. 4.1), the *CspContainer* is not connected directly to the *ProcessExpression*. They are either the definition of a *ProcessAssignment* or the operand of a *BinaryOperator*.

It is important to note, that in the CSP metamodel, there are no association relations, every element is contained within another one. This ensures that there are no unwanted connections.

In the following we will show for each case that (*i*) the pushout object will remain to be a valid CSP graph, and (*ii*) this gluing preserves the context when transformed to the term-based representation.

Case 1 *Merging Process Assignments.* As the *ProcessAssignments* can be connected only directly to the *CspContainer*, the result of creating a union of the *ProcessAssignments* will be will be a well-formed CSP graph.

When the gluing of two CSP graphs consists in creating a union of the *ProcessExpressions*, the result will trivially remain a well-formed CSP graph. *ProcessAssignments* can not be connected to a *Process* by both their *process* and *processIdentifier* edges, since both are containment relations. One element can not be contained in two containers.

The gluing of *ProcessAssignment* nodes to the same *CspContainer* is simply adding more process assignments to the set of present expression if we view it via the term-based representation. Thus the context, whichever process was embedded is preserved.

Case 2 *Process Definitions.* The case of inserting process definitions into the empty declarations follows the same argument as merge of process assignments. The only difference is that in this case, the *ProcessAssignment* and its *processIdentifier* are both part of the boundary graph. The *PE-tree* is glued to a *ProcessAssignment* with no *process* edge. Since a process got defined, the pushout object remains a valid CSP graph. □

These two ways to glue processes seem to be restricting. However, since our transformation creates such processes that contain only one process expression (except choice), this level of granularity is satisfactory.

6.6.4 Termination

The termination criteria introduced in [EEdL+05] sorts the rules of a graph transformation system into layers. Each layer is associated with the creation or deletion of a specific type. Our graph transfomation system, GTS_{smc} builds a CSP abstract syntax tree from the architectural model: the various node types in CSP (e.g *Process* or *Event*) are created by almost every rule. Hence, they cannot be sorted into creation layers (i.e. almost all rules would be in one big layer) and this criteria cannot be applied.

First, termination of transformations consisting of only nondeleting rules (Thm. 6.6.2) is proven. Instead of layers, we define a precedence on nondeleting production rules based on the *produce-enable* sequential dependency. Using this precedence relation, a dependency graph can be built that shows the possible application order of production rules. Nondeleting rules with self-disabling NACs can be applied only once at the same essential match (Lemma 1.5.1). If the dependency graph contains no directed cycles, the GTS is terminating as there is no infinite cycle of rules where new matches are created infinitely.

Then, these results are adapted to GTSs with deleting rules. Under the assumption that the effect of the deleting rules can be isolated into rule groups (Def. 6.6.3) we extend the precedence relation to these self-contained rule groups. Termination follows (Thm. 6.6.3) if the self-contained rule groups are terminating by themselves, and the dependency graph lacks directed cycles.

With the help of these two theorems, the termination or our graph transformation system GTS_{smc} translating *CSML* to CSP is proven.

When building the abstract syntax tree of CSP expressions, the elements closer to the root are created first, then the branches. As observed in Section 5.1.1, the

elements close to the root are the process declarations, and the branches are the *ProcessExpression*-subtrees (*PE-trees* Def. 5.1.1) specifying the behaviour. Hence, a *de-facto* rule application precedence can be observed: rules, that fill a process declaration can be applied only after the process declaration was created (or renaming rules can be applied only when the renamed process was created). This notion of *precedence* can be formalised as a *produce-enable* dependency: production rules p_1, p_2 are *produce-enable* dependent if p_1 produces some nodes that enable the application of p_2. Although there are other types of sequencial dependencies, we concern only *produce-enable* since the graph transformation was assumed to be nondeleting (i.e. we cannot have *delete-forbid* dependency with nondeleting rules).

Definition 6.6.1. ***(Produce-Enable Dependency)*** *Given a graph transformation system $GTS = (TG, P)$.*

Rule p_1, p_2 are in a produce-enable *dependency, notated by $p_1 \overset{pe}{\rightarrow} p_2$ if there exist two direct graph transformations $t_1 : G \overset{p_1,m_1}{\Rightarrow} H_1$, $t_2 : H_1 \overset{p_2,m_2}{\Rightarrow} H_2$ where*

- *p_2 is applicable on H_1*
- *some nodes and edges in the intersection of the comatch $n_1 : R_1 \rightarrow H_1$ and the match m_2 are created items with respect to t_1 and gluing items with respect to t_2, i.e.:*

$$n_1(R_1) \cap m_2(L_2) \subseteq n_1(R_1 \setminus r_1(K_1)) \cap m_2(l_2(K_2))$$

The *produce-enable* relation is a *partial order* on the production rules. This *partial order* relation gives rise to a directed graph where the nodes are the rules and the edges indicate the dependency between them.

Definition 6.6.2. ***(PE Dependency Graph)*** *The* produce-enable dependency graph *of a graph transformation system $GTS = (TG, P)$ is a simple directed graph $G = (P, E, s, t)$, where an edge $e \in E$ with $s(e) = p_1 \in P$ and $t(e) = p_2 \in P$ exists if $p_1 \overset{pe}{\rightarrow} p_2$.*

First, we assume to have a graph transformation system, that has only nondeleting productions with NACs sufficient to be self-disabling (Def. 1.5.11). The proof of termination is the following intuitively. The *produce-enable* dependency graph is the direct translation of the *produce-enable* dependency relation into graph-theoric terms: it has an edge $p \rightarrow r$ for every related $p \overset{pe}{\rightarrow} r$. If there are no directed cycles in the *produce-enable* dependency graph, rules would not produce corresponding matches for each other infinitely, resulting in finite rule application sequence and terminating transformation.

Theorem 6.6.2. ***(Termination of Nondeleting GTS with Acyclic PE Dependency Graph)*** *Given a graph transformation system $GTS = (TG, P)$, such that all rules are nondeleting and have self-disabling NAC (Def. 1.5.11). If the start graph G_0 and the rule set P is finite and the* produce-enable dependency graph *contains no directed cycles, then GTS is terminating.*

Proof. Termination is proven by contradiction, hence we assume the existence of an infinite rule application sequence σ_{inf}.

Since P contains m rules, and σ_{inf} is an infinite application sequence, by the Pidgeonhole principle, all rule applications cannot be distinct. Thus, there is set of rules $\{r_i, ..., r_j\} \in P$ that are applied infinite times, where $0 < i, j \leq m$.

For each direct derivation $G_i \overset{r_i}{\Longrightarrow} G_{i+1}$ with injective matches and injective morphism $d_i : G_i \to G_{i+1}$ (induced from $G_i \overset{r_i}{\Longrightarrow} G_{i+1}$ by nondeletion of r_i), each match $m_{i+1} : L_i \to G_{i+1}$ must have an *essential match* $m_i : L_i \to G_i$ with $d_i \circ m_i = m_{i+1}$. From Lemma 1.5.1 we conclude that we have at most one application of rule r_i on essential match m_i.

Thus the rules in set $\{r_i, ...r_j\}$ create matches for each other, i.e. there is at least one cycle, where $r_i \overset{pe}{\to} r_{i+1} \overset{pe}{\to} ... \overset{pe}{\to} r_j \overset{pe}{\to} r_i$. However, this means that the *produce-enable dependency graph* contains a directed cycle, which contradicts our assumptions. Thus, an infinite rule application cycle cannot exist.

□

Unfortunately Theorem 6.6.2 does not apply directly to GTS_{smc} because it contains some deleting rules. The strategy to deal with the deleting rules is to isolate them.

Rules in GTS_{smc} can be sorted into subsets that build different parts of the CSP graph. This means that such a rule group is self-contained: they work on their own subgraph and do not interfere with other groups. On more technical level it means that deletion is contained within a group of rules; one rule deletes only the product of other rules in the same group.

Definition 6.6.3. ***(Self-Contained Rule Set)*** *Given a typed graph transformation $GTS = (TG, P)$. A subset $P_i \subset P$ is* self-contained *if, when $q \in P$ is sequentially dependent on $p \in P_i$ but* not *produce-enable dependent, then $q \in P_i$.*

If p is nondeleting and there is no $q \in P$ such that q is sequentially dependent on p then it forms its own self-contained rule set, i.e. $P_i = \{p\}$.

Definition 6.6.4. ***(PE Dependency for Rule Sets)*** *Rule set P and R are* produce-enable *dependent, if exists rules $r \in R$ and $p \in P$ such that $r \overset{pe}{\to} p$.*

From a different perspective, this means that deletion only happens within that subgraph that was created by the rules of the self-contained group. The existence of these rule sets in GTS_{smc} were already motivated in Section 5.1.1; the deleting rules are always inserting new elements into *PE-trees*, and thus these rule groups are easily identified.

Definition 6.6.5. ***(Non-Interfering Rule System)*** *Given a graph transformation system $GTS = (TG, R)$. The named subsets of rules $R_1, R_2, ...R_n \subset R$ form a* non-interfering rule system *if $R_1 \cap R_2 \cap ... \cap R_n = \emptyset$ and each R_i is a self-contained rule set.*

Observation 6.6.3. *The self-contained rule sets of P_{smc} shown in Figure 6.10 form a* non-interfering rule system*.*

This observation hold for P_{smc} for the following reasons. The nondeleting rules are in their own self-contained rule set, thus they obviously do not interfere with each other. Each deleting rule is responsible for expanding its corresponding *PE-tree*. Also, it is grouped together with the other rules building that tree. The reason why they do not interfere is the permanence of the source model. All rules match the relevant element in the source model, and build the corresponding behaviour in CSP. Thus, each rule set works on its set of trees that are identified by elements of the same type in the source model. For example the rule set associated with a fork node will build all the syntax trees that are corresponding to fork nodes, other rule sets do not interfere.

Observation 6.6.4. ***(Self-Disabling NACs)*** *All rules $p = (L \xleftarrow{l} K \xrightarrow{r} R) \in P_{smc}$ have a* self-disabling *NAC (Def. 1.5.11), i.e. given $NAC(n)$ there is an injective $n' : N \to R$ such that $n' \circ n = r$.*

In order to prove termination, we update the definition of *produce-enable dependency graph* (Def. 6.6.2) and Theorem 6.6.2 to incorporate self-contained rule sets.

Definition 6.6.6. ***(PE Dependency Graph with Non-Interfering Rule System)*** *The* produce-enable dependency graph *of a graph transformation system $GTS = (TG, P)$ with* non-interfering rule system *$R_1, R_2, ...R_n \subset P$ is a simple directed graph $G = (V_R, E, s, t)$ with $V_R = \{R_1, ...R_n\}$. An edge $e \in E$ with $s(e) = R_1$ and $t(e) = R_2$ exists if $p_1 \xrightarrow{pe} p_2$ with $p_1 \in R_1$ and $p_2 \in R_2$.*

Theorem 6.6.3. ***(Termination of GTSs with With Non-Interfering Rule System)*** *Given a graph transformation system $GTS = (TG, P)$ with* non-interfering

rule system $R = R_1, R_2, ...R_n$ *such that* $R_i \subset P$ *and all rules have a self-disabling NAC (Def. 1.5.11). If the start graph* G_0 *is finite, the* produce-enable dependency graph *is acyclic and the self-contained rule sets* R_i *with deleting rules terminate individually, then* GTS *is terminating.*

Proof. The proof of Theorem 6.6.2 can be applied with the following modifications. Termination is proven by contradiction as well, we assume the existence of an infinite rule application sequence σ_{inf}.

Since P contains m rules, and σ_{inf} is an infinite application sequence, by the Pidgeonhole principle, all rule applications cannot be distinct. Thus, there is collection of rule sets $\{R_i, ..., R_j\}$ that are applied infinite times, where $0 < i, j \leq m$.

The rule sets R_i were individually assumed to be terminating, thus the sets in $\{R_i, ..., R_j\}$ must create matches for each other infinitely. It was shown in Theorem 6.6.2 however, that this implies a directed cycle in the *produce-enable dependency graph* which contradicts the assumtions.

□

Based on the AGG dependency check, the dependency graph of P_{smc} is shown in Figure 6.10. The nodes of the graph are the rule sets $R_1, R_2, ...R_m \subset P_{smc}$ that form a non-interfering rule system.

Corollary 6.6.1. *The graph transformation system* $GTS_{smc} = (P_{smc}, TG_{root})$ *with a finite start graph* G_0 *and injective matches is* terminating.

Proof. The rules of P_{smc} form a *non-interfering rule system* (Observation 6.6.3), all rules have *self-disabling NAC* and the *produce-enable dependency graph* acyclic. Hence, according to Theorem 6.6.2 and 6.6.3, we have to show that the rule groups that contain deleting rules terminate individually.

First, we show the finite application of the {*BhDecision1, BhDecision2*} self-contained rule set with *BhDecision1* shown in Figure 6.11 and *BhDecision2* shown in Figure 6.12.

BhDecision1 is a nondeleting rule creating *PE-tree* A_d. Unless the root of the *Choice*-tree is not deleted, this rule can be applied at most once with the same essential match. *BhDecision2* is a deleting rule, that builds up a binary tree. As there exists an injective $n' : N \to R$ such that $n' \circ n = r$, its NAC is self-disabling. Because of the *non-interfering rule system* (Observarion 6.6.3), and the fact that *BhDecision1* can be applied once with on the same *DecisionNode*; only *BhDecision2* may alter A_d. As observable in Figure 6.12, the rule only replaces the deleted *process*

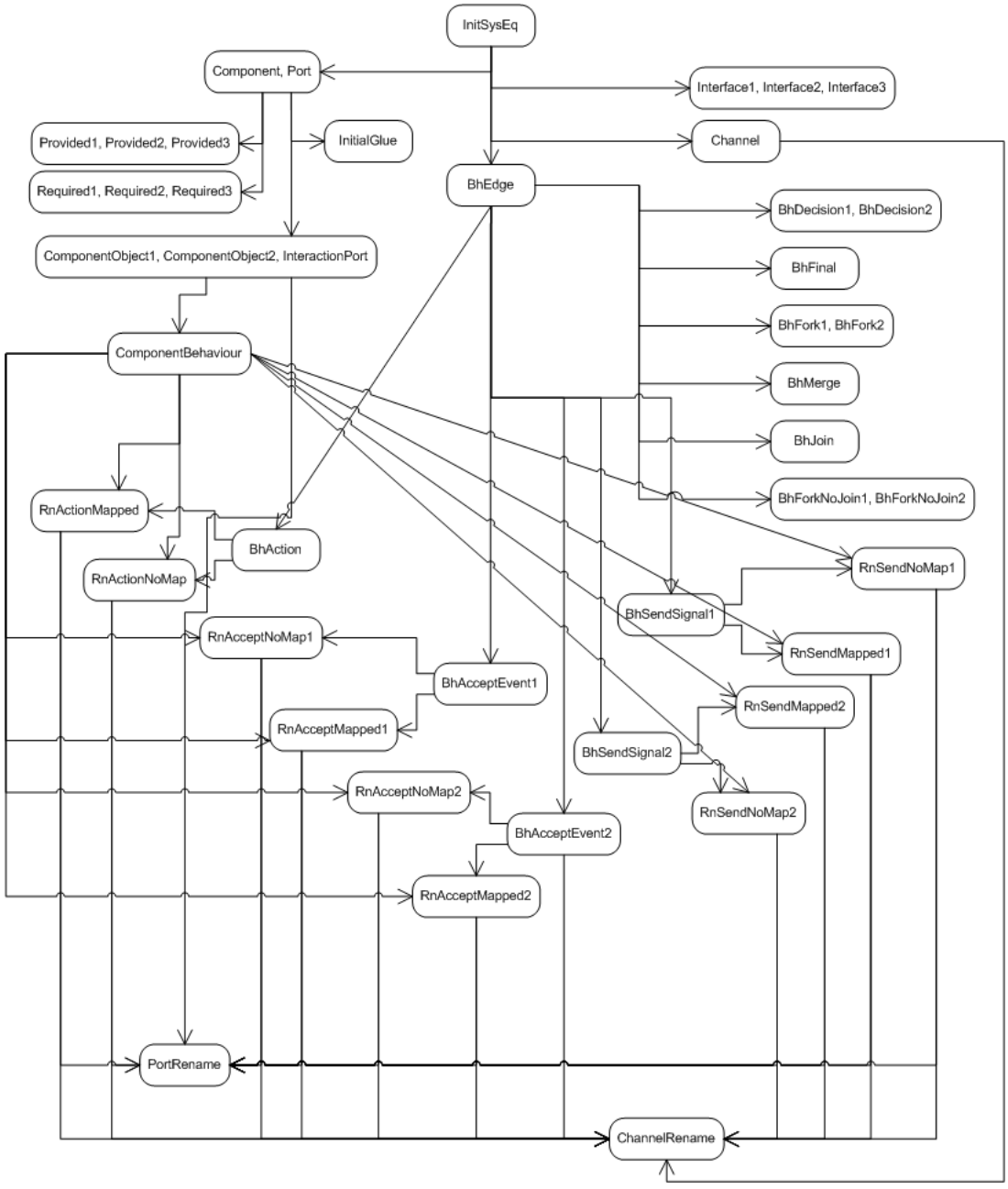


Figure 6.10: Produce-Enable dependency graph of P_{SMC}

edge, it does not delete *Choice* nodes or *Process* leafs from the tree (or anything else). Thus, if one *DecisionNode* is connected to n outgoing edges, this rule may be applied precisely $n-2$ times. Considering that G_0 is finite and there is only finite number of *DecisionNodes*, *BhDecision2* can be applied only finite amount of times.

The finite application of other deleting rule groups follow from the structural similarity to the group containing {*BhDecision1, BhDecision2*}.

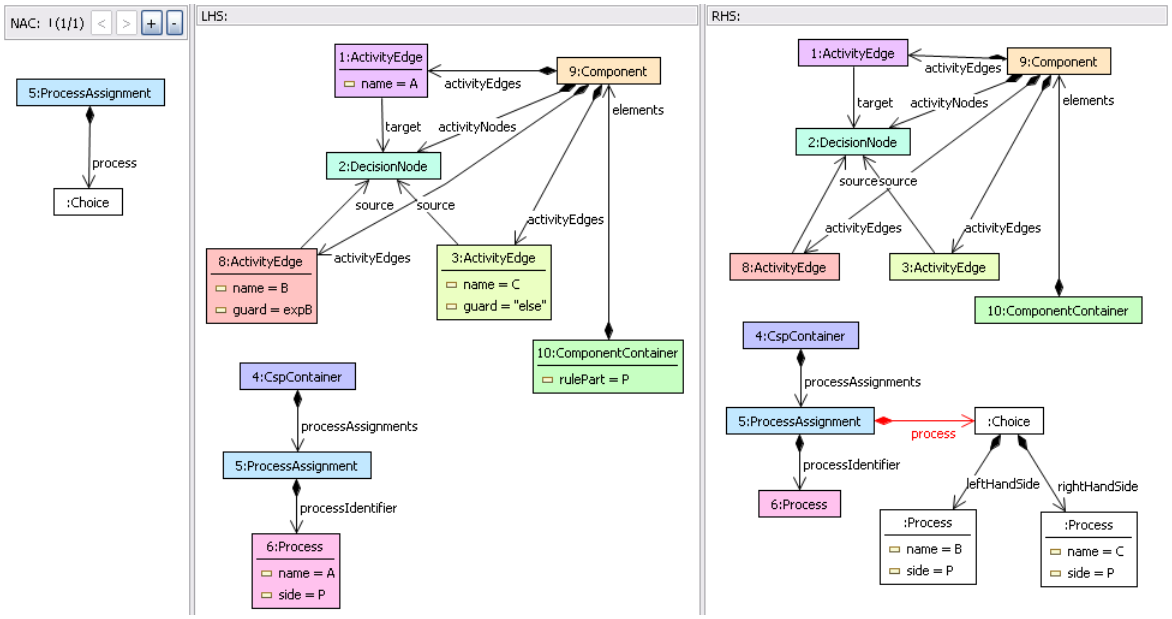


Figure 6.11: The *BhDecision1* rule

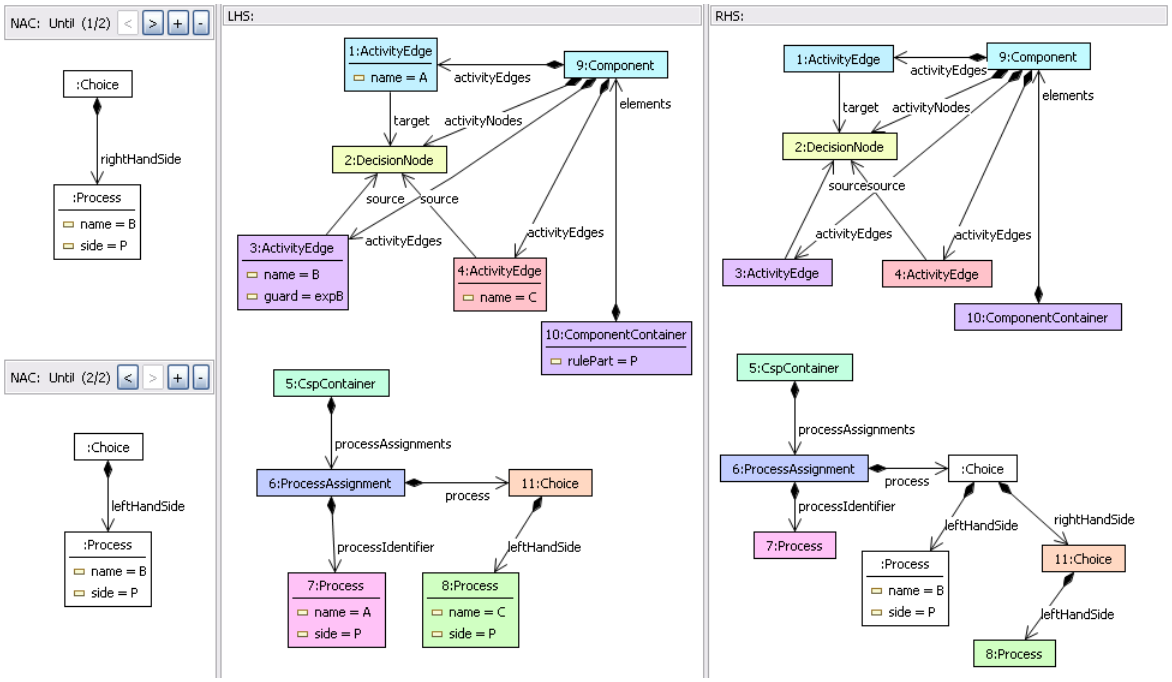


Figure 6.12: The *BhDecision2* rule

Hence, each cooperative rule group is applied finite times, termination follows from Theorem 6.6.3. □

6.6.5 Separability

To prove the *separability* (Def. 6.4.2) of a graph transformation system $GTS = (TG, P)$, a set of constraints $\mathcal{X}_\mathcal{S}$ need to be defined on $\mathbf{Graphs}_{\mathbf{TG_{mdl}}}$. These constraints, in case of our semantic mapping (as presented in Chapter 5), are based on a factorisation $G_0 \xrightarrow{x_g} G_\mathcal{X} \xrightarrow{x_h} H_0$. Dependent on the content of G_0, the constraints specify elements to be included into $G_\mathcal{X}$ from H_0.

Definition 6.6.7. ***(Separability Constraints)*** *Given graphs G_0, H_0 with injective inclusion $m : G_0 \to H_0$. A graph $G_\mathcal{X}$ satisfies the separability constrains $\mathcal{X}_\mathcal{S}$ if the factorisation $G_0 \xrightarrow{x_g} G_\mathcal{X} \xrightarrow{x_h} H_0$ exists with injective inclusions x_g, x_h and*

- *for all $x \in V_{G_0}$ such that $m(x) \in V_{H_0}$ and $type(x) = ActivityNode$: $G_\mathcal{X}$ must contain all $e \in E_{H_0}$ with $type(e) = ActivityEdge$ that has no preimage in G_0 and either $s(e) = m(x)$ or $t(e) = m(x)$.*
- *for all $x \in V_{G_0}$ such that $m(x) \in V_{H_0}$ and $type(x) = Port$: $G_\mathcal{X}$ must contain all $y \in V_{H_0}$ with $type(y) = OwnedInterface$ and $e \in E_{H_0}$ with $type(e) = provided$ or $type(e) = required$ that have no preimage in G_0 and $s(e) = m(x)$ and $t(e) = y$.*
- *for all $x \in V_{G_0}$ such that $m(x) \in V_{H_0}$ and $type(x) = SendSignalAction$ or $type(x) = AcceptEventAction$: $G_\mathcal{X}$ must contain all $y \in V_{H_0}$ with $type(y) = Port$ and $e \in E_{H_0}$ with $type(e) = engaged$ that have no preimage in G_0 and $s(e) = m(x)$ and $t(e) = y$.*
- *for all $x \in V_{G_0}$ such that $m(x) \in V_{H_0}$ and $type(x) = Component$: $G_\mathcal{X}$ must contain $y \in V_{H_0}$ with $type(y) = InitialNode$ and $e \in E_{H_0}$ with $type(e) = activityNodes$ that have no preimage in G_0 and $s(e) = m(x)$ and $t(e) = y$.*

Theorem 6.6.4. ***(Separability of* sem*)*** *The typed graph transformation system $GTS_{smc} = (TG_{arch}, P_{smc})$ with the set of constraints $\mathcal{X}_\mathcal{S}$ is separable.*

Proof. We have to show that for all pushouts (1) with $G \models \mathcal{X}_\mathcal{S}$ and $C \models \mathcal{X}_\mathcal{S}$ it holds that if $H_{GTS_{smc}} \gg$ then either $G_{GTS_{sem}} \gg$ or $C_{GTS_{smc}} \gg$.

$$\begin{array}{ccc} B & \longrightarrow & G \\ \downarrow & (1) & \downarrow \\ C & \longrightarrow & H \end{array}$$

Every transformation in *sem* that operate on the activities part of the metamodel, transform a single *ActivityNode* into the semantic domain. Every node type (which is a child type of *ActivityNode*) has a related production rule or rule group in *sem*. As *ActivityEdges* are transformed to process declarations in a delocated way, they form a frame around the *ActivityNodes*, enabling their transformation (Section 5.3). Thus, boundary graphs usually consist of only *ActivityEdges*. If all incoming and outgoing *ActivityEdges* are included with the relevant node, all rules corresponding to that node are triggered before the merge.

SendSignalActions and *AcceptEventActions* need their relevant *Ports* and *OwnedInterfaces* to identify the communication primitives they are (Fig. 5.13). Similarly *Ports* need their *OwnedInterfaces* to determine the actions they may engage in (Section 5.2). The rules related to these elements (*BhSendSignal1, BhSendSignal2, BhAcceptEvent1, BhAcceptEvent2, Provided1, Provided2, Provided3, Required1, Required2, Required3, Interface1, Interface2, Interface3*) can be triggered, if those elements are all present. Including them triggers the rule before merging.

The behaviour of a *Component* starts with its *InitialNode* (Section 5.3). If the *InitialNode* is not present in the *Component* when merging, the gluing of the behaviour process to the component process triggers.

Thus, as both C and G satisfies constraints $\mathcal{X}_{\mathcal{S}}$, no new structures are created in H that enables a previously disabled rule. If C and G were terminating, then H is terminating as well. □

Chapter 7

Architectural Refactoring Patterns

All the building blocks necessary to verify architectural refactorings at rule-level are now present. In Chapter 6, correctness of the rule-level verification of refactorings and the compositionality of the semantic mapping from Chapter 5 has been proven. In this chapter we apply these theoretical results to the architectural domain from Chapter 3. The general methodology and the outline of this chapter as summarised in Figure 7.1 is the following.

The initial starting point is an already finished refactoring. Both the old and the improved system are present, but the refactoring is not verified for behaviour preservation. Instead of verifying the refactoring as it is, we extract the changed parts to form a refactoring rule. As the system is modelled by typed graphs, the refactoring rule is a typed graph production. As a production rule is usually substantially smaller than the system, its verification saves time.

The design flaws the extracted refactoring rules solve, can be recurring in multiple systems. Generalising enhances them into refactoring patterns. When another system with a similar design flaw is encountered, instead of refactoring it by hand, we apply the refactoring pattern to produce the improved system. Verification is not necessary, as the refactoring pattern was previously shown to be behaviour preserving.

Corresponding to the above methodology, Section 7.1 presents the rule-level verification and rule extraction, Section 7.2 elaborates on enhancing refactoring rules into refactoring patterns and Section 7.3 introduces the necessary tool support.

7.1 Rule Extraction and Verification

To provide a general overview on the extraction of refactoring rules, we present a refactoring example on the *PersistentDatabase* component from Section 3.5 [BHE09c].

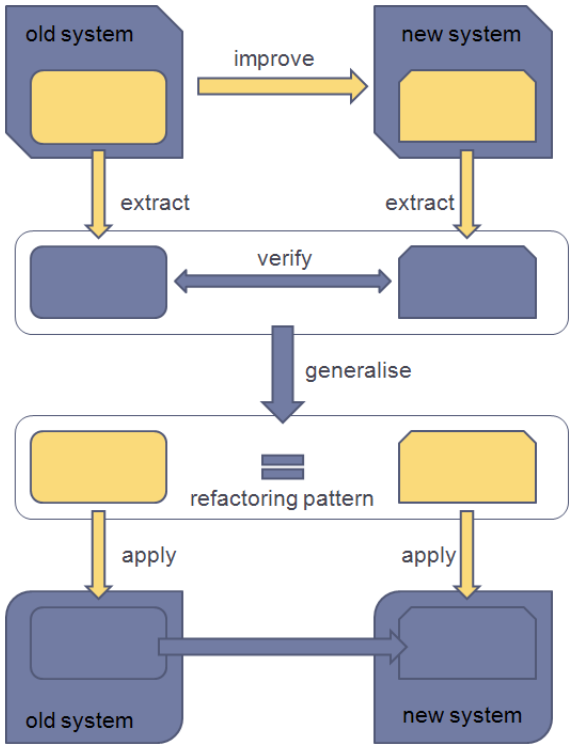


Figure 7.1: Methodology of rule extraction and pattern creation

The *PersistentDatabase* is a naïve design: it synchronises with the route query after every phone number access. This assumes that every distress signal is real, and thus needs an emergency route plan. In several cases the crash may not need medical attention or may even be a false alarm. As the system requires more independence, the synchronisation node is deleted to make the two database engines work completely independently. The refactoring is shown in Figure 7.2.

The extraction process consists of the following two major steps:

1. The minimal graph production rule that produces the refactored system when applied to the original one is extracted.

2. Necessary context is added to the minimal rule to make it semantically complete.

The extraction of the minimal graph transformation rule helps us to identify the changes in the system. The extracted minimal rule is shown in Figure 7.3. Minimality means that it is the smallest rule that produces the refactored system when applied on the original one at the appropriate match.

In most cases, including the present one, the minimal rule is not semantically complete. Moreover, it seems to suffer from syntactic errors (dangling edges). Note,

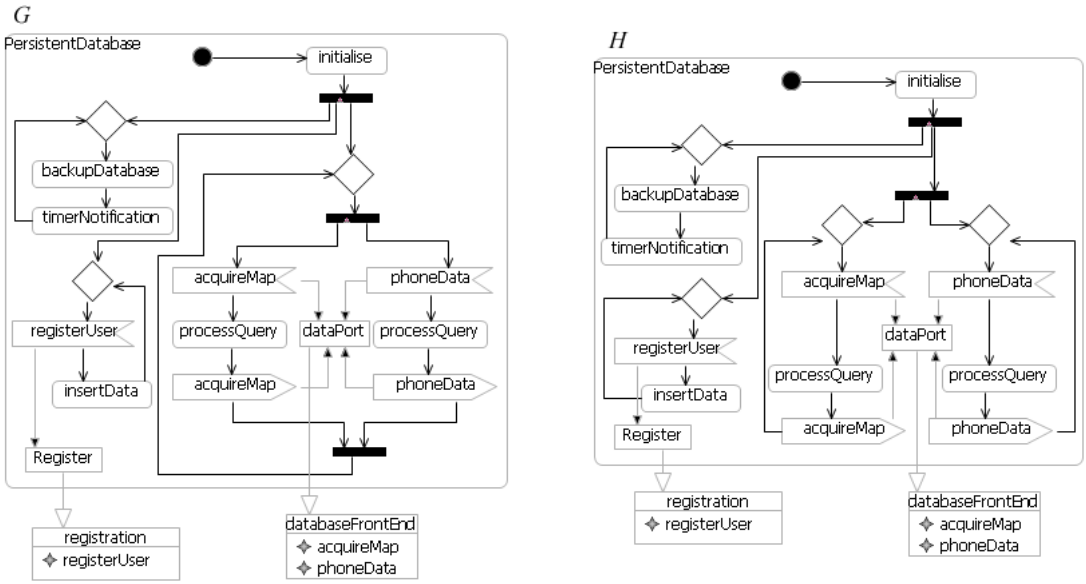


Figure 7.2: Refactoring *PersistentDatabase*

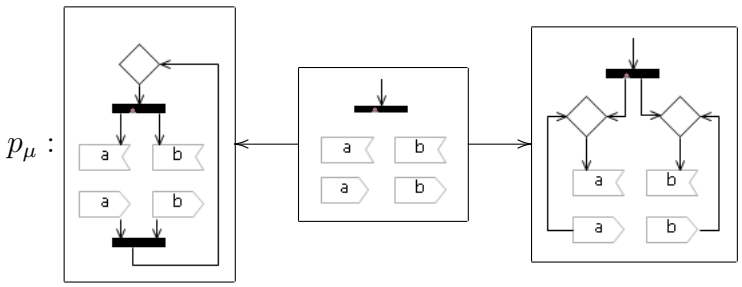


Figure 7.3: Minimal rule of the *PersistentDatabase refactoring*

however, that these diagrams are instances of metamodels where the edges are represented as nodes. To achieve semantical completeness to our minimal rule, we have to address two problems.

1. It is possible to have a valid instance of the metamodel that is not semantically complete. A *SendSignalAction* can mean both function call (if connected to a required interface) or replying the return value (if connected to a provided interface). In the minimal rule the *SendSignalAction* and *AcceptEventAction* are not connected to any port or interface making them ambiguous.

2. Another problem is that the minimal rule is not precisely what the refactoring intended to express. We assume that the fork and the join node form some kind of pair, and as such they are connected to the same line of control flow. In the minimal rule, this is not expressed: the join node can synchronise with arbitrary flows in the system.

To overcome these faults, context needs to be included from the original system. It is important to ensure that the included context is the same on both sides. After the developer selects the necessary context, the resulting rule is shown in Figure 7.4. Verification shows it to be behaviour preserving. Thus, when the rule is used for refactoring, there is no further need for verifying the system. Since we proved the rule to be behaviour preserving, we may generalise it to a refactoring pattern.

The theoretical contributions on refactoring rule creation following the steps outlined are presented in two sections. In Section 7.1.1, we introduce the construction that produces the minimal rule. Section 7.1.2 elaborates on *Step 2*: it establishes the formal framework of context inclusion process. In Section 7.1.3 we present a large example for an extracted architectural refactoring rule.

7.1.1 Extraction of Minimal Rule

The process of minimal rule extraction assumes that the original system G, refactored system H and their relation are given. Minimality, as mentioned, intuitively means the smallest rule that produces H when applied to G. The formal definition is the following:

Definition 7.1.1. ***(Minimality)*** *A graph transformation rule* $p: L \leftarrow K \rightarrow R$ *is minimal over direct graph transformation* $G \leftarrow D \rightarrow H$ *if for each rule* p' : $L' \leftarrow K' \rightarrow R'$ *with injective morphism* $K' \rightarrow D$ *and pushouts* (5) *and* (6)*, there are*

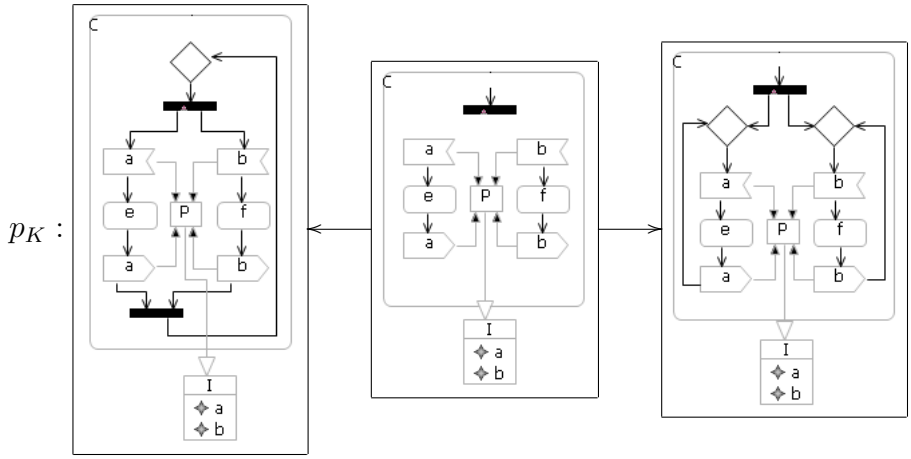


Figure 7.4: *PersistentDatabase* refactoring rule with context

unique $L \to L'$, $K \to K'$ *and* $R \to R'$ *morphisms such that the following diagram commutes and* (7), (8), (5) + (7) *and* (6) + (8) *are pushouts.*

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow & (7) & \downarrow & (8) & \downarrow \\ L' & \longleftarrow & K' & \longrightarrow & R' \\ \downarrow & (5) & \downarrow & (6) & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Now, we present the construction that leads to the minimal rule.

Definition 7.1.2. ***(Minimal Rule Construction)*** *Given direct graph transformation* $G \leftarrow D \to H$ *with initial pushouts* IPO_1 *over* $D \to G$ *and* IPO_2 *over* $D \to H$. *The following construction will define transformation rule* $p_\mu : L \leftarrow K \to R$ *over* $G \leftarrow D \to H$.

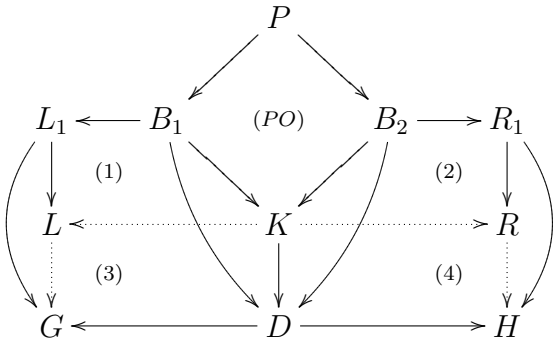
$$\begin{array}{ccccc} L_1 & \longleftarrow & B_1 & & B_2 & \longrightarrow & R_1 \\ \downarrow & (IPO_1) & \searrow & & \swarrow & (IPO_2) & \downarrow \\ G & \longleftarrow & & D & & \longrightarrow & H \end{array}$$

1. *Define* $B_1 \leftarrow P \to B_2$ *as a pullback of* $B_1 \to D \leftarrow B_2$ *and* $B_1 \leftarrow K \to B_2$ *as a pushout of* $B_1 \leftarrow P \to B_2$ *with induced morphism* $K \to D$.

2. *Construct* $L_1 \to L \leftarrow K$ *as a pushout* (1) *of* $L_1 \leftarrow B_1 \to K$ *with induced morphism* $L \to G$. *Similarly,* $R_1 \to R \leftarrow K$ *is a pushout* (2) *of* $R_1 \leftarrow B_2 \to K$ *with induced morphism* $R \to K$.

3. *Since* $IPO_1 = (1) + (3)$ *and* (1) *is a pushout, because of the pushout decomposition property (Def. A.1.1)* (3) *is also a pushout. Similarly* $IPO_2 = (2) + (4)$ *and* (2) *being a pushout implies pushout* (4)*.*

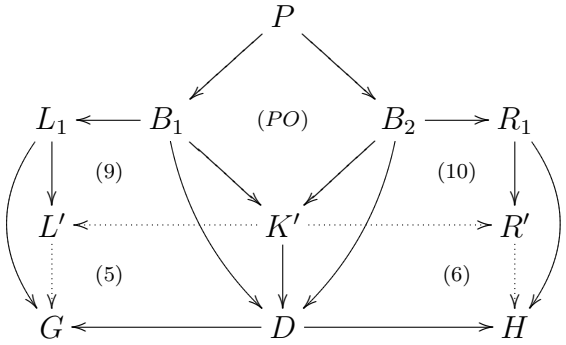
4. *By the constructions of the initial pushouts,* $B_1 \to D$ *and* $B_2 \to D$ *are injective and hence also* $K \to D$*.*



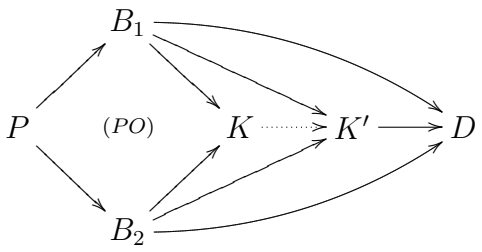
Then $L \leftarrow K \rightarrow R$ *with pushouts* (3) *and* (4) *is* p_μ *over* $G \leftarrow D \rightarrow H$ *with injective morphisms* $L \to G$*,* $K \to D$ *and* $R \to H$*. Moreover injective* $G \leftarrow D \rightarrow H$ *implies injective* $L \leftarrow K \rightarrow R$*.*

Theorem 7.1.1. ***(Minimal Rule Theorem)*** *Assuming a span of injective graph morphisms* $G \leftarrow D \rightarrow H$*, the graph transformation rule* $p_\mu : L \leftarrow K \rightarrow R$ *is minimal according to Definition 7.1.1.*

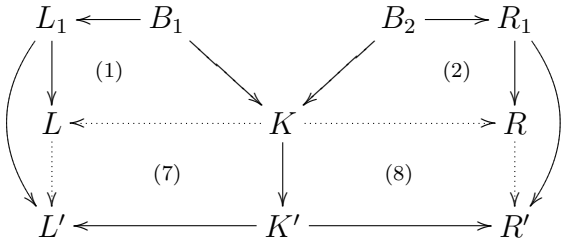
Proof. Given pushouts (5) and (6) over $G \leftarrow D \rightarrow H$ with injective morhpism $K' \to D$, we have by IPO_1 and IPO_2, unique $L_1 \to L'$, $B_1 \to K'$, $B_2 \to K'$ and $R_1 \to R'$.



As the above diagram commutes, (9) and (10) are pushouts. As $P \to B_1 \to D$ and $P \to B_2 \to D$ commutes and $K' \to D$ is an injective morphism, it implies that $P \to B_1 \to K'$ and $P \to B_2 \to K'$ also commutes and hence a unique $K \to K'$ morphism exists and the diagram below commutes.



Now pushouts (1) and (2) implies unique morphisms $L_1 \to L$ and $R_1 \to R$ such that the following diagram commutes and (7) and (8) are pushouts because of the pushout-decomposition of pushouts (9) and (10).



And also $L \to L' \to G$ commutes with $L \to G$ using the pushout properties of L and similarly $R \to R' \to H$ commutes with $R \to H$ using the pushout properties of R.

Uniqueness of $L \to L'$, $K \to K'$ and $R \to R'$ in the minimality diagram follows from the injectivity of $K' \to D$, $L' \to G$ and $R' \to H$. □

7.1.2 Inclusion of Necessary Context

As motivated in Section 7.1, extracting the structural differences between the old and new system is not necessarily enough to create the refactoring rule. To tackle this problem, the software engineer performing the refactoring includes context that makes the rule semantically complete.

Using the example sketched in Figure 7.3, we justify the necessity of including context over the minimal rule. The LHS of the refactoring rule is shown in Figure 7.5.

In our example, as shown in Figure 7.5, there are three 'layers' of included context. We elaborate on them in the following.

1. *Well-formedness.* The rule-graphs have to be valid instances of the metamodel. As the extracted minimal rule is the difference of the two systems, the rule-graphs may not fulfill the well-formedness criteria. The complementing parts that make them valid instances need to be included. In the example, the *Port*, the *CommunicationEvents* engage in, as well as the corresponding *OwnedInterface* and the containing *Component* is included.

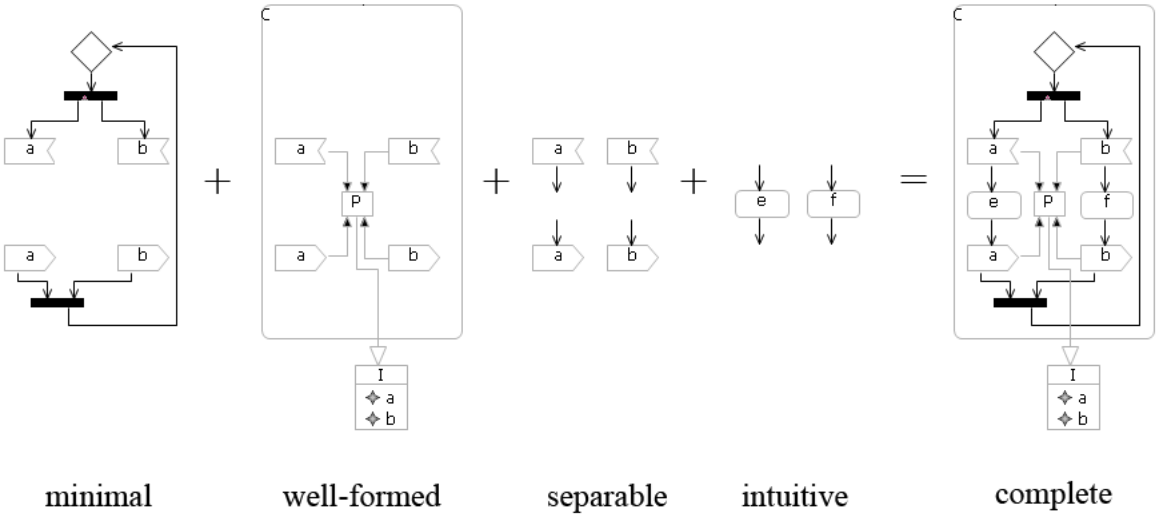


Figure 7.5: Inclusion of context

2. *Separability.* As introduced in Definition 6.4.2, a set of separability constraints is defined on the source model of the semantic mapping. Separability is necessary for the compositionality of the transformation. A non-separable, but well-formed instance of the metamodel can be transformed standalone, but (using the notation introduced in Def. 6.4.2) $sem(H)$ will not be the merge of $sem(G)$ and $sem(C)$. The separability constraints are based on the implementation of the semantic mapping. According to Section 6.6.5, all the incoming and outgoing edges of an *ActivityNode* need to be included from the complete system. Thus, the outgoing edges of the *AcceptEventActions* and incoming edges of the *SendSignalActions* are included.

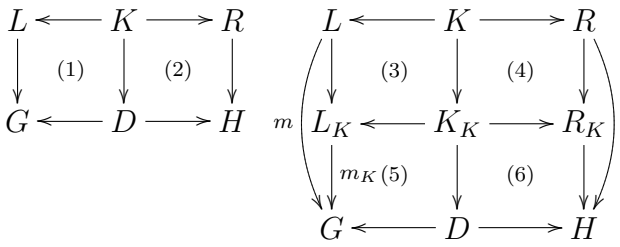
3. *Intuitive Requirements.* As mentioned in Section 7.1, we assume that the *ForkNodes* and the *JoinNodes* form some kind of pair in our example refactoring: they are connected to the same line of control flow. Thus, we include the elements that connect them. The inclusion of these elements have been intuitively decided based on the intention of the refactoring developer.

The intuitively chosen parts of the included context are different in each refactoring case: they cannot be categorised by any kind of formal constraints or rules. Thus formalising a notion of semantical completeness for context inclusions is rather unlikely.

The following process describes the theoretical solution of context inclusion.

Definition 7.1.3. ***(context inclusion process)*** *Given minimal rule* $p_\mu : L \leftarrow$

$K \to R$ over $G \leftarrow D \to H$ with pushouts (1) *and* (2)*.*



The context inclusion is defined by a suitable factorisation $K \to K_K \to D$ of $K \to D$ with injective $K_K \to D$, where L_K and R_K are defined by pushouts (3) *and* (4)*. By pushout decomposition this leads to pushouts* (5) *and* (6) *and $p_K : L_K \leftarrow K_K \to R_K$, where pushout* (1) = (3) + (5) *and pushout* (2) = (4) + (6)*.*

Hence we have $G \Rightarrow H$ via (p_K, m_K) by pushouts (5) and (6) with span $G \leftarrow D \to H$. Moreover the injectivity of $p_\mu : (L \leftarrow K \to R)$ implies that $p_K : (L_K \leftarrow K_K \to R_K)$ is injective as well.

The construction of the rule with additional context based on the factorisation $K \to K_K \to D$ of $K \to D$ using our database example is shown in Figure 7.6.

7.1.3 Architectural Refactoring

As an example intended to be as complicated as real life refactorings, we present a refactoring based on the *AccidentManager* component. This example illustrates the potential complexity of the problem at hand, with changes in all three diagram aspects to be handled.

Extracted Refactoring Rule

With the current *AccidentManager* (Fig. 3.9), scalability issues may arise. Assuming that 70% of the incoming alerts are not real emergencies, the analysis of 'false alerts' consumes considerable resources. The *AccidentManager* may thus turn out to be a bottleneck in the system. To address this scalability problem we extract the initial handling of alerts from the *AccidentManager* into an *AlertListener* component. The refactored system is shown in Figures 7.7 and 7.8 with its component and composite structure diagram respectively. The *AlertListener* receives alerts from cars, forwards them to the *AccidentManager* for processing while querying the database for the phone number and invoking the telephone service, which sends the results of its calls directly to the *AccidentManager*.

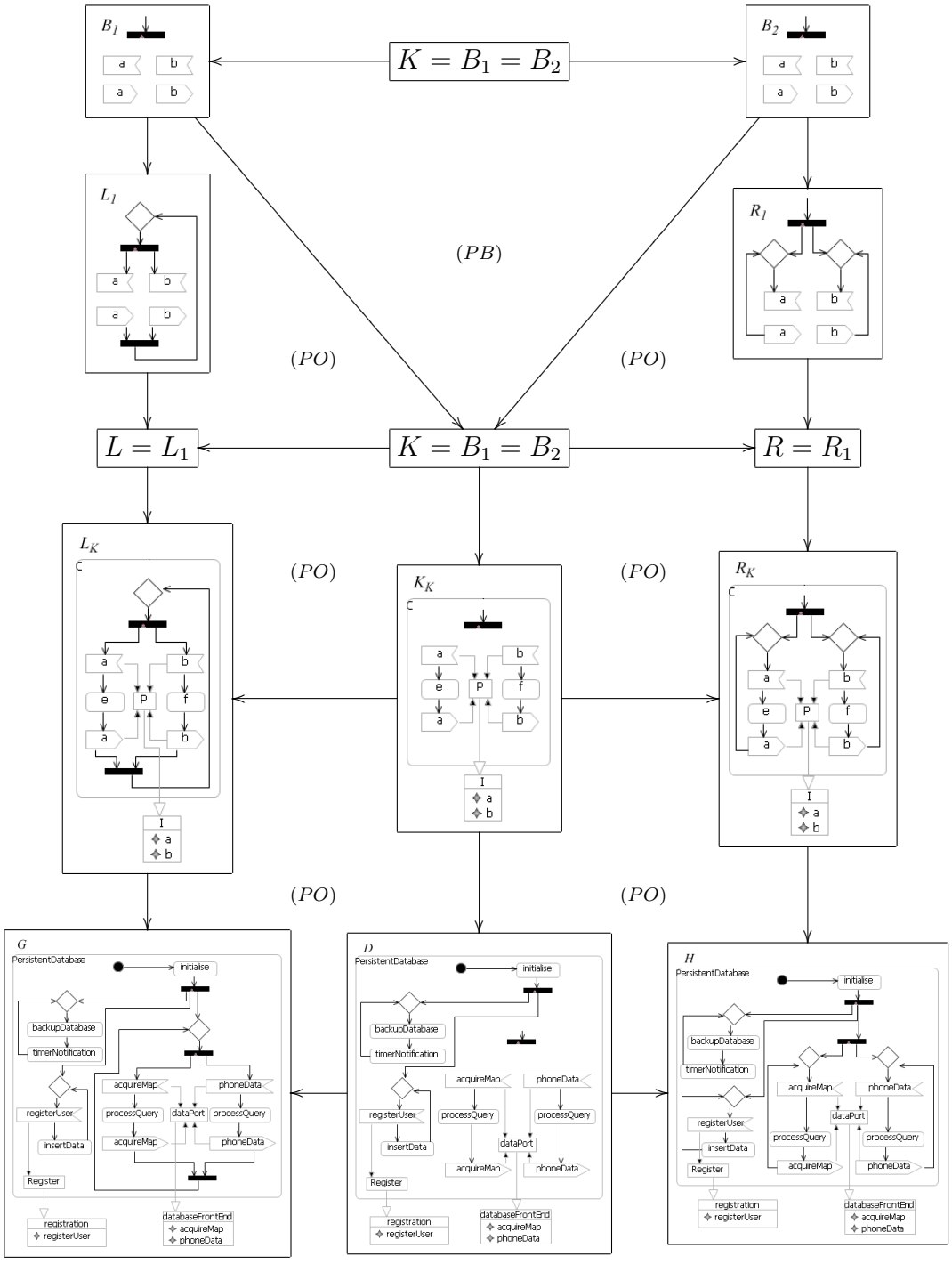


Figure 7.6: The construction of the minimal and extracted rule

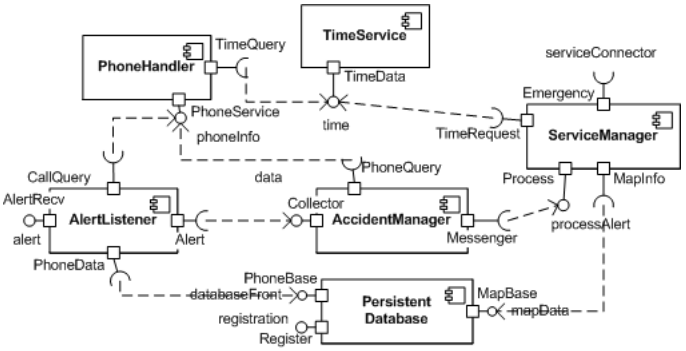


Figure 7.7: Architectural model of the refactored accident server

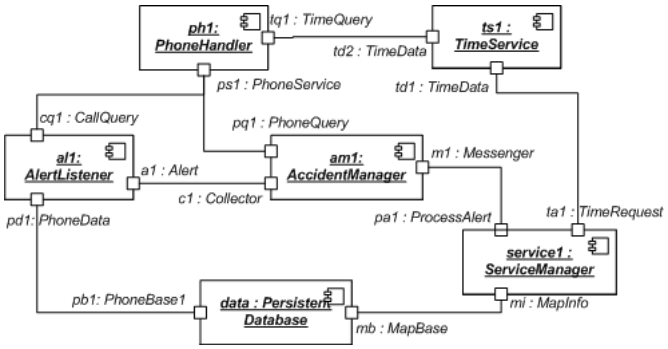


Figure 7.8: Configuration after the refactoring

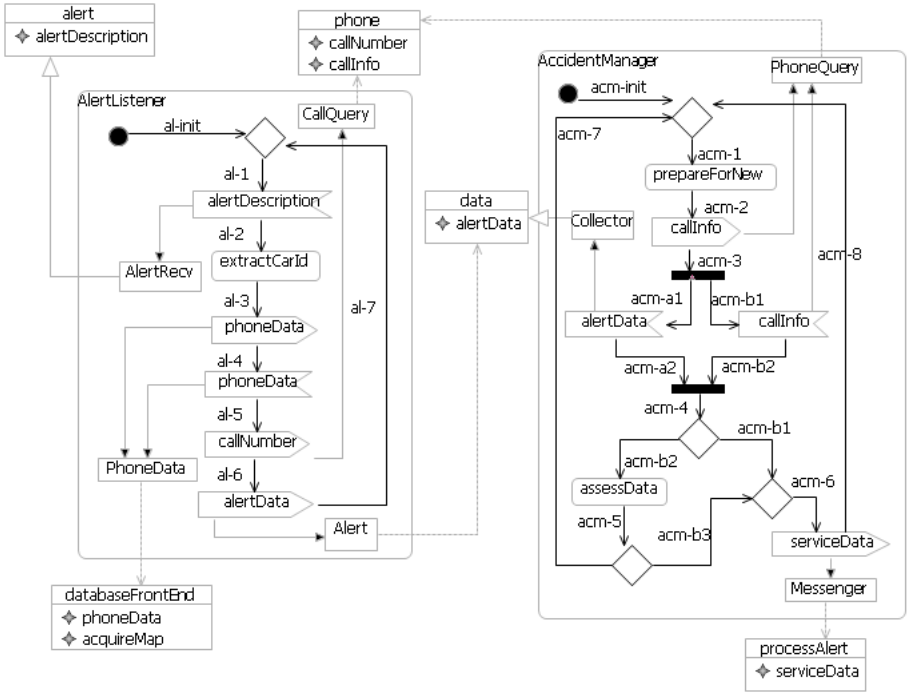


Figure 7.9: Component split refactoring

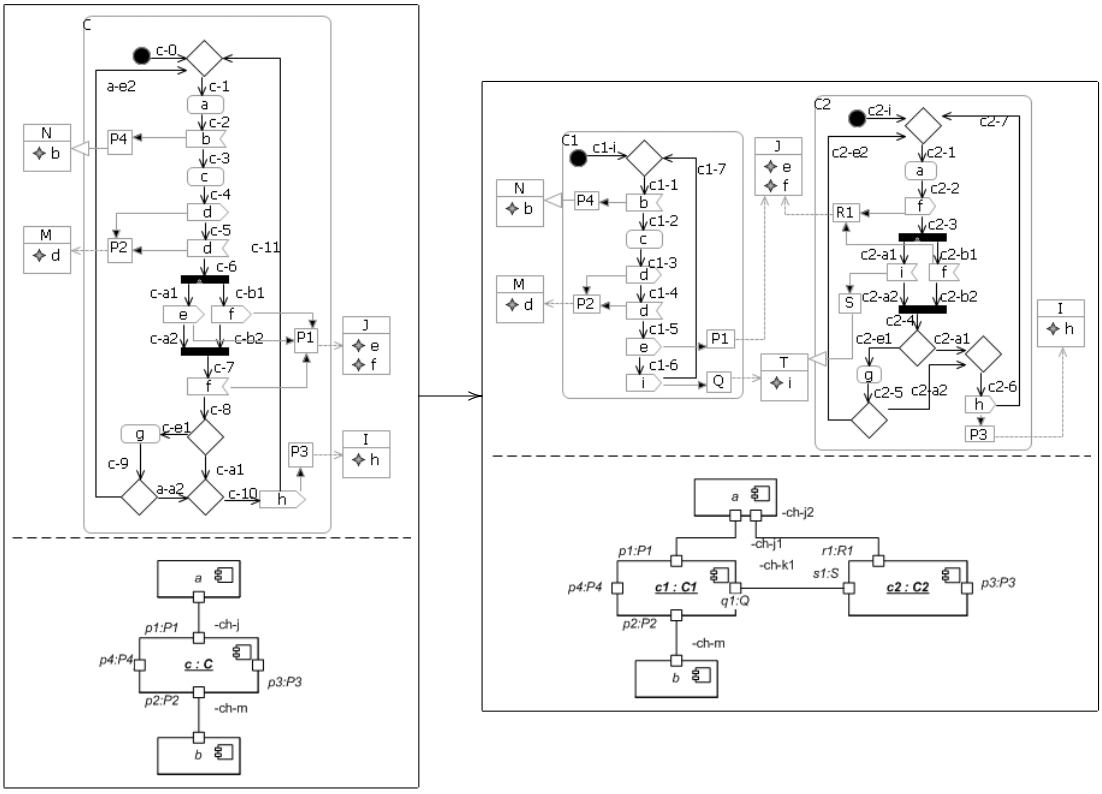


Figure 7.10: Refactoring rule of component split

The behaviour of the new *AlertListener* and the updated *AccidentManager* component is given in Figure 7.9 (the original behaviour of the *AlertListener* is in Fig. 3.9).

Rather than comparing the semantics of the entire accident server model before and after the change, we focus on the affected parts and their immediate contexts. In the present example, the refactoring rule is shown in Fig. 7.10. All three aspects of the system are present in the rule, and the necessary contexts are already included.

The rule is applied by selecting an occurrence isomorphic to the left-hand side of the rule at both type and instance levels in the source model. Thus, component C is matched by *AccidentManager* from Fig. 3.3, interface J corresponds to *phone*, N to *processAlert*, and M to *phoneData*. At instance level, a similar correspondence is established.

The System Equation

To verify the behaviour preservation property of the refactoring rule, its semantical representation is created using the semantic mapping. However, the semantical representation is a large set of CSP expressions both for the LHS and RHS. To perform the verification, a single process is necessary that is essentially an entry point to the

behaviour of the whole system. This process is called the system equation. The system equation is created in three major steps:

1. All component instances and connectors are placed in parallel composition. As components are modular units with encapsulated behaviour, their independent nature is captured this way. The communication channels synchronise the event passing between the components. The system equation is the following in the present case:

```
((((c1.C1_def || ch-k1.T) || c2.C2_def) || ch-j1.J)
      || ch-j2.J) || ch-m.M
```

2. The unmapped events are hidden. They are the events either created or deleted by the production rule. In our case, the unmapped events are the ones associated with the newly introduced I interface. Thus, the system equation is altered in the following way:

```
(((((c1.C1_def || ch-k1.T) || c2.C2_def) || ch-j1.J) ||
      ch-j2.J) || ch-m.M ) \ {| q1.i_send, s1.i_recv |}
```

3. The synchronisation points are included. As mentioned in Section 4.1, the target platform FDR2 is different from the 'official' CSP. The synchronisation events are not included implicitly in the parallel relation; they need to be stated explicitly. Thus, as a final step, we include them into the equation.

```
System_RHS = (((((c1.C1_def [| {q1.i_send,s1.i_recv} |]
 ch-k1.T) [| {q1.i_send,s1.i_recv} |] c2.C2_def)
 [| {p1.e_send} |] ch-j1.J) [| {r2.f_send, r1.f_recv} |]
 ch-j2.J) [| {p2.d_send, p2.d_recv} |] ch-m.M )
 \ {| q1.i_send, s1.i_recv |}
```

The assertion of $System_{LHS} \sqsubseteq_T System_{RHS}$ is successful in FDR2, and indicates trace refinement.

7.2 Refactoring Patterns

The definitions, differences and commonalities of design- and refactoring patterns were discussed in Section 1.3. Refactoring patterns have a precise and well-documented

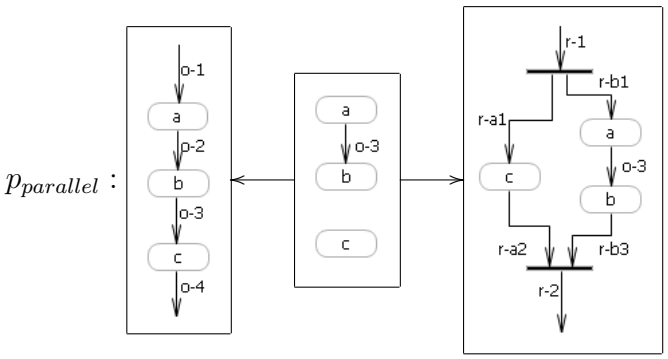


Figure 7.11: Refactoring pattern: parallelization

definition and solution proposal, but their preservation of behaviour was not treated formally (Sec. 2.3). Refactoring patterns require a proper mathematical treatment and proof of behaviour preservation.

Our approach fulfills this requirement. A refactoring pattern is technically a graph transformation rule $p : L \leftarrow K \rightarrow R$ with formal behaviour represented as traces (or failures/divergences) in CSP. The behaviour was verified to be preserved, i.e. $sem(L) \sqsubseteq_T sem(R)$. This graph transformation rule can be applied to a system G. Then, the direct graph transformation $t : G \overset{p,m}{\Rightarrow} H$ produces the refactored system H. As the rule is behaviour preserving and the semantic mapping is compositional (Definition 6.1.2), according to Theorem 6.3.1 the transformation is behaviour preserving as well, i.e. $sem(G) \sqsubseteq_T sem(H)$.

Although it is possible to create a refactoring pattern without a related system; a refactoring pattern solves a particular problem and as such, arises when an actual refactoring is performed. Thus, the creation of a refactoring pattern starts as described in Section 7.1: a relevant refactoring rule is extracted and verified. However, a refactoring pattern is more general than a simple extracted rule as demonstrated in Section 7.2.2.

This section consists of two parts. In Section 7.2.1 an example refactoring pattern is presented and applied. Section 7.2.2 details the differences between refactoring patterns and extracted refactoring rules.

7.2.1 Example Refactoring Pattern

Parallelization of sequential activities is a common task when performance is critical. As introduced in [BH07], a refactoring pattern can be created to parallelise previously sequential tasks as shown in Figure 7.11.

It can be easily proven that the pattern shown in Figure 7.11 is behaviour pre-

$sem(LHS(p_{parallel}))$	$sem(RHS(p_{parallel}))$

`o-1 = a -> o-2`
`o-2 = b -> o-3`
`o-3 = c -> o-4`
	`r-1 = (r-a1 || r-b1); r-2`
	`r-a1 = c -> r-a2`
	`r-a2 = SKIP`
	`r-b1 = b -> o-3`
	`o-3 = c -> r-b3`
	`r-2 = SKIP`

$traces$(`o-1`)	$traces$(`r-1`)

`a -> b -> c`	`a -> b -> c`
	`a -> c -> b`
	`c -> a -> b`

Table 7.1: Verification details of $p_{parallel}$

serving. Table 7.1 details the results of the semantic mapping; the resulting CSP expressions as well as their traces. After studying the traces in Table 7.1 it can be concluded that $traces$(`o-1`) $\subseteq$ $traces$(`r-1`). Thus, it is indeed a refactoring.

We apply this pattern to the *PhoneHandler* component presented in Section 3.5. In the original system (Fig. 3.9), the sending and receiving of *timeRequest* and *collectConnectionData* are executed sequentially. However, the data collection can be run in parallel with the acquisition of the timestamp. Thus, we apply the refactoring pattern with *a* as *timeRequest send*, *b* as *timeRequest receive* and *c* as *collectConnectionData*. The resulting system is shown in Figure 7.12.

As the rule was shown to be behaviour preserving, and the semantic map is compositional, it follows from Theorem 6.3.1 that the new system preserves the behaviour of the old system, and no further verification is needed.

7.2.2 Difference from Extracted Rule

Refactoring patterns are usually extracted refactoring rules. However, there is a subtle but important difference between refactoring patterns and extracted refactoring rules that is more than philosophical: extracted refactoring rules usually solve a particular problem, but refactoring patterns are as general as possible. To illuminate the point, assume that the refactoring presented in Section 7.2.1 was performed by the developer without the help of the presented refactoring pattern. Let us generalise this refactoring rule.

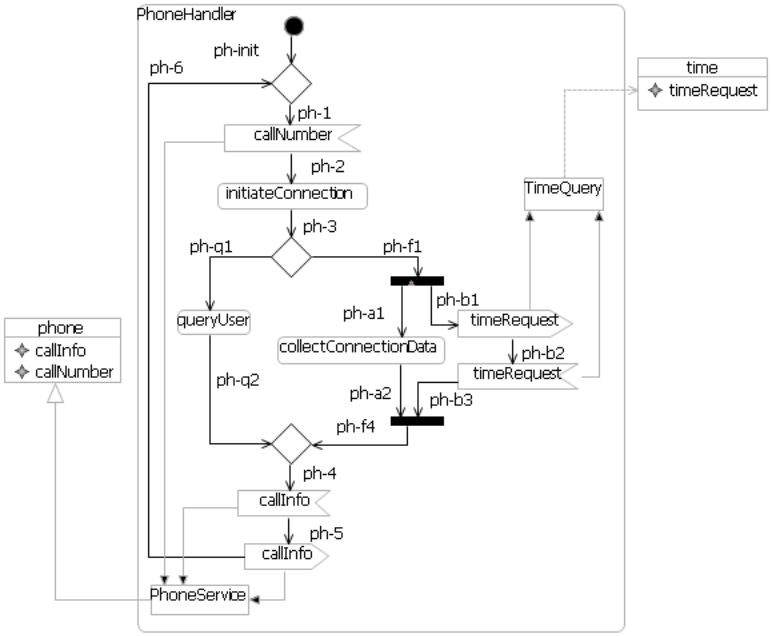


Figure 7.12: Behaviour change of *PhoneHandler*

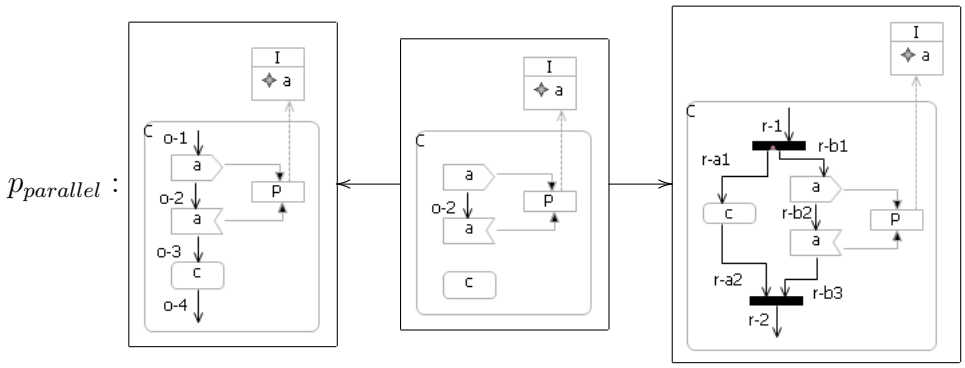


Figure 7.13: Parallelization rule if extracted

The types of a and b in the refactoring pattern shown in the Figure 7.11 and in *PhoneHandler* component in Figure 3.9 do not match. In *PhoneHandler*, a is a *SendSignalAction*, and b is an *AcceptEventAction*. In the other rule extraction case in Section 7.1.2 both the relevant ports and owned interfaces were extracted from the system as a necessary context to make the extracted rule semantically complete in the presence of communication events. Thus, the proper extracted rule would look like the one depicted in Figure 7.13

However, our aim is generality. In the refactoring pattern shown in Figure 7.11 the event types are higher in the inheritance hierarchy: they are *EventNodes*, a common parent class for both *Actions* and *CommunicationEvents*. When applying the pattern as a graph transformation rule, a and b are gluing points, thus the port engagement

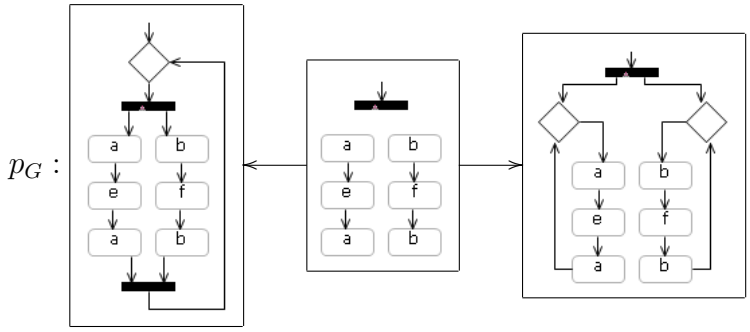


Figure 7.14: Generic *EventNodes* in the *PersistentDatabase* refactoring

connections remain intact in the context system. We can, and do exchange the communication events to *EventNodes* in the extracted rule. Figure 7.14 shows another example: the refactoring pattern derived by the same means from the extracted rule in Figure 7.4.

These examples illustrate well how the refactoring patterns differ from extracted rules.

7.3 Tool Support

This section discusses the tool support that enables the developers refactor system architecture, extract refactoring rules and verify them [BHE09c]. The chain of tools that are used for rule extraction and verification is illustrated in Figure 7.15

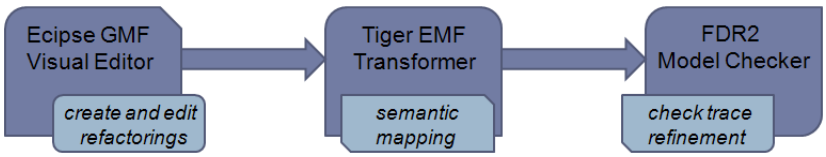


Figure 7.15: Block diagram of the tool chain

7.3.1 Visual Editor

A visual editor is necessary to perform architectural refactorings at the model level. The editor has been implemented using the Eclipse Graphical Modelling Framework (GMF) [GMF07].

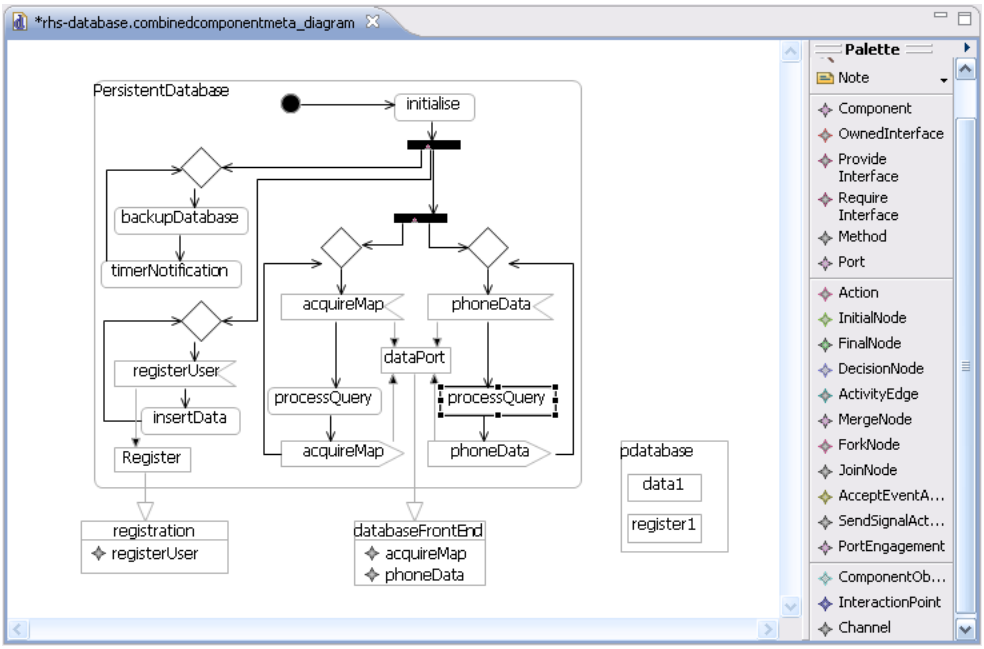


Figure 7.16: Basic functionality of the Visual Editor

Editing Architecture

As mentioned in Section 3.5, the components and their instances are situated in the same diagram, which is called a *combined structure diagram*. Such a diagram is being edited in Figure 7.16. The round rectangle with label *PersistentDatabase* is a type-level component, with *registration* and *databaseFrontEnd* interfaces. Interface implementation is denoted by the usual inheritance and dependance connections: both interfaces are provided by the relevant ports. The behaviour is encapsulated within the component. The other rectangle with label *pdatabase* is a component instance. It contains two *interaction points* (i.e. port instances): *data1* and *register1*.

The metamodel of the combined structure diagram (Fig. 3.1) is represented as an Eclipse Modeling Framework (EMF) model, which is essentially an attributed typed graph as mentioned.

To help rule creation, every object in the diagram has an integer *match* value (0 by default) that expresses matches between the LHS and RHS. Elements with the same *match* are matched. This *match* property is the same as the *match* value discussed in Section 5.5.

Rule Extraction and Context Inclusion

It is desirable to have an automated method for minimal rule extraction. Although there are algorithms solving similar problems [Var06], and formal theory was presented in Section 7.1.1, the conception and implementation of the extraction algorithm is future work.

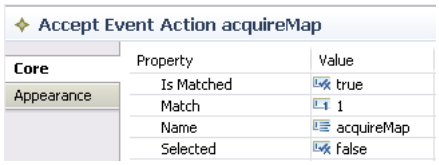


Figure 7.17: *Selected* attribute

Aside from editing, the visual editor enables the definition of the refactoring rule. The attribute *Selected* is used for this purpose: it defines if the particular element is in the rule, regardless of its match status. In Figure 7.17 the accept event action *acquireMap* is not included in the rule yet.

This method of selection is used by the developer to perform the tasks of minimal rule extraction as well as context inclusion.

7.3.2 Semantic Mapping

The transformation that maps the CSML to CSP is implemented using the Tiger EMF Transformer tool [Tig07]. The implementation was detailed in Chapter 5. The rules were designed using the EMT Visual Editor.

The behavioural part of transformation was used for benchmarking the EMF Transformer in [VAB+08] with the case study provided. An average transformation run was around 0.2734 seconds after the first run on an Intel Core Duo T2400 1.83 GHz computer with 1 GB RAM. The times were tested on Windows XP, Vista and Ubuntu Linux, but no significant difference was detected. The full transformation was tested as well, but no thorough benchmarking was done. As an average, the transformation successfully terminated after 2-3 seconds (which is one magnitude slower than the behaviour only version).

7.3.3 Formal Verification

After the mapping to CSP, the expressions are checked for trace refinement with FDR2, a refinement checker for establishing properties of models expressed in CSP

[FSEL05]. All the refactoring cases presented in Sections 7.1.2, 7.1.3 and 7.2.2 among others have been verified for behaviour preservation.

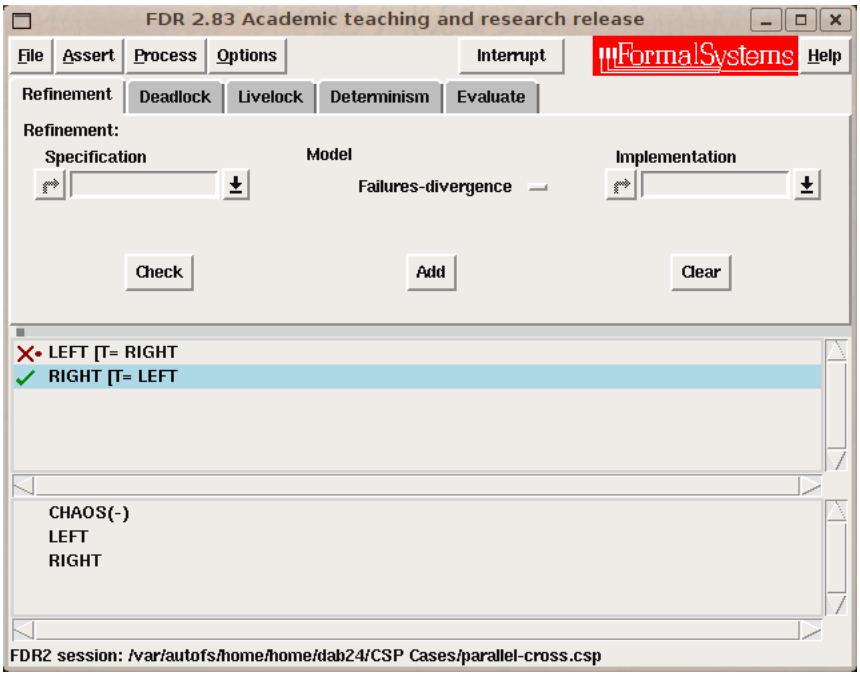


Figure 7.18: FDR2 Verifying the parallel pattern from Section 7.2.2

The verification process is simple: the CSP file is loaded in FDR2 as shown in Figure 7.18, then the trace refinement check is run on the desired expression. Unfortunately, time is a problematic factor in the verification process; the exponential state space explosion is apparent. The smaller verifications, like the ones in Section 7.1.2 and 7.2.2 finished in 5 minutes. The larger one from Section 7.2.2 finished in at least 20-30 minutes. We tried to compare the verification time of the extracted rules versus the whole system. Hopefully the fact that the verification of the whole system crashed FDR2 after 9 hours at all three attempts gives justice to the effectiveness of our approach.

The theoretical problems of not using well-structured activity diagrams have been shown in Section 5.3.5. Apparently, the verification of well-structured diagram based CSP expressions saves time. In complicated cases, where not just behaviour, but also components are involved, the verification time was nearly halved.

Conclusions

The results presented in this work are now reviewed. First, the contributions are summarised, then our experiences and the limitations of the presented work is discussed. The chapter concludes with a list of future challenges.

Summary

The contributions presented can be grouped into three main categories: theoretical, applications of theory, and implementation.

Starting with a concise summary, the main result is a comprehensive methodology that enables the verification of software model refactorings at rule-level. The results presented span two levels of abstraction.

The software models were represented as typed graphs and refactorings as direct graph transformations. Formal behaviour was expressed in a denotational style and mapped onto the software models via a typed graph transformation system. The correctness of verifying refactorings at rule-level assuming a compositionality condition on this semantic mapping was established and proven. The notion of compositionality was defined and guaranteed on this semantic mapping by a structural condition on the form of the production rules. Both basic production rules and rules with negative application conditions were considered.

The theoretical results were applied to two main scenarios. The first is to extract and verify refactoring rules instead of the whole system thus saving resources. The second is the concept of refactoring patterns, which are generalised and verified refactoring rules applicable to a system without the need for further verification. To support the above approaches, a theoretical solution was created that establishes a method to extract the minimal rule and justifies the need of further context inclusion.

As an extensive case-study, software architectural models were chosen for domain of refactoring. The *combined structure modelling language* (CSML) was developed as an architectural description language based on the UML component, composite

structure and activity diagrams. Its behaviour was formalised using CSP with trace semantics. Both the architectural and semantic domain were represented as instances of type graphs.

In order to show the validity of the theoretical contributions, a complete tool chain was assembled. First, the graph representation of both the CSML and CSP was implemented using the EMF platform. A graphical editor for CSML was implemented in GEF enabling the developer to create architectural models and perform refactorings on them. The semantic mapping from CSML to CSP was implemented using the Tiger EMF Transformer and also proven to be compositional. And finally the generated CSP expressions were verified using the FDR2 tool.

Evaluation

As mentioned in Section 6.6, the compositionality of a semantic map holds only under certain requirements. To prove confluency with AGG, we used the complete critical pair analysis first. The main problem with this method is the computational complexity that makes the verification impossible for larger models. However, the latest notion of *essential critical pairs* [LEO08] allowed us to re-run the critical pair analysis with success. Although constructiveness is a simple criteria on rules, the complexity of proving separability and termination varies wildly on the transformation design.

The context inclusion to the extracted minimal refactoring rule needs to be reflected on as well. As mentioned in Section 7.1, the selection of included context is a task of the developer. It is indeed true that the general rules for determining this context are completely dependent on the domain model. There were attempts to define a least bound context for our architecture model. A reasonable candidate for the owned behaviour was the single entry single exit (SESE) region [JPP94] as the smallest bounding context around the changed parts: in nearly every test case, this bounding region produced a very small included context. Unfortunately we were not able to define the notion of the *smallest necessary contexts*.

Future Work

Future work may encompass several orthogonal directions. The most important ones are the following:

- Although the semantic mapping from CSML to CSP is compositional, adapting it to new domains asks for re-evaluation. As the complexity of proving termination and separability is based on the semantic mapping implementation, it could be advantageous to move to a slightly different graph transformation paradigm. The use of strict control flow, as implemented in VMTS [LLC06], would be helpful. Proving termination of a transformation regulated by a control flow is a much easier task. However, it would not only need a completely new implementation of the transformation, but also a fundamental assessment of the validity of the theoretical contributions in the new paradigm.

- Another interesting aspect would be the addition of time properties to the whole system. Instead of CSP, stochastic process algebras like PEPA [Hil96, TG07] could be used. Although, the architectural model would need adaptation to incorporate stochastic properties.

- Obviously, other possible application domains apart from architectural refactorings would pose interesting research topics. For instance in the domain of business processes, formally verified refactoring pattern creation was already researched [KGK+08].

- The method of *cloning* and *expanding* on graph transformation rules introduced by [HJvE06] is another interesting direction. The idea of rule instantiation would be perfect to generalise refactorings like the one shown in Figure 7.4. There, we assume that there can be only one action between the *AcceptEventAction* and *SendSignalAction*. With *cloning* and *expanding* it would be possible to create a refactoring pattern, where the number of intermediate actions is not fixed.

- Our concept of semantics was based on process algebras. Although our theoretical contributions are generic, it would be important to investigate how does it work with other types of semantical domains, like temporal logic expressions.

- And finally the area of context inclusion into minimal rules definitely needs future work. It would be important to create an algorithmic way of producing the necessary context in our architecture domain. Also, the term *'necessary context'* needs precise definition. The largest context is ironically the whole model, thus it would be advantageous to extract something smaller. Hence, definitions are also required for the smallest necessary context if it can be defined concisely.

Refactoring software, especially models of software is a relatively new discipline. The results of this thesis are novel achievements of this constantly evolving area. Hence they must go through several adjustments based on substantial experience of practical applications to obtain relevance in the industry. Hopefully they will promote the use of formal methods and advance the efficiency of present verification techniques.

Appendix A: Basic Concepts of Category Theory

Hereby we introduce the necessary fundamental concepts of category theory for clarity.

Definition A.1.1. ***(Category** [Pie91]) A **category C** comprises*

1. *a collection of **objects**;*
2. *a collection of **morphisms**;*
3. *operations assigning to each morphism f an object $dom f$, its **domain**, and an object $cod f$, its **codomain** (we write $f : A \to B$ or $A \xrightarrow{f} B$ to show that $dom\ f = A$ and $cod\ f = B$; the collection of all morphisms with a domain A and codomain B is written $\mathbf{C}(A, B)$);*
4. *a composition operator assigning to each pair of morphisms f and g, $cod\ f = dom\ g$, a **composite** morphism $g \circ f : dom\ f \to cod\ g$, satisfying the following associative law:*

   for any arrows $f : A \to B$, $g : B \to C$, and $h : C \to D$ (with A,B,C and D not necessarily distinct),

   $$h \circ (g \circ f) = (h \circ g) \circ f$$

5. *for each object A, an **identity** morphism $id_A : A \to A$ satisfying the following* identity law*:*

   for any morphism $f : A \to B$, $id_B \circ f = f$ and $f \circ id_A = f$.

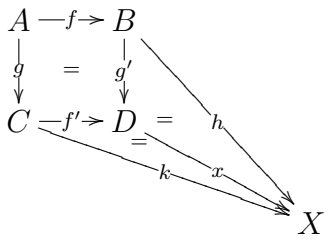
In graph transformations, the gluing of graphs along a common subgraph is a common task. The idea of a pushout generalises the gluing construction in the sense

of category theory, i.e. a pushout object emerges from gluing two objects along a common subobject.

Definition A.1.2. ***(Pushout** [EEPT06])* *Given morphisms $g : A \to B$ and $g : A \to C$ in a category* $\mathbf{C}$*, a* pushout (D, f', g') *over f and g is defined by*

- *a pushout object D and*
- *morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \circ g = g' \circ f$*

such that the following universal property is fulfilled: For all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$, there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$:



We shall use the abbreviation PO for pushout. We use $D = B +_A C$ for the pushout object D, where D is called the gluing of B and C via A, or more precisely, via (A, f, g).

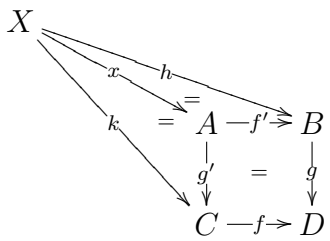
The pushout object D is unique up to isomorphism

The dual construction of a pushout is a pullback. Pullbacks are the generalisation of both intersection and inverse image.

Definition A.1.3. ***(Pullback** [EEPT06])* *Given morphisms $f : C \to D$ and $g : B \to D$ in category* $\mathbf{C}$*, a* pullback (A, f', g') *over f and g is defined by*

- *a pullback object A and*
- *murphisms $f' : A \to B$ and $g' : A \to C$ with $g \circ f' = f \circ g'$*

such that the following universal property is fulfilled: For all objects X with morphisms $h : X \to B$ and $k : X \to C$ with $f \circ k = g \circ h$, there is a unique morphism $x : X \to A$ such that $f' \circ x = h$ and $g' \circ x = k$:

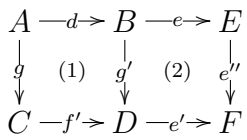


We shall use the abbreviation PB for pullback.

The pullback object A is unique up to isomorphism.

The uniqueness, composition and decomposition properties are essential for the theory of graph transformation.

Theorem A.1.1. ***(Composition and Decomposition of POs and PBs** [EEPT06])* *Given a category* **C** *with the following commutative diagram, the statements below are valid:*



Pushout composition and decomposition:

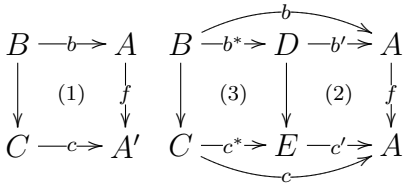
- *if* (1) *and* (2) *are pushouts, then* (1) + (2) *is also a pushout.*
- *if* (1) *and* (1) + (2) *are pushouts, then* (2) *is also a pushout.*

Pullback composition and decomposition:

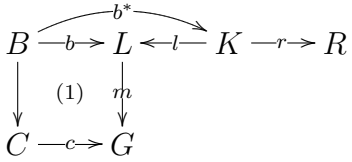
- *if* (1) *and* (2) *are pullbacks, then* (1) + (2) *is also a pullback.*
- *if* (2) *and* (1) + (2) *are pullbacks, then* (1) *is also a pullback.*

Initial pushout is a complement construction. The context graph C as shown in [EEPT06] is the smallest subgraph of A' that contains $A' \setminus f(A)$.

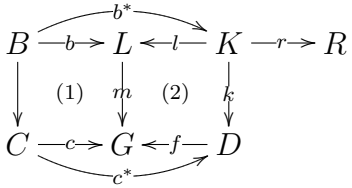
Definition A.1.4. ***(Initial Pushout** [EEPT06])* *Given a morphism $f : A \to A'$, an injective morphism $b : B \to A$ is called the* boundary *over f if there is a pushout complement of f and b such that (1) is a pushout initial over f. Initiality of (1) over f means that for every pushout (2) with injective b' there exists unique morphism $b^* : B \to D$ and $c^* : C \to E$ with injective b^* and c^* such that $b' \circ b^* = b$, $c' \circ c^* = c$ and (3) is a pushout. B is then called the boundary object and C the context with respect to f.*



Definition A.1.5. ***(Gluing Condition with Initial Pushouts** [EEPT06])* *In* $\mathbf{Graphs_{TG}}$ *a match* $m : L \to G$ *satisfies the gluing condition with respect to a production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *if, for the initial pushout* (1) *over* m*, there is a morphism* $b^* : B \to K$ *such that* $l \circ b^* = b$*:*



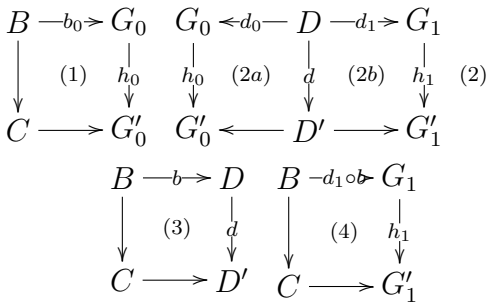
Theorem A.1.2. ***(Existence and Uniqueness of Contexts** [EEPT06])* *In* $\mathbf{Graphs_{TG}}$ *a match* $m : L \to G$ *satisfies the gluing condition with respect to a production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *if and only if the context object* D *exists, i.e. there is a pushout complement* (2) *of* l *and* m*:*



Lemma A.1.1. ***(Closure Property of Initial POs** [EEPT06])* *Given an initial pushout* (1) *over injective* h_0 *and double pushout diagram* (2) *with pushouts* (2*a*) *and* (2*b*) *and injective* d_0, d_1*, we have the following:*

1. *The composition of* (1) *with* (2*a*)*, defined as pushout* (3) *by the initiality of* (1)*, is an initial pushout over injective morphism* d*.*

2. *The composition of the initial pushout* (3) *with pushout* (2*b*)*, leading to pushout* (4)*, is an initial pushout over injective morphism* h_1



Bibliography

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382, 1998.

[AGG07] AGG - Attributed Graph Grammar System Environment. `http://tfs.cs.tu-berlin.de/agg`, 2007.

[Ala04] Alan Brown. *An introduction to Model Driven Architecture*. IBM, 2004. `http://www.ibm.com/developerworks/rational/library/3100.html`.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[Amb04] Scott W. Ambler. *The Object Primer: Agile Modeling-Driven Development with UML 2*. Cambridge University Press, 2004.

[Arn86] Robert S. Arnold, editor. *Tutorial on software restructuring*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[BBER08] J. Bauer, I. Boneva, Kurbán M. E., and A. Rensink. A modal-logic based graph abstraction. In *ICGT 2008*, 2008.

[BCK04] P. Baldan, A. Corradini, and B. König. Verifying finite-state graph tranformation grammars: an unfolding-based approach. In *Proceedings of CONCUR '04*, number 3170 in LNCS, pages 83–98. Springer, 2004.

[BCK08] P. Baldan, A. Corradini, and B. König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 2008. to appear.

[BCM98] P. Baldan, A. Corradini, and U. Montanari. Concatenable graph processes: relating processes and derivation traces. In S. Larsen, K. Skyum and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 283–295. Springer Verlag, 1998.

[BCM99] P. Baldan, A. Corradini, and U.. Montanari. Unfolding and event structure semantics for graph grammars. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, number 1578 in LNCS, pages 73–89. Springer, 1999.

[BCMR07] P. Baldan, A. Corradini, U. Montanari, and L. Ribeiro. Unfolding Semantics of Graph Transformation. *Information and Computation*, 205:733–782, 2007.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems - an introduction to focus. Technical Report TUM-I9202, Technische Univerität München, 1992.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Emf model refactoring based on graph transformation concepts. *ECEASST*, 3, 2006.

[Ber91] Paul L. Bergstein. Object-preserving class transformations. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 299–313, New York, NY, USA, 1991. ACM.

[BGMM08] L. Baresi, C. Ghezzi, A. Mocci, and M. Monga. Using graph transformation systems to specify and verify data abstractions. In Claudia Ermel, Reiko Heckel, and Juan de Lara, editors, *Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, volume X, pages 277–290, Budapest, Hungary, March 2008. EASST.

[BH07] Dénes Bisztray and Reiko Heckel. Rule-level verification of business process transformations using csp. In *Proc of 6th International Workshop on Graph Transformations and Visual Modeling Techniques (GTVMT'07)*, 2007.

[BHE08] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Verification of architectural refactorings by rule extraction. In *Fundamental Approaches to Software Engineering*, volume 4961/2008 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin / Heidelberg, 2008.

[BHE09a] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositional verification of architectural refactorings. In *Architecting Dependable Systems VI*, Lecture Notes in Computer Science, pages 308–333. Springer, 2009.

[BHE09b] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositionality of model transformations. *Electronic Notes in Theoretical Computer Science*, 236:5–19, 2009.

[BHE09c] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Verification of architectural refactorings: Rule extraction and tool support. *Electronic Communications of the EASST*, 16, 2009.

[Bis08] Dénes Bisztray. Verification of architectural refactorings: Rule extraction and tool support. In *Graph Transformations*, Lecture Notes in Computer Science, pages 475–477. Springer Berlin / Heidelberg, 2008.

[BK02] P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *Proceedings of ICGT '02*, number 2505 in LNCS, pages 14–30. Springer, 2002.

[BKK03] P. Baldan, B. König, and B. König. A logic for analysing abstractions of graph transformation systems. In R. Cousot, editor, *Proceedings of SAS '03*, number 2694 in LNCS, pages 255–272. Springer, 2003.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 16(3):311–322, 1987.

[BKPPT05] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Termination of high-level replacement units with application to model transformation. *Electr. Notes Theor. Comput. Sci.*, 127(4):71–86, 2005.

[BKR05] P. Baldan, B. König, and A. Rensink. Graph grammar verification through abstraction. In *Graph transforamtion and process algebras for*

modeling distributed and mobile systems, volume 04241 of *Dagstuhl Seminar*, 2005.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns*, volume Volume 1 of *Pattern-Oriented Software Architecture*. John Wiley and Sons, 1st edition, August 1996.

[BPPT03] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Coordinated distributed diagram transformation for software evolution. *Electronic Notes in Theoretical Compututer Science*, 72(4), 2003.

[BPPT04] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Specifying integrated refactoring with distributed graph transformations. In *Applications of Graph Transformations with Industrial Relevance*, volume Volume 3062/2004 of *Lecture Notes in Computer Science*, pages 220–235. Springer Berlin / Heidelberg, 2004.

[CCI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CMR96] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.

[CMR+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars*, pages 163–246, 1997.

[CW04] Alexandre L. Correa and Cláudia Maria Lima Werner. Applying refactoring techniques to uml/ocl models. In *UML*, pages 173–187, 2004.

[DB08] Hartmut Ehrig Dénes Bisztray, Reiko Heckel. Verification of architectural refactoring rules. Technical report, Department of Computer Science, University of Leicester, 2008. `http://www.cs.le.ac.uk/people/dab24/refactoring-techrep.pdf`.

[DDN08] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.

[DLRdS03] F. L. Dotti, Foss L., L. Ribeiro, and O. M. dos Santos. Verification of distributed object-based systems. In *FMOODS '03*, pages 261–275, 2003.

[DMdS05] F. L. Dotti, O. M. Mendizabal, and O. M. dos Santos. Verifying fault-tolerant distributed systems using object-based graph grammars. In *LADC '05*, pages 80–100, 2005.

[Ecl09] Eclipse Integrated Development Environment. `http://www.eclipse.org/`, 2009.

[EEdL+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *Proc. FASE 2005: Internation Conference on Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 49–63, Edinburgh, UK,, April 2005. Springer.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Ehr87] Hartmut Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 3–14, London, UK, 1987. Springer-Verlag.

[EJ03] Niels Van Eetvelde and Dirk Janssens. A hierarchical program representation for refactoring. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[EK06] Hartmut Ehrig and Barbara Koenig. Deriving bisimulation congruences in the dpo approach to graph rewriting with borrowed contexts. *Mathematical Structures in Computer Science*, 16(6), 2006.

[EMF07] Eclipse Modeling Framework. `http://www.eclipse.org/emf`, 2007.

[EPLF03] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit*. Wiley Publishing, 2003.

[Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, 1994.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition edition, 1999.

[FFR07] A. P. L. Ferreira, L. Foss, and L. Ribeiro. Formal verification of object-oriented graph grammars specifications. *Electron. Notes Theor. Comput. Sci.*, 175(4):101–114, 2007.

[FR98] Richard Fanta and Václav Rajlich. Reengineering object-oriented code. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 238, Washington, DC, USA, 1998. IEEE Computer Society.

[FSEL05] Formal Systems Europe Ltd. *FDR2 User Manual*, 2005. http://www.fsel.com/documentation/fdr2/html/index.html.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[GHK98] F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-intepreted temporal logic. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[GJ02] Alejandra Garrido and Ralph Johnson. Challenges of refactoring c programs. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 6–14, New York, NY, USA, 2002. ACM.

[GMF07] Eclipse Graphical Modeling Framework. `http://www.eclipse.org/gmf`, 2007.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[GP95] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, 1995.

[Hec98a] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Proceedings of FASE'98*,

volume 1382 of *Lecture Notes in Computer Science*, pages 138–153. Springer Verlag, 1998.

[Hec98b] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In *FASE '98*, volume 1382 of *LNCS*, pages 138–153. Springer-Verlag, 1998.

[Hen88] Matthew Hennessy. *Algebraic theory of processes*. MIT Press, Cambridge, MA, USA, 1988.

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[HJvE06] Berthold Hoffmann, Dirk Janssens, and Niels van Eetvelde. Cloning and expanding graph transformation rules for refactoring. *Electronic Notes Theoretical Compututer Science*, 152:53–67, 2006.

[HKT02a] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 161–176, London, UK, 2002. Springer-Verlag.

[HKT02b] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 161–176, London, UK, 2002. Springer-Verlag.

[HLM06] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Stochastic graph transformation systems. *Fundamenta Informaticae*, 72:1–22, 2006. To appear.

[Hoa85] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, April 1985.

[Hof05] Berthold Hoffmann. Graph transformation with variables. In *Formal Methods in Software and Systems Modeling*, pages 101–115, 2005.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus: A tool for distributed systems specification. In *FTRTFT*, pages 467–470, 1996.

[Int96] International Standard Organisation. *Information technology - Syntactic metalanguage - Extended BNF*, 1996. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153`.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. *SIGPLAN Not.*, 29(6):171–185, 1994.

[Kas05] H. Kastenberg. Towards attributed graphs in groove. In *GT-VC '05*, Electronic Notes in Computer Science. Elseviee, 2005.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. The Addison-Wesley Signature Series. Addison Wesley Professional, 2004.

[KGK+08] Jana Koehler, Thomas Gschwind, Jochen Küster, Cesare Pautasso, Ksenia Ryndina, Jussi Vanhatalo, and Hagen Völzer. Combining quality assurance and model transformations in business-driven development. pages 1–16, 2008.

[KK06] B. Koenig and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *TACAS 2006*, volume 3920 of *LNCS*, pages 197–211. Springer, 2006.

[KK08] B. Koenig and V. Kozioura. Towards the verification of attributed graph transformation systems. In *ICGT 2008*, 2008.

[KMJ02] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. *SIGPLAN Not.*, 37(11):231–245, 2002.

[Koc00] M. Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems*. PhD thesis, Technische Universität Berlin, 2000.

[KR06] H. Kastenberg and A. Rensink. Model checking dynamic states in groove. In A. Valmari, editor, *SPIN '06*, number 3925 in LNCS, pages 229–305, 2006.

[Kre02] Peter Kreeft. *How to Win the Culture War*. InterVarsity Press, 2002.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2003.

[Lae25] Diogenes Laertius. *Lives of Eminent Philosophers*, volume 2. Loeb Classical Library, January 1925.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[LEO08] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Efficient conflict detection in graph transformation systems by essential critical pairs. *Electron. Notes Theor. Comput. Sci.*, 211:17–26, 2008.

[LEPO08a] Leen Lambers, Hartmut Ehrig, Ulrike Prange, and Fernando Orejas. Embedding and confluence of graph transformations with negative application conditions. In *ICGT '08: Proceedings of the 4th International Conference on Graph Transformations*, pages 162–177, Berlin, Heidelberg, 2008. Springer-Verlag.

[LEPO08b] Leen Lambers, Hartmut Ehrig, Ulrike Prange, and Fernando Orejas. Parallelism and concurrency in adhesive high-level replacement systems with negative application conditions. *Electronic Notes in Theorerical Computer Science*, 203(6):43–66, 2008.

[LLC06] Tihamr Levendovszky, Lszl Lengyel, and Hassan Charaf. Termination Properties of Model Transformation Systems with Srtict Control Flow. In *5th International Workshop on Graph Transformation and Visual Modeling Techniques*, Vienna, Austria, April 2006.

[LPE07] Tihamér Levendovszky, Ulrike Prange, and Hartmut Ehrig. Termination criteria for dpo transformations with injective matches. *Electron. Notes Theor. Comput. Sci.*, 175(4):87–100, 2007.

[LRW+97] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[Mar96] Alfio Martini. Elements of basic category theory. Technical report, Technische Universität Berlin, 1996.

[MB08] Slavisa Markovic and Thomas Baar. Refactoring ocl annotated uml class diagrams. *Software and System Modeling*, 7(1):25–47, 2008.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[MCvG05] Tom Mens, Krzysztof Czarnecki, and Pieter van Gorp. A taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[MDJ02] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 286–301, London, UK, 2002. Springer-Verlag.

[MGB06] Tiago Massoni, Rohit Gheyi, and Paulo Borba. An approach to invariant-based program refactoring. *ECEASST*, 3, 2006.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[MM03] Jishnu Mukerji and Joaquin Miller. *MDA Guide Version 1.0.1*. OMG, 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[MTR07] Tom Mens, Gabi Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling (SoSyM)*, pages 269–285, September 2007.

[NG03] Colin J. Neill and Bharminder Gill. Refactoring reusable business components. *IT Professional*, 5(1):33–38, 2003.

[oC02] Free Online Dictionary of Computing. *Unified Modelling Language*, 2002. `http://foldoc.org/UML`.

[OCN98] Mel Ó Cinnéide and Paddy Nixon. Composite refactorings for java programs. In *Proceedings of the Workshop on Formal Techniques for Java Programs*, ECOOP Workshops, 1998.

[OMG06a] OMG. *Object Constraint Language, version 2.0*, 2006. `http://www.omg.org/technology/documents/formal/ocl.htm`.

[OMG06b] OMG. *Unified Modeling Language, version 2.1.1*, 2006. `http://www.omg.org/technology/documents/formal/uml.htm`.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[PC07] Javier Pérez and Yania Crespo. Exploring a method to detect behaviour-preserving evolution using graph transformation. In van Paesschen Ellen Maja DHondt Tom Mens, Kim Mens, editor, *Proceedings of the Third International ERCIM Workshop on Software Evolution*, pages 114–122. ERCIM, October 2007. Informal Workshop proceedings.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press, 1991.

[Plu93] Detlef Plump. Hypergraph rewriting: critical pairs and undecidability of confluence. pages 201–213, 1993.

[Plu99] D. Plump. Term graph rewriting. pages 3–61, 1999.

[Por02] Ivan Porres. A toolkit for manipulating uml models. Technical report, Turku Centre for Computer Science, 2002.

[Por03] Ivan Porres. Model refactorings as rule-based update transformations. In *UML*, pages 159–174, 2003.

[PR97] Jan Philipps and Bernhard Rumpe. Refinement of information flow architectures. In *ICFEM*, pages 203–212, 1997.

[QVT05] MOF Query/View/Transformation (QVT) Final Adopted Specification. `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[Rei85] W. Reisig. *Petri Nets: An Introduction*. EACTS Monographs on Theoretical Computer Science. Springer Verlag, 1985.

[Ren03] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. Lo Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

[Ren04a] Arend Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *Programming Languages and Systems — European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 2004.

[Ren04b] Arend Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[Ren04c] Arend Rensink. State space abstraction using shape graphs. In *Automatic Verification of Infinite-State Systems (AVIS)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[Ren08] A. Rensink. Explicit state model checking for graph grammars. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*. Springer Verlag, 2008.

[Rib96] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.

[RKE07] Guilherme Rangel, Barbara König, and Hartmut Ehrig. Bisimulation verification for the dpo approach with borrowed contexts. *ECEASST*, 6, 2007.

[RLK+08] Guilherme Rangel, Leen Lambers, Barbara König, Hartmut Ehrig, and Paolo Baldan. Behavior preservation in model refactoring using dpo transformations with borrowed contexts. In *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, pages 242–256, Berlin, Heidelberg, 2008. Springer-Verlag.

[Rob99] Donald B Roberts. Practical analysis for refactoring. Technical report, Champaign, IL, USA, 1999.

[Ros97] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1st edition, November 1997.

[RSV04] A. Rensink, A. Schmidt, and D. Varró. Model checking graph transformations: a comparison of two approaches. In *ICGT '04*, number 3256 in LNCS, pages 226–241. Springer, 2004.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*, number 903, pages 151–163. Springer-Verlag, 1994.

[Sel05] Bran Selic. *Unified Modeling Language version 2.0: In support of model-driven development*, 2005. `http://www.ibm.com/developerworks/rational/library/05/321_uml/?S_TACT=105AGX78&S_CMP=HP`.

[SPTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK, 2001. Springer-Verlag.

[Ste07] Perdita Stevens. Bidirectional model transformations in qvt: Semantic issues and open questions. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, volume 4735, pages 1–15. Springer LNCS, October 2007.

[Ste08] Perdita Stevens. A landscape of bidirectional model transformations. pages 408–424, 2008.

[Sun09] Sun Microsystems. *Enterprise JavaBeans Technology*, 2009. `http://java.sun.com/products/ejb/index.jsp`.

[SV03] A. Schmidt and D. Varró. CheckVML: a tool for model checking visual modeling languages. In *UML '03*, volume 2863 of *LNCS*, pages 92–95. Springer-Verlag, 2003.

[TB01] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1):89–120, 2001.

[TG07] Mirco Tribastore and Stephen Gilmore. The PEPA Project for Eclipse. `http://homepages.inf.ed.ac.uk/mtribast/`, 2007.

[THD03] Kevin C. Desouza Thomas H. Davenport, Robert I. Thomas. *Reusing intellectual assets*, 2003. `http://www.entrepreneur.com/tradejournals/article/105440989.html`.

[Tig07] Tiger Developer Team. *Tiger EMF Transformer*, 2007. `http://www.tfs.cs.tu-berlin.de/emftrans`.

[TKB03] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 13–26, New York, NY, USA, 2003. ACM.

[TM07] Gabriele Taentzer, Dirk Müller 0002, and Tom Mens. Specifying domain-specific refactorings for andromda based on graph transformation. In *AGTIVE*, pages 104–119, 2007.

[VAB+08] Dániel Varró, Márk Asztalos, Dénes Bisztray, Artur Boronat, Duc-Hanh Dang, Rubino Geiß, Joel Greenyer, Pieter Gorp, Ole Kniemeyer, Anantha Narayanan, Edgars Rencis, and Erhard Weinell. Transformation of uml models to csp: A case study for graph transformation tools. pages 540–565, 2008.

[Var04] D. Varró. Automated formal verification of visual modelign languages. *Software and System Modeling*, 3(2):85–113, 2004.

[Var06] Dániel Varró. Model transformation by example. In *Proc. Model Driven Engineering Languages and Systems (MODELS 2006)*, volume 4199 of *LNCS*, pages 410–424, Genova, Italy, 2006. Springer.

[vEJ04] Niels van Eetvelde and Dirk Janssens. Extending graph rewriting for refactoring. In *Graph Transformations*, volume Volume 3256/2004 of *Lecture Notes in Computer Science*, pages 399–415. Springer Berlin / Heidelberg, 2004.

[vGSMD03] Pieter van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158. Springer-Verlag, 2003.

[Vit03] Marian Vittek. Refactoring browser with preprocessor. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance*

and Reengineering, page 101, Washington, DC, USA, 2003. IEEE Computer Society.

[vKCKB05] Marc van Kempen, Michel Chaudron, Derrick Kourie, and Andrew Boake. Towards proving preservation of behaviour of refactoring of uml models. In *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 252–259, , Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[VS09] Microsoft Visual Studio. `http://msdn.microsoft.com/en-us/vstudio/default.aspx`, 2009.

[WCG+06] Martin Wirsing, Allan Clark, Stephen Gilmore, Matthias Hölzl, Alexander Knapp, Nora Koch, and Andreas Schroeder. Semantic-Based Development of Service-Oriented Systems. In E. Najn et al., editor, *Proc. 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems(FORTE'06), Paris, France*, LNCS 4229, pages 24–45. Springer-Verlag, 2006.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979.