Verification of Architectural Refactoring Rules

Dénes Bisztray, Reiko Heckel, Hartmut Ehrig

Department of Computer Science,
University of Leicester,
dab24@mcs.le.ac.uk

October 27, 2008

Abstract

With the success of model-driven development as well as component-based and service-oriented systems, models of software architecture are key artefacts 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.

To avoid the costly verification of refactoring steps on large systems we present a method which allows us to extract a (usually much smaller) rule from the transformation performed and verify this rule instead. The main result of the paper shows that the verification of rules is indeed sufficient to guarantee the desired semantic relation between source and target models. We apply the approach to the refactoring of architectural models based on UML component, structure, and activity diagrams, with using CSP as a semantic domain.

Contents

Chapter 1

Introduction

Nothing endures but change, as the philosopher says [10]. As much as anywhere else, this applies to the world of software. In order to improve the internal structure, performance, or scalability of software systems, changes may be required that preserve the observable behaviour of systems. In OO programming, such behaviour-preserving transformations are known as refactorings [7]. Today, where applications tend to be distributed and service-oriented, the most interesting changes take place at the architectural level. Even if these 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 refactorings if preservation of observable behaviour is intended.

In this paper, refactoring is addressed at the level of models. Given a transformation from a source to a target model we would like to be able to verify their relation. In order to make this precise we have to fix three ingredients: the modelling language used, its semantics, and the relation capturing our idea of behaviour preservation. Notice however that in the mathematical formulation of our approach, these parameters can be replaced by others, subject to certain requirements. For *modelling language* we use the UML, which provides the means to describe both structure (by component and static structure diagrams) and behaviour (by activity diagrams) of service-oriented systems [17]. The *semantics* of the relevant fragment of the UML is expressed in a denotational style, using CSP [9] as semantic domain and defining the mapping from UML diagrams to CSP processes by means of graph transformation rules. As different UML diagrams are semantically overlapping, the mapping has to produce one single consistent semantic model [5]. The *semantic relation* of behaviour preservation can conveniently be expressed using one of the refinement and equivalence relations on CSP processes.

Based on these (or analogue) ingredients, we can formalise the question by saying that a model transformation $M_1 \to M_2$ is behaviour-preserving if $sem(M_1)\ \mathcal{R}\ sem(M_2)$ where *sem* represents the semantic mapping and $\mathcal{R}$ the desired relation on the semantic domain. However, the verification of relation $\mathcal{R}$ over sufficiently large M_1 and M_2 can be very costly, while the actual refactoring might only affect a relatively small fragment of the overall model. Hence, it would be advantageous if we could focus our verification on those parts of the model that have been changed, that is, verify the refactoring *rules* rather than the actual steps. This is indeed possible, as we show in this paper, if both semantic mapping *sem* and semantic relation $\mathcal{R}$ satisfy suitable compositionality properties. We satisfy these requirements by specifying the mapping *sem* by graph transformation rules of a certain format and choosing CSP refinements as semantic relations.

However, model-level architectural refactorings are unlikely to be created directly from semantics-preserving rules. Such rule catalogues as exist 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 $M_1 \to M_2$ from which a verifiable refactoring rule has to be extracted first. In this we follow the idea of *model transformation by example* [23] where model transformation rules expressed as graph transformations are derived from sample transformations.

The paper is structured as follows. In Sect. 3 we present our architectural models along with an example, on which a refactoring step is performed in Sect. 4. Section 6 introduces CSP as the semantic domain and describes the mapping and the semantic relation. The formal justification for rule-level verification is discussed in Sect. 7. It is demonstrated that the method is sound if the semantic mapping is compositional, which is true based on a general result which derives this property from the format of the mapping rules. Section 9 concludes the paper.

Chapter 2

Related Work

After refactorings for Java were made popular by Fowler [7], several proposals for formalisation and verification based on first-order logics and invariants have been made [20, 13, 15]. The first formal approach to refactoring based on graph transformations is due to Mens [16], focusing on the analysis of conflicts and dependencies between rules.

Refactoring of architectural models has been studied formally in architectural description languages (ADLs) like WRIGHT [1] or Darwin [14], using process calculi like CSP or π-calculus for expressing formal semantics. Our semantic mapping to CSP follows that of [5] for UML-RT [19], an earlier component-based extension to the UML, but distinguishes type and instance level architectural models in UML 2.

A number of authors have studied instance level architectural transformations, or reconfigurations. For example, Taentzer [21] introduces the notion of distributed graph transformation systems to allow architectural reconfiguration by means of two-level rules to express system-level and local transformations. The approach of [24] uses an algebraic framework to represent reconfigurations based on the coordination language Community. In [8] the architecture is represented by hypergraphs, where the hyperedges are the components, and the nodes are the communication ports. Architectural reconfigurations are represented by synchronised hyperedge replacement rules.

Our approach combines the type level, typical of source code refactoring, which happens at the level of classes, with the instance level typical of architectural transformations.

Chapter 3

Architectural Models

This chapter presents our choice of architectural modelling language by means of an example based on the *Car Accident Scenario* from the SENSORIA Automotive Case Study [25].

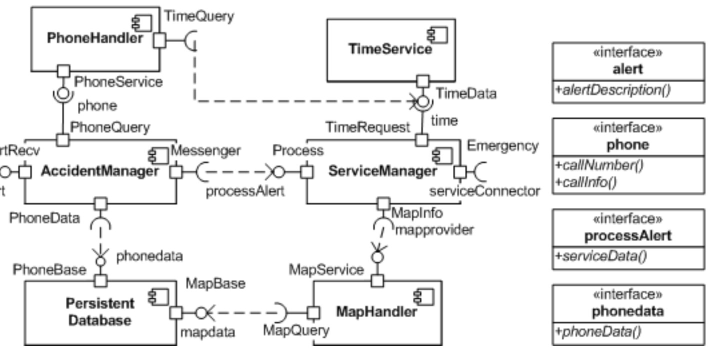
We use UML *component* and *composite structure diagrams* for representing the type and instance-level architecture of our system in conjunction with *activity diagrams* specifying the workflows executed by component instances [17].

Briefly, the scenario is as follows. A car company is offering a service by which, in case one of the sensors in their car detects an accident, customers are contacted via their mobile phones to check if they require assistance. If they do, a nearby ambulance is dispatched. The system consists of three main parts: the agent in the car, the accident server, and the interface to the local emergency services. We present the architecture and behaviour of the accident server in detail.

3.1 Type-Level

Component diagrams specify the components, ports, interfaces that make up the building blocks of the system. Figure 3.1(a) shows the component diagram of the accident server.

The *AccidentManager* is the core component, responsible for receiving



(a) Component Diagram (b) Interfaces

Figure 3.1: Architectural model of the accident server

incoming alerts from cars through the *AlertRecv* port. In order to initiate a phone call it acquires the number of the driver from the *PersistentDatabase*, and passes it to the *PhoneService*, which calls the driver. In case the driver replies saying that assistance is not required, the alert is cancelled. Otherwise, the call is returned to the *AccidentManager*, which assesses the available data (including sensorial and location data from the car) and decides if the situation is a real emergency. In this case it passes the necessary data to the *ServiceManager*, which matches the GPS location of the car using the *MapHandler*, creates a service description, and contacts the *serviceConnector* interface that provides access to local emergency services.

In the diagram, components are represented by rectangles with a component icon and classifier name. Smaller squares on the components represent the *ports*, *provided interfaces* are represented by circles and *required interfaces* by a socket shape [17]. Dashed arrows represent dependencies between the provided and required interfaces.

3.2 Instance-Level

The composite structure diagram specifying the configuration of the accident server is shown in Figure 3.2. Boxes named *instance : type* represent component instances. Ports are typed by interfaces defining the possible actions that can happen through that port. For instance, the possible actions of the *PhoneQuery* port are defined by the *phone* interface. Links between port instances represent connectors, enabling communication between component instances [17].

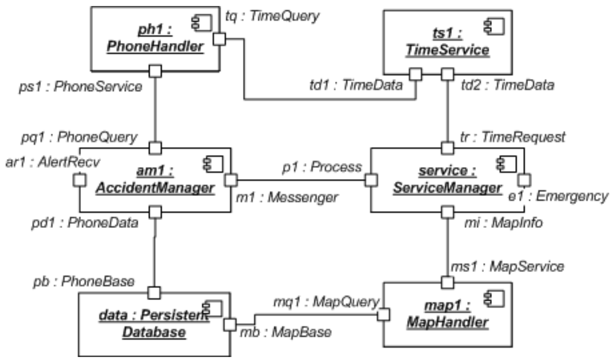


Figure 3.2: Static Structure Diagram of the Accident Server

3.3 Behaviour

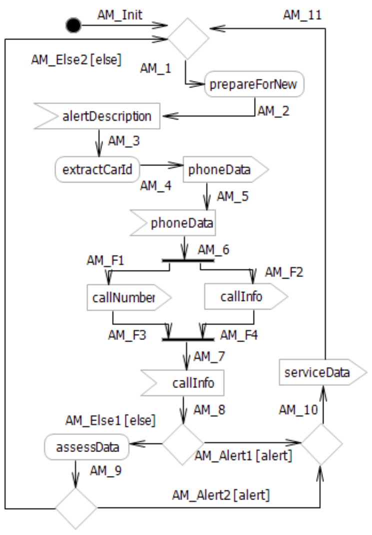


Figure 3.3: Activity Diagram of the AccidentManager component

The behaviours of components are described by *activity diagrams*, like the one depicted in Figure 3.3 associated with the *AccidentManager* component.

Apart from the obvious control flow constructs they feature *accept event actions*, denoted by concave pentagons, that wait for the occurrence of specific events triggered by *send signal actions*, shown as convex pentagons [17]. They fit into the communication framework by representing functions calls from the corresponding *interface* through the relevant *port*. For instance, the *phoneData* send signal action in Fig. 3.3 represents the function call from *phone* interface through *PhoneQuery* port.

3.4 Metamodel

Formally, the UML models are instances of metamodels represented by attributed typed-graphs. The combined metamodel of the three diagrams is shown in Figure 3.4.

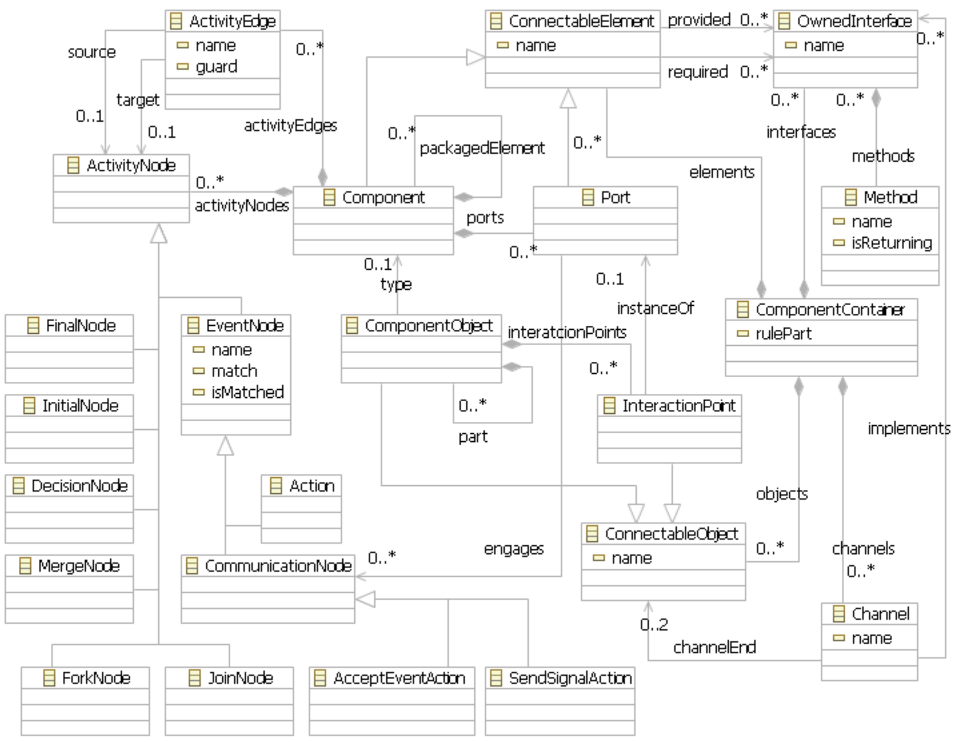


Figure 3.4: Metamodel for the Structure Model

The reason for including the activity diagram and component diagram is the interconnection between the different nodes. Not only the component is the container of activity nodes and edges, but also the ports are engaged with communication nodes. The component diagram is the metamodel of a composit structure diagram, thus it should be an abstraction level higher. However both are included in the same metamodel. Unfortunately there is no tool (and theoretical) support for graph transformations that spans more than two abstraction levels.

Aside from the elements that have a self interpreting name, the component container contains all elements corresponding to one particular side of the refactoring rule. In the activity diagram part, the event node is a common parent for all nodes that represents an event. Communication node represents the special nodes that engage in communication through a port. On instance level, component object is the instance of a component, while the interaction point is the instance of a port. Connectable object is a similar base class on instance level as the connectable element. Channels are semantically not instances of owned interfaces, they are rather a realisation of them.

Chapter 4

Model Refactoring

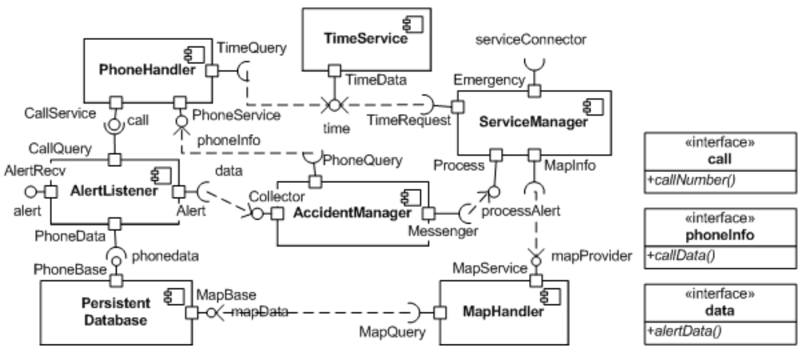
Continuing the description of the case study, a refactoring is presented in this chapter.

4.1 Description of the Refactoring

With the current architecture 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 solution is depicted in Figure 4.1. 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 to the *AccidentManager*.

The behaviour of the new *AlertListener* component is given in Figure 4.3(a), while the updated behaviour of the *AccidentManager* is shown in Figure 4.3(b).



(a) Component Diagram (b) Interfaces

Figure 4.1: Architectural model of the refactored Accident Server

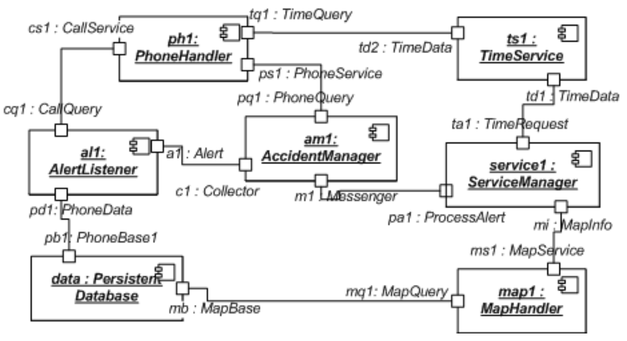
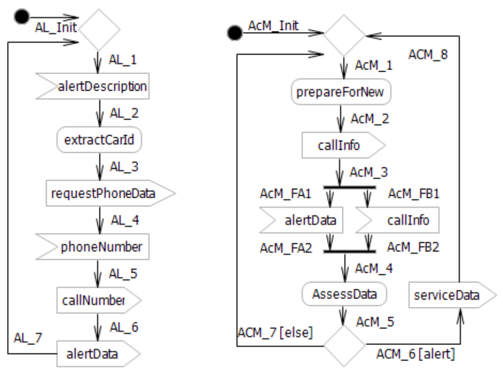


Figure 4.2: Configuration after the refactoring

4.2 Rule Extraction

However, rather than comparing the semantics of the entire system model before and after the change, we focus on the affected parts and their immediate context. More precisely, we are proposing to extract a model transformation *rule* which, (1) when applied to the source model produces the target model of the refactoring and (2) is semantics preserving in the sense that its left-hand side is in the desired semantic relation with its right-hand side. We will demonstrate in Sect. 7 that this is indeed sufficient to guarantee the corresponding relation between source and target model. In the example present, such a rule is shown in Fig. 4.4 for the structural part only. The behaviour transformation is given by the new and updated activity diagrams associated with the components in the rule.



(a) AlertListener (b) AccidentManager

Figure 4.3: Owned behaviour after the refactoring

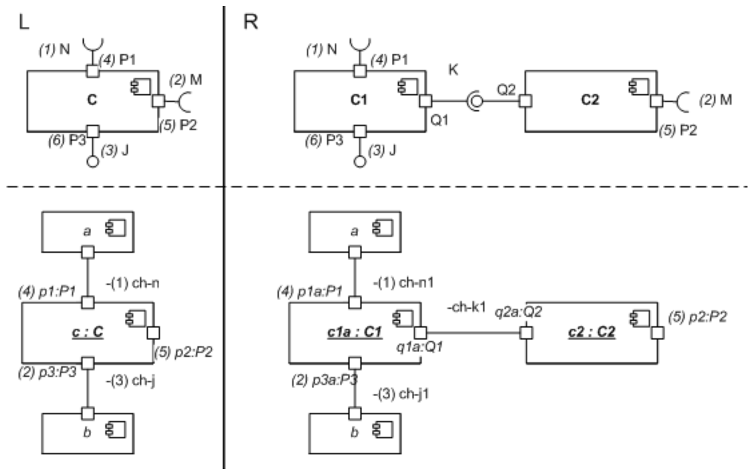


Figure 4.4: Refactoring rule

The rule is applied by selecting in the source model an occurrence isomorphic to the left-hand side of the rule at both type and instance level. Thus, component C is matched by *AccidentManager* from Fig. 3.1(a), interface N corresponds to *phone*, M to *processAlert*, and J to *phoneData*. At instance level a similar correspondence is established.

A rule is extracted as follows: G denotes the original model while H denotes the refactored one. The smallest consistent submodel of G containing $G \setminus H$ would form the left-hand side L of the rule, while the smallest submodel of H containing $H \setminus G$ would form the right hand side R. In the algebraic approach to graph transformation, which provides the formal background of this work, this is known as Initial Pushout Construction [4].

Recently a similar construction has been used as part of the *model transformation by example* approach, where a transformation specification is derived inductively from a set of sample transformation rules [23]. Notice that while the rule thus obtained is known to achieve the desired transformational effect, it is not in general guaranteed that the semantic relation between L and R can indeed be verified, even if it holds between G and H. The reason is that additional context information present in G and H may be required to ensure semantic compatibility. It is the responsibility of the modeller to include this additional context into the rule. However, as in the example presented, a minimal rule might not be enough because some additional context may have to be taken into account in order to guarantee the preservation of the semantics. In the example this has led to the introduction into the rule of generic component instances a and b (the PhoneHandler and Database in the concrete model).

The example illustrates the potential complexity of the problem at hand, with changes in all three diagrams types to be coordinated in order to lead to an equivalent behaviour. In the following section we will see how the combined effect of these three models is reflected in the semantic mapping to CSP.

Chapter 5

Semantic Domain

In this section the introduction of CSP, the chosen semantic domain is presented. As the theoretical results proposed in Chapter 7 and 8 are generic with respect to semantic domain, we propose its formal definition. Moving from the abstract to the concrete, after the formal definitions and introductions, the implementation details are provided

Definition 5.0.1 ***(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* $C[\ ] \in \mathcal{C} : D \to D$*, called contexts, such that* $d \sqsubseteq e \implies C[d] \sqsubseteq C[e]$ *(*$\sqsubseteq$ *is closed under contexts).*

The equivalence relation $\equiv$ *is the symmetric closure of* $\sqsubseteq$*. Contexts* C*,* D *are equivalent if* $\forall d \in D$*,* $C[d] \equiv D[d]$*.*

5.1 Communicating Sequential Processes.

Communicating Sequential Processes [9] is a process algebra providing for concurrent systems and supported by tools [6]. A *process* is the behaviour pattern of a component with an alphabet of events. Processes are defined using recursive equations based on the following syntax.

$$P ::= event \to P \mid P \sqcap Q \mid P \;\square\; Q \mid P \parallel Q \mid P \setminus a \mid SKIP \mid STOP$$

The prefix $a \rightarrow P$ performs action a and then behaves like P. The processes $P \sqcap Q$ and $P \square Q$ represent internal and external choice between processes P and Q, respectively. The process $P \mid\mid Q$ behaves as P and Q engaged in a lock-step synchronisation. Hiding $P \setminus a$ behaves like P except that all occurrences of event a are hidden. $SKIP$ represents successful termination, $STOP$ is a deadlock. Due to the distinction of type and instance level, in our application it is important to define groups of processes with similar behaviour. To this end, we use renaming: Each process within a structural group is renamed to a different name, which is also used to distinguish its events. Renaming is an injective function $r : \alpha P \rightarrow A$ that maps the alphabet of process P to a set of symbols A. The renamed process $r(P)$ engages in the event $r(e)$ whenever P would have engaged in e [9].

For clarity, we use the terminology as shown in the expression below. A CSP expression that defines the behaviour of a process is called a process assignment. The definition is the behaviour assigned to the particular process, while the declaration is the name of the process itself.

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

The semantics of CSP is defined in terms of traces, failures, and divergences [9]. A trace of a process behaviour is a finite sequence of events in which the process has engaged up to some moment in time. The complete set of all possible traces of process P is denoted by $traces(P)$. For the three semantics domains, corresponding equivalence and refinement relations can be deducted. Two processes are trace equivalent, i.e. $P \equiv_T Q$ if the traces of P and Q are the same, i.e. $traces(P) = traces(Q)$. Trace refinement means that $P \sqsubseteq_T Q$ if $traces(Q) \subseteq traces(P)$. Hence, every trace of Q is also a trace of P. Analogously the equivalence and refinement relations based on failures and divergences can be defined. These relations shall be used to express behaviour preservation of refactoring rules and compatibility of system components.

CSP is a semantic domain in the sense of Definition 5.0.1. D is the set of CSP expressions and $\sqsubseteq$ can be trace, failure or divergence refinement as they are closed under context [9]. A context is a process expression $E(X)$ with a single occurrence of a distinguished process variable X.

Despite the existence of more expressive mathematical models, the compositional property and tool support are most important to our aim. FDR2 [6] enables the automatic verification of the above mentioned equivalence and refinement relations.

5.2 Representation in FDR2

FDR2 is a refinement checker for establishing properties of models expressed in CSP [6]. As FDR2 is used for refinement checking, we introduce its CSP representation and use it for presenting CSP expressions in the followings. As FDR2 is implementation level, its file format is introduced with the differences to official CSP.

An FDR2 compliant CSP file consists of three major parts: channel definitions, system specifications and system equations. The *channel definition* is a kind of collective alphabet of the described systems: it lists all possible events. The *system specification* is the actual set of CSP expressions that define the behaviour of a system. A file may contain multiple systems. The *system equation* is the root process of a system.

FDR2 treats the process alphabets and their parallel composition in a significantly different way than the official CSP. According to [9], every process has its own, intrinsic alphabet αP. A single parallel composition operator ($P \parallel Q$) is used. The synchronised events are not defined explicitly, they are the intersection of the respective alphabets, i.e. $\alpha P \cap \alpha Q$. An interleaving process $P \mid\mid\mid Q$ is the truly concurrent process, with no synchronisation even if the intersection is not empty.

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. Thus, a parameterised concurrency operator `P[|X|]Q` is used where X is the set of synchronised events. All events outside the set are interleaved.

Except renaming, most of the other operators are represented according to CSP. Renaming has a different notation. Assuming the renaming $r(event_1) = renamed_1$, $r(event_2) = renamed_2$ and $r(P) = R$ in CSP. The respective FDR2 representation is:
`R = P [[renamed1<-event1, renamed2<-event2]].`

5.3 Abstract Syntax

The abstract syntax of CSP can be also represented as a typed graph. The typed graph based metamodel for CSP is defined in this section. This way, the semantic mapping can be defined as a typed graph transformation.

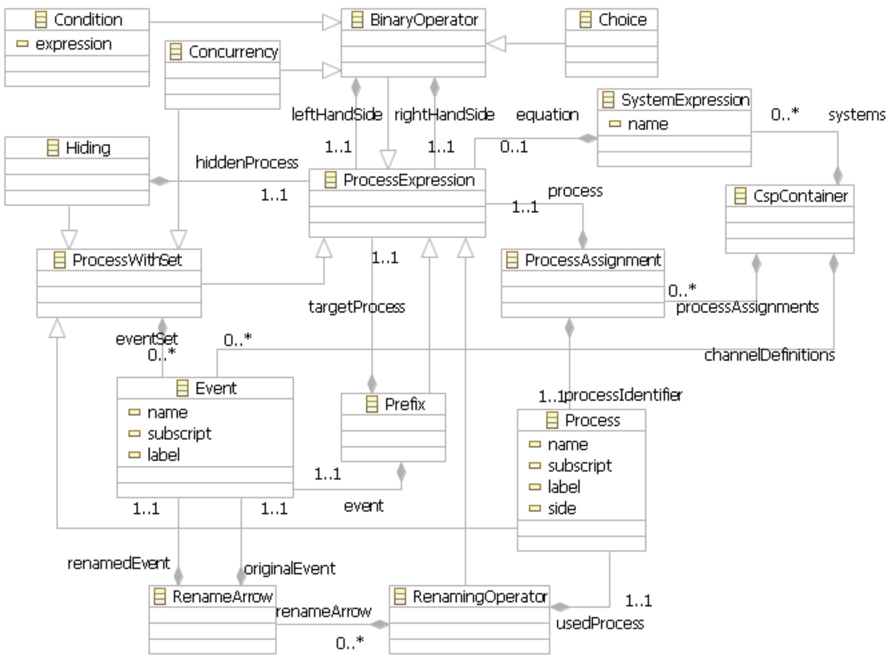


Figure 5.1: Metamodel for CSP

The elementary classes are the Event and Process with obvious meanings. In both classes *name*, *subscript* and *label* attributes denotes a $label.name_{subscript}$ pattern. As CSP is used for refinement checking, identical process names are not allowed. The *side* attribute indicates the particular side of the rule where the process is.

Following the Composite Pattern [3], a process expression either represents a prefix, a process with set of events, renaming or other general binary operator combining two expressions. A process with set is such a process expression that has events associated with it. Hiding has the set of hidden events, concurrency has the set of synchronised events and processes have their own alphabets. The process identifier connected to the process assignment is the declared process while the *process* connection is the definition. CSP container represents the CSP file. Channel definitions and process assignments are defined through the corresponding aggregations. System equations are represented by the *SystemExpression* class with the systems name. The meaning of Concurrency, Prefix, Choice and Hiding is discussed in Section 5.1. The following two examples are presented for clarity.

5.3.1 Concurrency

An example concurrency scenario is shown in Figure 5.2. It shows the expression `P = E [| {|a, b|} |] F`.

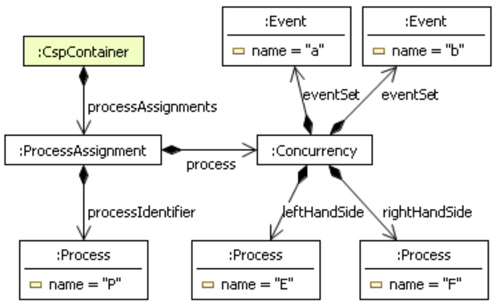


Figure 5.2: Example Concurrency

The ProcessAssignment assigns the concurrency expression to the process P. The Concurrency node has two operands inherited from *BinaryOperator* with the left-hand side process E and the right-hand side F. The synchronised events a and b are contained in the *eventSet* attribute inherited from *ProcessWithSet*.

5.3.2 Renaming

The renaming `R = P [[renamed1<-event1, renamed2<-event2]]` from Section 5.2 is shown in Figure 5.3.

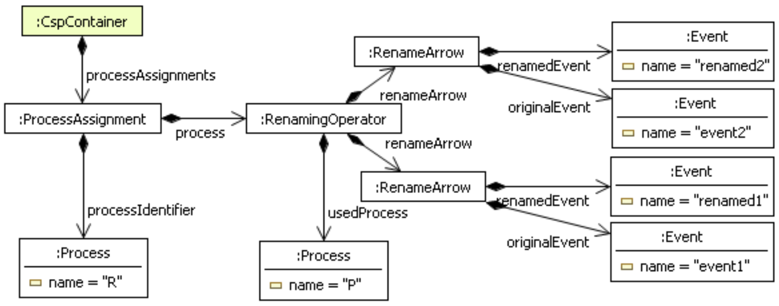


Figure 5.3: Example Renaming

The ProcessAssignment defines the process R as a renaming. The *usedProcess* attribute of RenamingOperator shows the original process, which is P in this case. The connected RenameArrows represent the `<-` arrow of the renaming.

Chapter 6

Semantic Mapping

In order to verify the semantic relation between source and target models, UML models are mapped to CSP processes. The mapping is inspired by triple graph grammar rules [18] but was implemented using the Tiger EMF Transformer [22] tool. It consists of 45 rules organised in 4 major groups (type-level, owned behaviour, instance-level, renaming). The production rules are defined by rule graphs, namely a left-hand side (LHS), a right-hand side (RHS) and possible negative application conditions (NACs). These rule graphs are object-structures that contain objects typed over EMF metamodels of UML diagrams (Fig. 3.4) as well as CSP expressions (Fig. 5.1). These object-structures are also essentially attributed typed graphs.

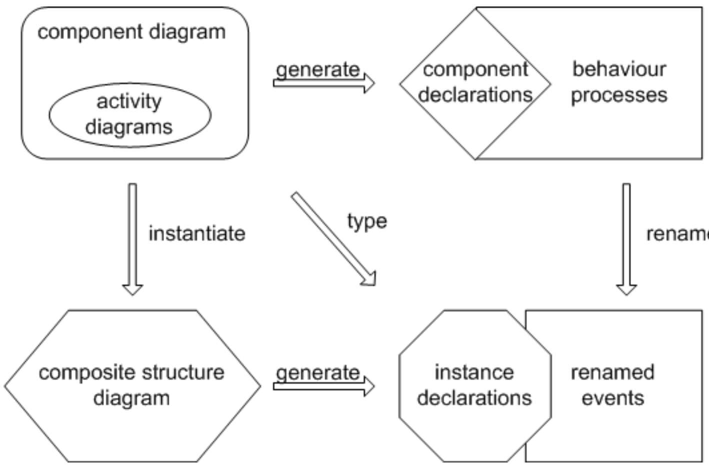


Figure 6.1: Overview of the Transformation

The general mechanics of the transformation is as shown in Figure 6.1. The component declarations derived from the components, port and interfaces form the framework of the semantic model. The behaviour of the system is generated from the corresponding activity diagrams, thus completing the component definitions. The instance level declarations are created using both the component and composite structure diagrams. For every instance, the behaviour is identical to the component. Thus, the behaviour of the component instances are renamed from the type-level behaviour.

The transformation is initiated with rule InitSysEq shown in Figure 6.2. It matches a component container, and creates the corresponding *System-Expression* class. The attribute value *P* denote a variable that holds the same value in both sides of the rule. It can be omitted in the RHS in case of mapped objects.

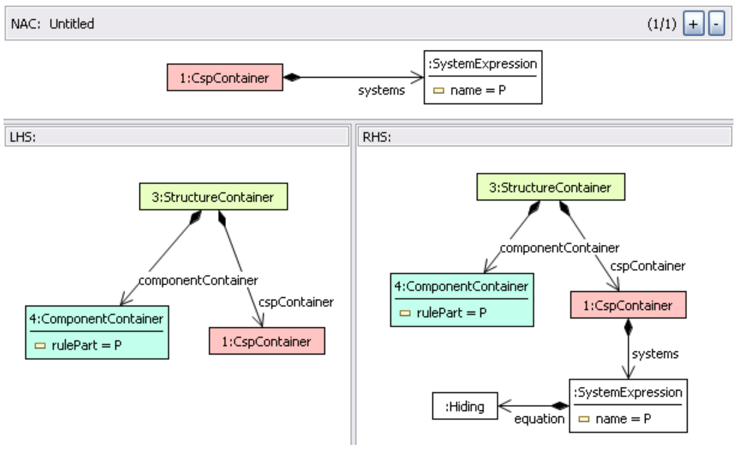


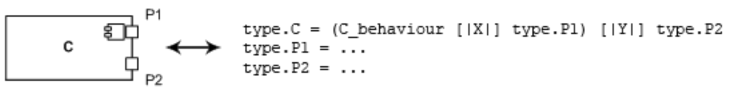
Figure 6.2: Implementation of InitSysEq Rule

6.1 Type-Level Mapping.

The type-level mapping realises the generation of the component declarations.

6.1.1 Components

The mapping of a component and its ports are shown in Fig. 6.3. The component is mapped to a process definition with its owned behaviour (obtained from the activity diagram) and port procsses in parallel. X and Y denotes the shared events between the behaviour and the ports.



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

Figure 6.3: Mapping of a Component and its Ports

The rules in Figure 6.4 and 6.5 create the component and port processes, as well as weaves the port processes to the component definitions.

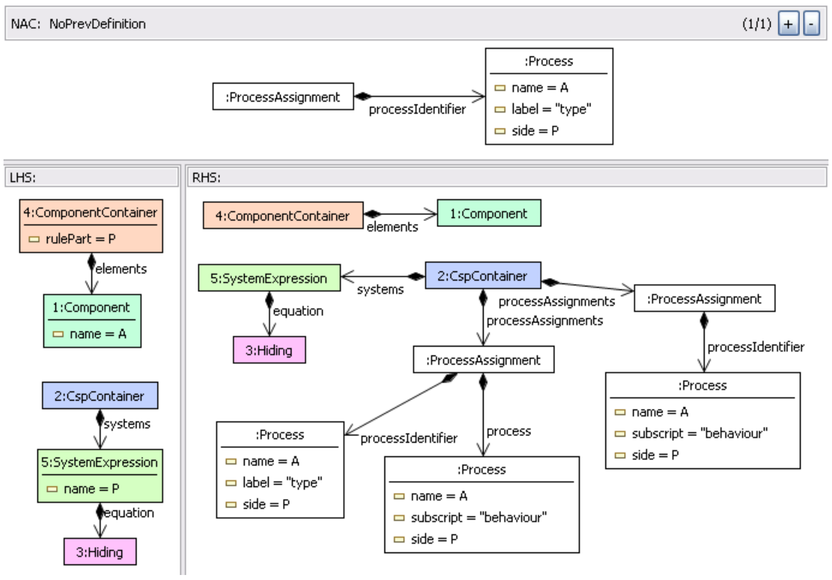


Figure 6.4: Implementation of Component Rule

The *Component* rule in Figure 6.4 creates the process declaration and definition for the corresponding component. The NAC, defined on CSP expressions, checks the existence of a similar process declarations. If none exist, the matched component has not been transformed yet. Thus it creates the CSP expressions `type.C_def = type.C_behaviour` and the empty `type.C_behaviour` process definition. As the absence of an attribute is not matchabe in EMT, the `type` label is used in both the component and port processes, to indicate a type-leve process.

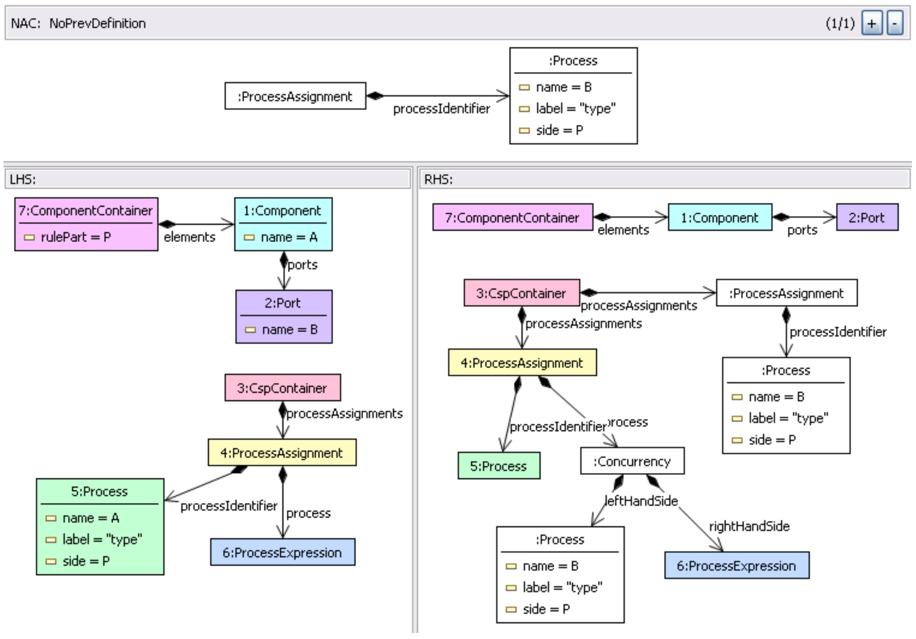


Figure 6.5: Implementation of Port Rule

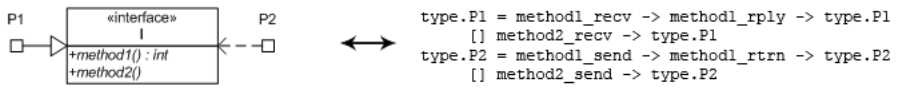
The *Port* rule in Figure 6.5 creates the process declarations for the corresponding ports and inserts them to the definition of the parent component. The NAC works the same way as the one in the *Component* rule. The rule, if applicable creates the empty `type.P1` and `type.P2` process definitions. As the component definition is not empty, the root element can be matched as a general process expression. Hence, the port processes are added to the component definition in two steps:

1. `type.C = type.P1 || C_behaviour`

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

6.1.2 Ports

The ports are mapped to processes with the events corresponding to their interfaces. As shown in Figure 6.6, port `type.P1` implementing provided a interface engage with receive and reply events. In case of required interface with port `type.P2`, the definition consists of the initial send and possible return event.



```
type.P1 = method1_recv -> method1_rply -> type.P1
     [] method2_recv -> type.P1
type.P2 = method1_send -> method1_rtrn -> type.P2
     [] method2_send -> type.P2
```

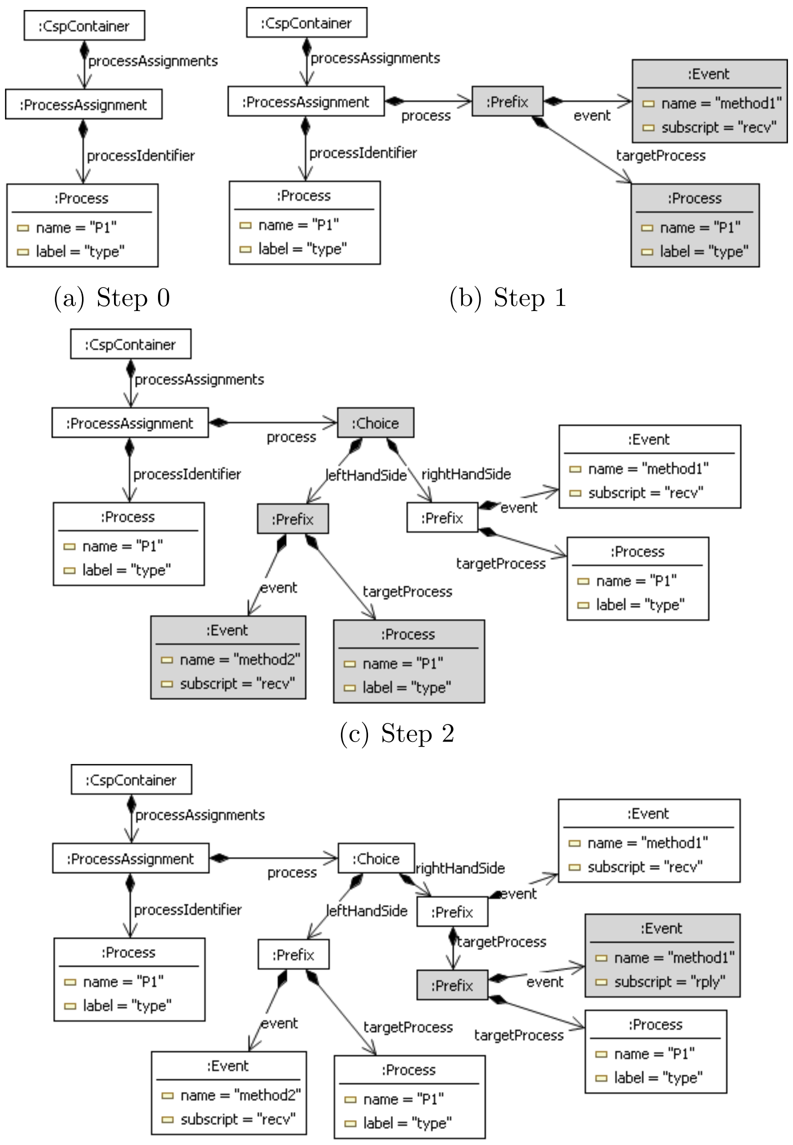
Figure 6.6: Mapping of Port Behaviour

The process of generating the provided interface definition of `type.P1` is shown in Figure 6.7. Step 0 in Figure 6.7(a) is the initial declaration of the port process. When an accept event action is found with the identical method name in the corresponding interface a definition is created for the empty process declaration as shown in Figure 6.7(b). This definition is the prefix `method1_recv -> type.P1`. When a communication event corrresponding to a different method is found, a choice is introduced in Figure 6.7(c). The rule in Step 1 searched for an empty process definition, this rule needs a generic process expression to insert the choice operator. When a send signal action is present in the owned behaviour besides an accept event action, the system sends a reply. Az shown in Figure 6.7(d), the necessary prefix `method1_rply -> type.P1` is inserted. Although not shown in Figure 6.7, the created events are added to the corresponding event set of the concurrency operator in the component and channel definition.

6.1.3 Interfaces

In the CSP representation ports are facades synchronising the events between communication channels and owned behaviour. Interfaces are themselves the communication behaviour. They contain all the possible communication events, and through event synchronisation they define the allowed order of events.

The mechanics behind the rules building the interface definitions are similar to the ones wiht ports. The only difference is that the events are created in $send - receive$ and $reply - return$ pairs if necessary.

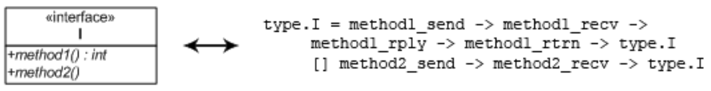


(a) Step 0 (b) Step 1

(c) Step 2

(d) Step 3

Figure 6.7: Mechanics of the provided interface transformation



```
type.I = method1_send -> method1_recv ->
      method1_rply -> method1_rtrn -> type.I
      [] method2_send -> method2_recv -> type.I
```

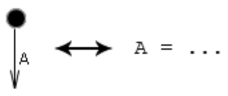
Figure 6.8: Mapping of an Interface

6.2 Behavioural Mapping

Every component has exactly one activity diagram as its owned behaviour. This activity diagram is transformed to CSP to define the behaviour of the component. This behavioural mapping was first presented in [2]. The idea behind the mapping is to relate an Edge in the owned behaviour to a Process in CSP.

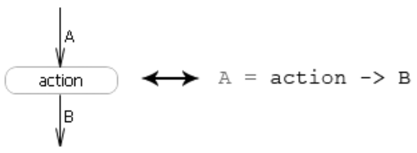
6.2.1 Initial Node, Action and Final Node

First, we consider the initial node. Although this node is not mapped to anything directly, its outgoing edge is related to a process declaration of the same name. This will be the first process.



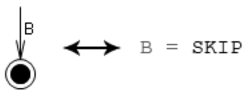
```
A = ...
```

A previously declared process A is defined in terms of a new prefix expression, and the process B is declared from the outgoing activity edge.



```
A = action -> B
```

The definition of an activity edge that ends in a final node is the `SKIP` process



```
B = SKIP
```

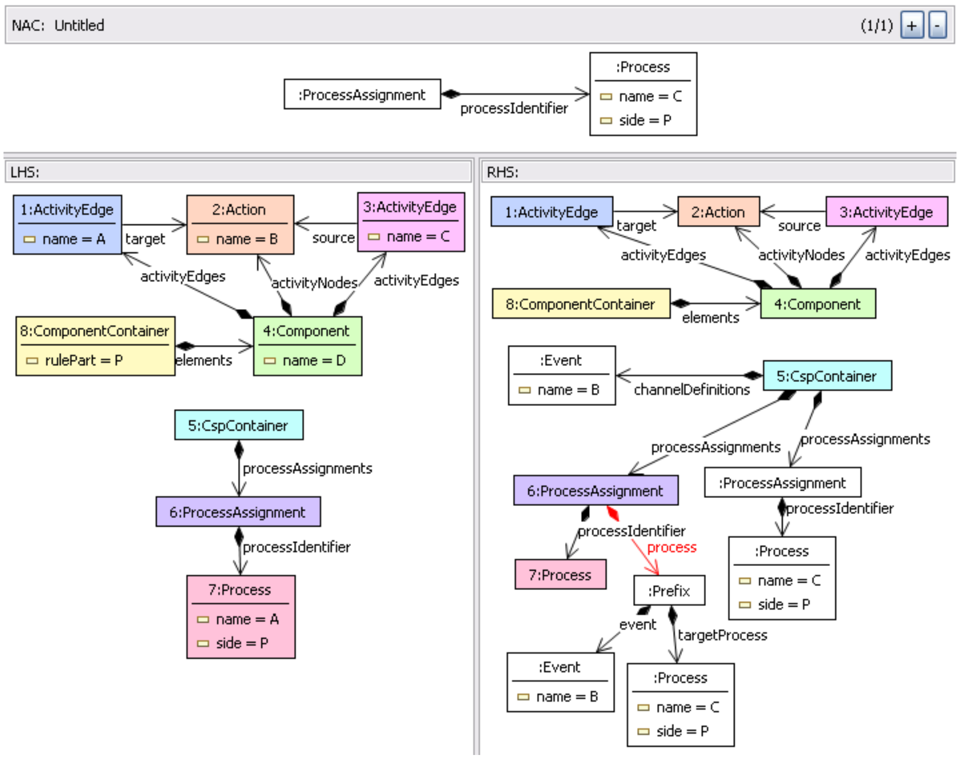


Figure 6.9: BhAction rule

As the most important rule from this group, the *BhAction* rule is presented in Figure 6.9 that implements the mapping of an action.

In the LHS the Action, its incoming and outgoing edges are matched along with the process declaration corresponding to the incoming edge. In the RHS, a prefix is created as well as the empty process declaration for the outgoing edge. The created event is added to the channel definition.

6.2.2 Communication Events

Both the send signal and accept event action maps to a similar prefix as an action node. As mentioned in Section 3.3, these events represent function calls through the ports they are engaged in. Thus, the difference is the distinction of the four communication primitives. Send and reply for the send signal action; receive an return for the accept event action. Both actions have two possible meanings, thus the transformation uses two rules for each.

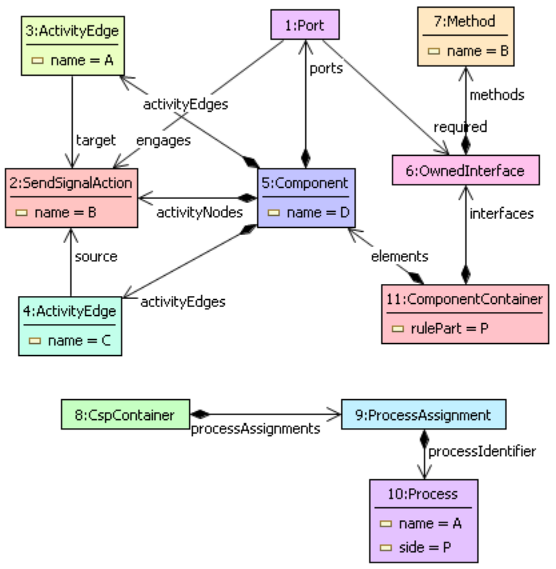


Figure 6.10: LHS of SendSignalAction Rule 1

The LHS of one of the rules transforming send signal action is shown in Figure 6.10. The pattern is similar to the one in Figure 6.9: the send signal action, its incoming and outgoing edges. However, the corresponding port is also matched. The corresponding port has to be connected to an interface with a method of a similar name as the send signal action. If the interace 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.

6.2.3 Decision Node and Merge Node

The transformation of a DecisionNode depicted in Figure 6.12 is a more complicated case. The concrete syntax is obvious, but the choice is a binary operator. Thus, we have to build a binary tree bottom-up as depicted in Figure 6.11. First *else* branch is matched with an arbitrary edge and create the lowest element of the tree (in dark shade grey). Then the tree is built by adding the elements one-by-one (in light shade grey).

Note that this transformation, which creates non-determinism at the syntactic level, leads to semantically equivalent processes. According to [17], *the*

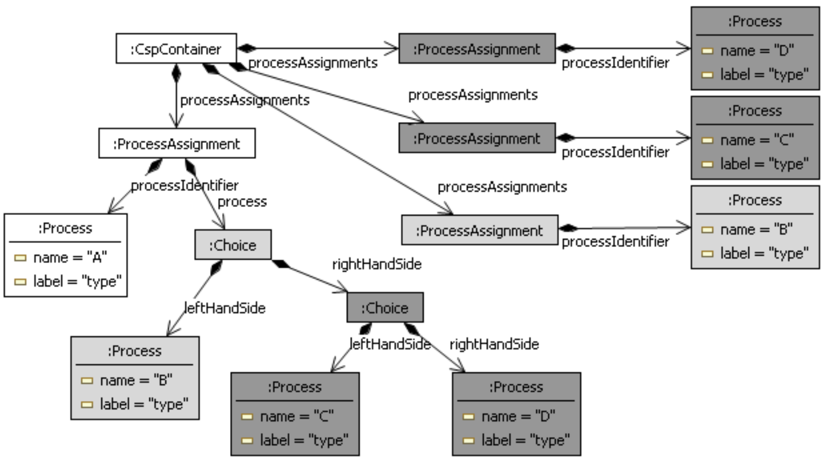


Figure 6.11: Abstract syntax tree for the result of DecisionNode transformation

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, if guard conditions are disjoint, syntactically different nestings are semantically equivalent.

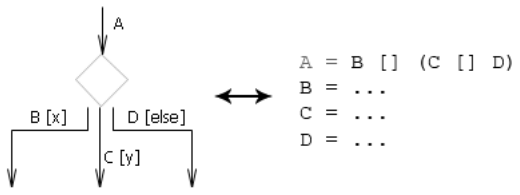


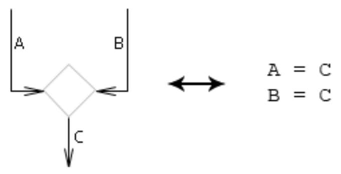
Figure 6.12: DecisionNode

The MergeNode is a simpler case, as illustrated in Figure 6.13. It is mapped to an equation identifying the processes corresponding to the two incoming edges.

6.2.4 Fork Node and Join Node

Fork node and join node represent semantically the most complex cases. Before describing the transformation, we discuss some observations.

If in an activity diagram the names of the action nodes are unique, the

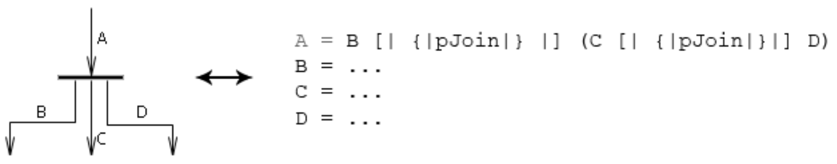


```
A = C
B = C
```

Figure 6.13: MergeNode

intersection of the alphabets of the corresponding processes is empty. This would cause problems in official CSP, where the shared events are automatically synchronised, and the processes may get stuck while waiting for some random other process that accidentally has events with similarly names. As mentioned in Section 5.2, the synchronisation points are explicitly defined, and all other events interleave. We need synchronisation points in order to implement the joining of processes. Thus we use an event `pJoin` as a synchronisation point. This way all participating processes require simultaneous participation. This fact is used to join concurrent processes by blocking them until they can perform the synchronisation event.

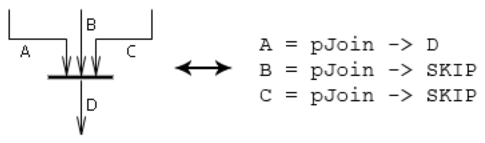
The mapping for the fork node is shown in Figure 6.14. The concurrency operator is binary, so by generating the nodes one-by-one, we create a binary abstract syntax tree of concurrency nodes the same way we did in Figure 6.11 for the decision node. Since $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$, the different trees are semantically equivalent.



```
A = B [| {|pJoin|} |] (C [| {|pJoin|}|] D)
B = ...
C = ...
D = ...
```

Figure 6.14: ForkNode

The transformation of a join node is depicted in Figure 6.15. The first edge that meets the join node is chosen to carry the continuation process, while the others terminate in a $SKIP$. All processes engange with the `pJoin` event before continuing.



```
A = pJoin -> D
B = pJoin -> SKIP
C = pJoin -> SKIP
```

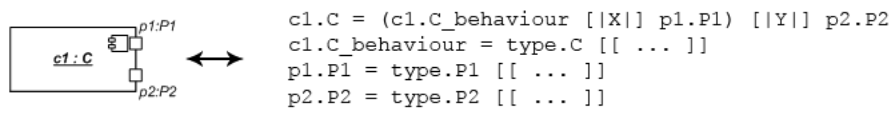
Figure 6.15: Join Node

6.3 Instance-Level Mapping.

The composite structure diagram models the dynamic behaviour of a component system. Hence, it needs to be used for checking of behavioural refinement. The instance-level mapping first creates the process declarations. The behaviour is renamed from the type-level as described in Section 6.4.

6.3.1 Component Objects

To deal with multiple instances, component and port instance processes are renamed according to their instance names as shown in Figure 6.16.



```
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 6.16: Mapping of a Component Instance

Aside from the generated renaming definition and instance name labels, the rules creating the process declarations are similar to component and port rules shown in Figures 6.4 and 6.5.

6.3.2 Channels

Channels are implementations of interface definitions. The channel object maps to a process declaration as shown in Figure 6.17, since its behaviour is renamed from the corresponding interface.

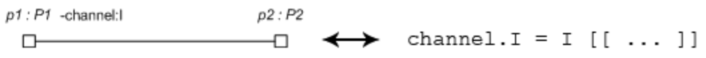


Figure 6.17: Mapping of a Channel

6.4 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. This instantiation is done by renaming the events. The structural elements are mapped to processes, thus the distinction between the RHS and LHS was important. On the contrary, similar events have to bear similar name.

As shown in Section 6.3, 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. They are considered to be deleted or created. However, event instances that are similar on both rule sides need a distinction. As events are possibly relocated in the hierarchy, their renaming cannot be based on structural notions. These event instances, are renamed by a unique *mapid* overriding the structure based renaming.

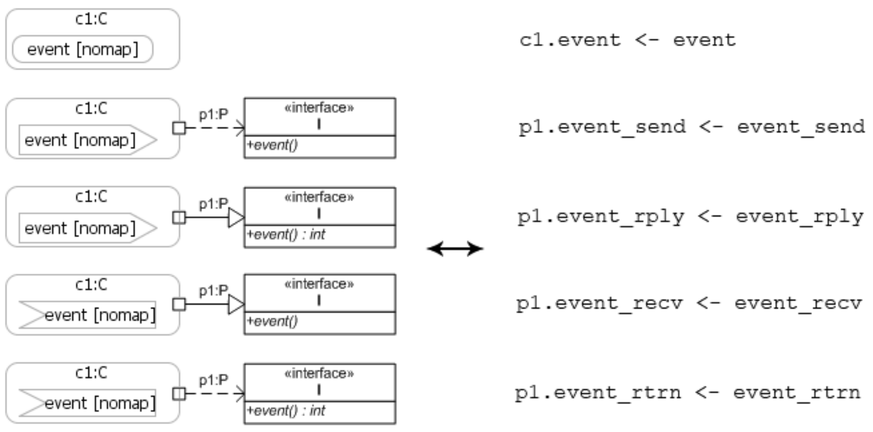

c1:C
event [nomap]
c1.event <- event
p1:P
«interface»
I
+event()
p1.event_send <- event_send
+event() : int
p1.event_rply <- event_rply
p1.event_recv <- event_recv
p1.event_rtrn <- event_rtrn

Figure 6.18: Renaming of Non-Mapped ActionNodes

The three object from the architectural model that maps to events are the action, send signal action and accept event action. As the two communication actions can indicate two different communication primitives, this resolves to five different cases as shown in Figure 6.18 for the non-mapped actions only.

As the simplest of all, the renaming of a non-mapped action is shown in Figure 6.19. 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 CSP expression, where the original event is renamed with a label bearing the name of the relevant 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 6.5. The renaming of the communication nodes are works the same way, they only encompass a more complicated pattern in the LHS.

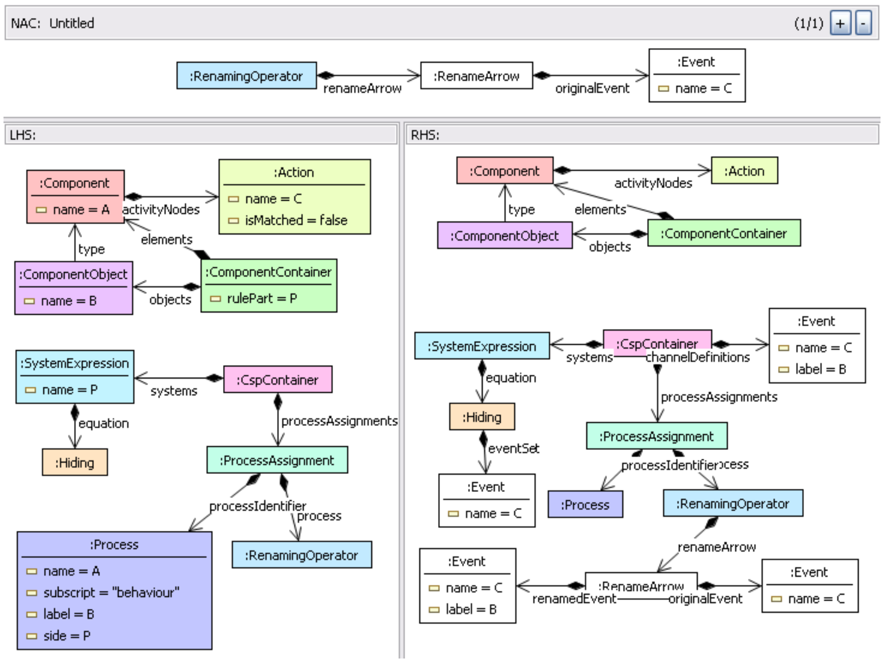


Figure 6.19: RnActionNoMap rule

The rules for the mapped elements are slightly different. The $isMapped$ attribute in their case is $true$ with an integer $MapId$ present, that holds their system-wide unique identification. All mapped elements, regardless of the structural status, are labelled with the map id.

6.5 Application to the Rule

To verify the compatibility of the rule with a semantic relation, say trace refinement, we map the instance levels of both left- and right-hand side to their semantic representation and verify the relation between them. For the left-hand side, for example the refactoring rule from Figure 4.4, this yields

```
System_LHS = (((a [|X|] ch-n) [|Y|] c.C) [|Z|] ch-j) [|W|] b \
{|unmapped1, unmapped2, ... |}
```

by placing all component instances and connectors in parallel and hiding the unmapped events. As the set of events `X`, `Y`, `Z` and `W` are the communication events, whenever a *send* event happens at the component, the channel attached changes state to and waits for the corresponding *recv* event at the other end.

On the right hand side we hide all internal communication between instances of $C1$ and $C2$. For example, referring to our activity diagram in Fig. 4.3(a), the *alertData* and *callStarted* events are hidden because they serve the combination between the two parts of the newly split component C. To check if $sem(L) \sqsubseteq sem(R)$ we would take into account the CSP mappings of all activity diagrams of components involved in the transformation.

The assertion of $System_{RHS} \sqsubseteq_T System_{LHS}$ successful in FDR2, and indicates trace refinement.

Chapter 7

Correctness of Rule-level Verification

In this chapter we demonstrate that the method of verifying a transformation by verifying an extracted rule is indeed correct. The crucial condition is the compositionality of the semantic mapping, which guarantees that the semantic relation $\mathcal{R}$ (think refinement or equivalence) is preserved under embedding of models. We will first formulate the principle and prove that, assuming this property, our verification method is sound. Then we establish a general criterion for compositionality and justify why this applies to our semantic mapping.

7.1 Correctness

The overall structure is illustrated in Fig. 7.1. The original model (component, composite structure and activity diagrams) is given by graph G. The refactoring results in graph H by the application of rule $p : L \to R$ at match m. Applying the semantic mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ to the rule's left- and right-hand side, we obtain the semantic expressions $sem(L)$ and $sem(R)$. Whenever the relation $sem(L)\ \mathcal{R}\ sem(R)$ (say $\mathcal{R}\ = \sqsubseteq$ is for example CSP trace refinement, so all traces of the left processes are also traces of the right), we would like to be sure that also $sem(G)\ \mathcal{R}\ sem(H)$

$$\begin{tikzcd}
L \arrow[rr] \arrow[d, mapsto, "m"'] \arrow[ddd, mapsto, bend right=40, "sem"'] & & R \arrow[d, mapsto, "m*"] \arrow[ddd, mapsto, bend left=40, "sem"] \\
G \arrow[rr] \arrow[d, mapsto, "sem"'] & & H \arrow[d, mapsto, "sem"] \\
sem(G) \arrow[r, dotted, no head] & \mathcal{R} \arrow[r, dotted, no head] & sem(H) \\
sem(L) \arrow[u] \arrow[r, dotted, no head] & \mathcal{R} \arrow[u, Rightarrow] \arrow[r, dotted, no head] & sem(R) \arrow[u]
\end{tikzcd}$$

Figure 7.1: Semantic correspondence for behaviour verification

(traces of $sem(G)$ are preserved in $sem(H)$).

Both the source and target models are represented as typed graphs. For clarity, we present the definition of typed graphs that we use.

Definition 7.1.1 ***(typed graph and typed graph morphism** [4]**)*** *A type graph is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} and E_{TG} are called vertex and edge label alphabet respectively. Then a tuple $(G, type)$ of a graph G together with a graph morphism $type : G \to TG$ is called a typed graph.*

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 $type_2 \circ f = type_1$.

$$\begin{tikzcd}
G_1 \arrow[r, "f"] \arrow[d, "type_1"'] \arrow[dr, phantom, "="] & G_2 \arrow[dl, "type_2"] \\
TG &
\end{tikzcd}$$

As the concept of context is central in compositionality, a context C in graph G is the set of surrounding nodes and edges around an arbitrary subgraph D in G, i.e. $C = G \setminus D$. Context C is not necessarily a graph due to the possible dangling edges.

The main assumption is the compositionality of the semantic mapping *sem*. Intuitively, it is similar to the compositionality property of denotational semantics. As for simple mathematical expressions, we assume that the meaning of expression $2+5$ is determined by the meaning of 2, 5 and the semantics of the + operator, i.e. $[[2+5]] = [[2]] \bigoplus [[5]]$.

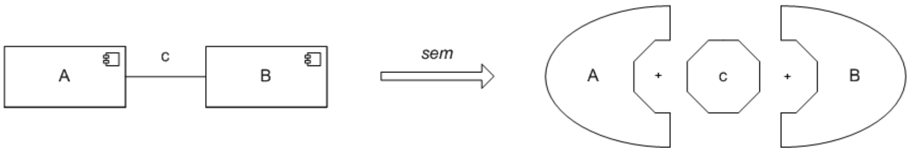


Figure 7.2: Compositinal Semantic Mapping

In terms of model transformations, compositionality is presented in Figure 7.2. A system consisting of components A and B with a connector c is mapped to a semantic domain through transformation sem. The result is such a set of semantic expressions where $sem(A)$, $sem(B)$ and $sem(c)$ are distinguishable and their composition represents the semantics of the whole system.

Definition 7.1.2 ***(compositionality)*** *A mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$ is compositional if for each injective graph morphisms $m : L \to G$ there exists a context E such that $sem(G) \equiv E[sem(L)]$. Moreover, this context is uniquely determined by the part of G not in the image of L, i.e., given a pushout diagram as below with injective morphisms only, and a context F with $sem(D) \equiv F[sem(K)]$, then E and F are equivalent.*

$$\begin{tikzcd} K \arrow[r, "l"] \arrow[d, "d"] & L \arrow[d, "m"] \\ D \arrow[r, "g"] & G \end{tikzcd}$$

The concept of compositionality is depicted in Figure 7.3. The semantic expression generated from G contains the one derived from L (through the inclusion morphism m). Also it is uniquely determined by the part of G not in the image of L.

Definition 7.1.2 applies particularly where L is the left hand side of a rule and G is the given graph of a transformation. In this case, the semantic expressions generated from L contains the one derived from G up to equivalence, while the context E is uniquely determined by $G \setminus m(L)$.

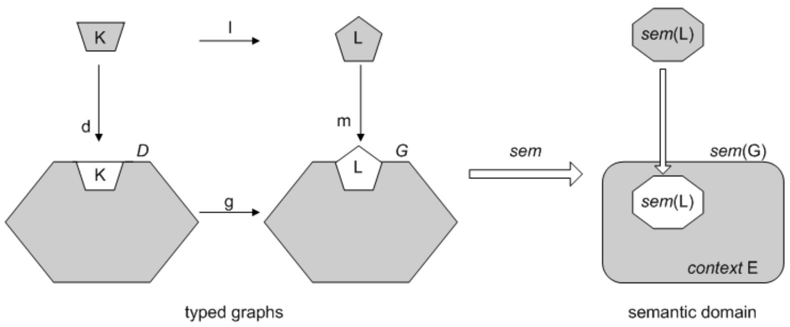


Figure 7.3: Unique determination of the context

Theorem 7.1.1 *Assume a compositional mapping sem* $: Graphs_{TG} \to (D, \sqsubseteq, \mathcal{C})$. *Then, for all transformations* $G \overset{p,m}{\Longrightarrow} H$ *via rule* $p : L \to R$ *with injective match* m, *it holds that* $sem(L) \sqsubseteq sem(R)$ *implies* $sem(G) \sqsubseteq sem(H)$.

Proof 1 *By assumption the match* m, *and therefore the comatch* $m^* : R \to H$ *are injective. Since the mapping sem is compositional, according to Definition 7.1.2 there are contexts* E *and* F *such that* $sem(G) \equiv E[sem(L)]$ *and* $sem(H) \equiv F[sem(R)]$. *Now,* $E[sem(L)] \sqsubseteq E[sem(R)]$ *since* $sem(L) \sqsubseteq sem(R)$ *and* $\sqsubseteq$ *is closed under context. Finally,* $E[sem(R)] \equiv F[sem(R)]$ *by the uniqueness of the contexts.*

The statements in Theorem 7.1.1 hold for the relation $\equiv$, being the symmetric closure of $\sqsubseteq$.

Chapter 8

Compositionality.

In this section we present a proof sketch for the fact that our semantic mapping is compositional. The result is interesting by itself because it holds for a large class of mappings described by triple graph grammars [18]. The idea is that triple graph grammars describe model transformations by creating the target from the source model and linking both by a relation model. Hence it is not necessary to remove the source model, and rules can be designed in such a way that also on the relation and target model rules are non-deleting.

For simple rules (without negative application conditions), compositionality then follows directly from the fact that the transformations realising the semantic mapping can be embedded into larger contexts.

8.1 Simple Graph Transformations

In this section we give a condition for compositionality for semantic mappings specified by simple graph transformations.

The semantic mapping *sem* is defined by a typed graph transformation system $GTS = (TG, P)$ consisting of a type graph TG and a set of typed graph productions P.

The result of the semantic mapping *sem* on a source graph G_0 is $sem(G_0) = G_n$ if and only if it is a typed graph transformation $G_0 \overset{p_1}{\Rightarrow} G_1 ... G_{n-1} \overset{p_n}{\Rightarrow} G_n$ with rules $p_1, ..., p_n \in P$ and it is terminating ($\nexists p \in P$ that can be applied to G_n).

It is important to note that only a globally deterministic graph transformation produces a unique result for a source graph, regardless of the rule application order. A graph transformation is globally deterministic only if it is *confluent* or *locally confluent* and *terminating* [4]. As termination was already required for *sem*, it has to be locally confluent, otherwise it is not well-defined.

Theorem 8.1.1 *Assume a mapping* $sem : G_0 \overset{*}{\Rightarrow} G_n$ *from typed graphs to semantic domain* $(D, \sqsubseteq, \mathcal{C})$ *described by a graph transformation system* GTS*. If all rules of* GTS *are non-deleting and do not contain negative application conditions, then sem is compositional.*

Proof 2 *Given the mapping* $sem : G_0 \overset{*}{\Rightarrow} G_n$ *from graphs to semantic domain* $(D, \sqsubseteq, \mathcal{C})$ *by* $sem(G_0) = G_n$ *and a pushout* (1) *with injective morphisms* m_0, n_0.

$$\begin{tikzcd}
G_0 \arrow[r] \arrow[d, "m_0"'] \arrow[dr, phantom, "(1)"] & G'_0 \arrow[d, "n_0"] \\
H_0 \arrow[r] & H'_0
\end{tikzcd}$$

If sem is compositional, then exists a context E *such that* $sem(H'_0) \equiv E[sem(G'_0)]$ *and* $sem(H_0) \equiv E[sem(G_0)]$.

The main argument is based on the Embedding Theorem [4].

$$\begin{tikzcd}
B \arrow[r, "b_0"] \arrow[d] \arrow[dr, phantom, "(2)"] & G_0 \arrow[r, Rightarrow, "sem"] \arrow[d, "m_0"] \arrow[dr, phantom, "(3)"] & G_n \arrow[d, "m_n"] \\
C \arrow[r] & H_0 \arrow[r, Rightarrow, "sem"] & H_n
\end{tikzcd}$$

For the transformation $sem : G_0 \overset{*}{\Rightarrow} G_n$ *we create a boundary graph* B *and a 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 *[4] of the morphism* $m_0 : G_0 \to H_0$.

If none of the productions of sem deletes any item of B*, then* m_0 *is consistent with sem and there is an* extension diagram *over sem and* m_0 *[4]. This basically means that* H_n *is the pushout complement of sem and* m_0*, thus* *can be determined without applying the transformation sem on H_0. By the assumption that all rules of sem are non-deleting, sem is consistent with m_0. Thus, according to the* Embedding Theorem*, we have an* extension diagram*, as depicted below, where* (2) *and* (2′) *are initial pushouts and the following equations hold.*

$$H_n \cong G_n +_B E \;=\; E[G_n] \implies H_n \equiv E[G_n] \tag{8.1}$$

$$H'_n \cong G'_n +_B E \;=\; E[G'_n] \implies H'_n \equiv E[G'_n] \tag{8.2}$$

$$\begin{tikzcd}
 & & G'_0 \arrow[rr, Rightarrow, "sem"] \arrow[dd, "n_0"] & & G'_n \arrow[dd, "n_n"] \\
B \arrow[r] \arrow[urr] \arrow[dd] \arrow[ddr, phantom, "(2)"] & G_0 \arrow[rr, Rightarrow, "sem"] \arrow[ur] \arrow[dd, "m_0"] \arrow[dr, phantom, "(2')"] & & G_n \arrow[dd, "m_n"] & \\
 & & H'_0 \arrow[rr, Rightarrow, "sem"] & & H'_n \\
E \arrow[r] \arrow[urr] & H_0 \arrow[rr, Rightarrow, "sem"] \arrow[ur] & & H_n &
\end{tikzcd}$$

We have $sem(G_0) = G_n$, $sem(H_0) = H_n$ and $sem(G'_0) = G'_n$, $sem(H'_0) = H'_n$.

Thus the following equations hold.

$$sem(H'_0) = H'_n \equiv E[G'_n] \;=\; E[sem(G'_0)] \tag{8.3}$$

$$sem(H_0) = H_n \equiv E[G_n] \;=\; E[sem(G_0)] \tag{8.4}$$

8.2 Graph Transformations with NACs

In Section 8.1 the compositionality of graph transformations with non-deleting rules was proved. However in order to control the transformation, negative application conditions (NACs) are used. In Theorem 8.2.1 we show, that NACs do not disturb the compositionality, thus a non-deleting transformation with NACs remain compositional.

Definition 8.2.1 ***(negative application condition** [4])* *A simple negative application condition is of the form $NAC(x)$, where $x : L \to X$ is a* *morphism. A morphism $m : L \to G$ satisfies $NAC(x)$ if there does not exist a morphism $p : X \to G$ in M' with $p \circ x = m$:*

$$\begin{tikzcd}L \arrow[r, "x"] \arrow[d, "m"] & X \arrow[dl, dotted, "p"] \\ G & \end{tikzcd}$$

Before the establishment of Theorem 8.2.1, the necessary definitions are presented.

Definition 8.2.2 ***(gluing and created points)*** *Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$.*

- *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 created points CP are those nodes and edges in R that are created by p, i.e. $CP = r_V(V_K)\backslash V_K \cup r_E(E_K)\backslash E_K$.*

The concept of created points is demonstrated in Figure 8.1.

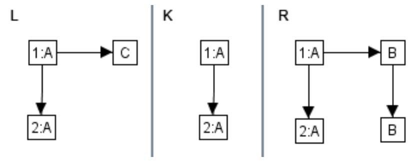


Figure 8.1: Production Rule with Created Points

Since the C node is deleted, the only gluing points are the two A nodes. They are not deleted by the rule in Fig. 8.1. The created points are the B nodes on the right hand side of the rule.

It is possible that the B nodes are always - if present - created or gluing points in every production rule of a graph transformation system. This means that the node type B is such a special type that none of its instances are deleted. This observation leads to the definition of a *stable type*.

Definition 8.2.3 ***(stable types)*** *Assume a typed graph transformation system $GTS = (TG, P)$.*

Stable types $ST \subseteq TG = (V_{ST}, E_{ST})$ are those nodes and edges in the type graph TG, whose instances are not deleted by any production $p \in P$. i.e.

$ST = \{v \in V_{TG} \mid \forall p \in P : v = type_V(w) \wedge w \in (CP_{p_i} \cup GP_{p_i})\} \ \cup \ \{e \in E_{TG} \mid \forall p \in P : e = type_E(f) \wedge f \in (CP_{p_i} \cup GP_{p_i})\}$.

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

Although NACs used to be connected to the LHS, thus forming a precondition, NACs can be defined as a postcondition and connected to RHS of the production rule. The left NACs are equivalent to the right NACs as they can be constructed from them [12]. The left equivalent of a NAC_R is denoted as $L_p(NAC_R)$.

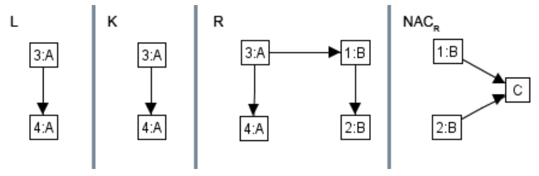


Figure 8.2: Right Negative Application Condition

As stated in Definition 8.2.1, a NAC is true, thus enables the application of the production rule, if the pattern *cannot* be found in the host graph. However, as shown in Figure 8.2, a right NAC can form a restriction on created points. If the NAC_R is different from the RHS (as in that case it disables the application of the rule altogether) it is always true. As shown in Figure 8.2 it is simply impossible to find a connected C node to the B nodes that has just been created. The created points are in fact just came to being, and thus the host graph will never contain the disabling pattern.

Corollary 8.2.0.1 *Assume a rule $p : L \xleftarrow{l} K \xrightarrow{r} R$ with negative application condition $NAC(n_i)$ where $n_i : L \rightarrow N_i$ that has only stable points with at least one created point in $N_i \setminus n_i \circ r(K)$. According to the construction of left from right application conditions, the pushout complement (1) of $(K \rightarrow Z, Z \rightarrow N_i)$ does not exists. Thus, the $n_i' = id$ and $NAC(n_i')$ is satisfied by arbitrary match in Q.*

$$\begin{tikzcd}
L \arrow[d, "n'_i"'] & K \arrow[l] \arrow[r] \arrow[d] \arrow[rd, phantom, "(1)"] & R \arrow[d, "n_i"] \\
N'_i & Z \arrow[l] \arrow[r] & N_i
\end{tikzcd}$$

Theorem 8.2.1 *Assume a mapping sem* $: G_0 \overset{*}{\Rightarrow} G_n$ *from typed graphs to semantic domain* $(D, \sqsubseteq, \mathcal{C})$ *described by a graph transformation system* $GTS = (TG, P)$ *with stable types* $ST \subseteq TG$. *The mapping sem is compositional if:*

1. $\forall p \in P$ *is non-deleting*
2. $\forall NAC(n), n : L \to N$ *for* $p \in P : L \overset{l}{\leftarrow} K \overset{r}{\to} R$ *contains only stable points in* $N \setminus n \circ r(K)$.
3. G_0 *does not contain any stable points.*

Proof 3 *If the extension diagram exists for a non-deleting transformation, Theorem 8.1.1 showed it to be compositional. To prove compositionality for a transformation with NACs, we have to prove the existence of the extension diagram, and then compositionality follows from Theorem 8.1.1. As the equivalent left NACs can be constructed from the right NACs, 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 [11], but also NAC-consistent [12]. According to the synthesis construction of Concurrency Theorem *a concurrent rule* p_c *with a concurrent match* g_c *exists [4]. The concurrent rule* p_c *is basically the merge of all rules for 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} *the concurrent NAC,* g_c *the concurrent match induced by* t *and* $k_0 : G_0 \to H_0$ *the inclusion morphism [12].*

The main idea of the proof is, according to Corollary 8.2.0.1, if a NAC_R contains created points, the corresponding NAC_L is always true. We distinguish two cases:

Case 1: *There are* no *production rules $p_i, p_{i+1} \in P$ such that NAC_{i+1} contains only elements that are gluing points in p_i. The proof is by mathematical induction over the direct transformation steps n.*

Basis. $n = 1$. In the beginning of the transformation it is possible to have several transformation rules that do not contain stable elements at all. Since the NACs have restrictions only on the stable elements (assumption 2), and G_0 does not contain any stable element (assumption 3), the NACs won't apply enabling the application of the rules. This case is similar to the one where there are no NACs at all.

Thus we have to consider $n = 1$ when the direct transformation $t_0 : G_0 \overset{p_0, m_0}{\Longrightarrow} G_1$ is applied via rule $p_0 : L_0 \leftarrow K_0 \rightarrow R_0$ and NAC_{p_0}.

The concurrent rule p_c and concurrent NAC_{p_c} induced by $G_0 \Rightarrow G_1$ is defined by $p_c = p_0$ and $NAC_{p_c} = NAC_{p_0}$. Since the NAC_{p_c} contains only stable elements, the morphism k_0 is NAC-consistent. Thus, the extension diagram exists over t_0 and k_0.

$$\begin{tikzcd}
N_i \arrow[d, dotted, "e_i"'] & & & & & N_j \arrow[dl, dotted, "z_1"'] & & \\
N'_i \arrow[r, phantom, "(1)"] & L'_c \arrow[ul, "n_i"'] \arrow[d, dotted, "m_c"] \arrow[dddd, bend right, "g'_c"'] & K'_c \arrow[l] \arrow[r] \arrow[d, dotted] & R'_c \arrow[dr, dotted, "e'_c"] \arrow[dddr, bend left, "h'_c"'] & N'_j \arrow[r, phantom, "(3)"] & L_n \arrow[u, "n_j"'] \arrow[dl, dotted, "e_n"] \arrow[dddl, bend left, "g_n"] & K_n \arrow[l] \arrow[r] \arrow[d, dotted] & R_n \arrow[d, dotted] \arrow[dddd, bend left, "g_{n+1}"] \\
& L_c \arrow[ul, dotted, "n'_i"'] \arrow[ddd, dotted, "g_c"] & C_c \arrow[l, dotted, "l"'] \arrow[rr, dotted] \arrow[ddd, dotted] & & E \arrow[u, dotted, "n'_j"] \arrow[ddd, dotted, "h"] & & C_n \arrow[ll, dotted] \arrow[r, dotted, "r"] \arrow[ddd, dotted] & R_c \arrow[ddd, dotted, "h_c"] \\
& & & K_c \arrow[ul, dotted, "k_c"] \arrow[urrr, dotted, "k_n"] \arrow[ul, phantom, "(2)"'] & & & & \\
& & & & & & & \\
& G_0 & D_n \arrow[l] \arrow[rr] & & G_n & & D \arrow[ll] \arrow[r] & G_{n+1}
\end{tikzcd}$$

Induction Step. Consider $t_n : G_0 \overset{n}{\Rightarrow} G_n \Rightarrow G_{n+1}$ via the rules $p_0, p_1, ..., p_n$. The concurrent rule for $p_0, p_1, ..., p_{n-1}$ is $p'_c = L'_c \leftarrow K'_c \rightarrow R'_c$ as shown in *the diagram above.*

To prove the NAC-consistency, a deeper analysis of the concurrent NAC_{p_c} *construction is necessary. The concurrent* NAC_{p_c} *consists of two parts:* $D_{m_c}(NAC_{L'_c})$ *and* $DL_{p_c}(NAC(n_j))$.

$D_{m_c}(NAC_{L'_c})$ *is created along pushout* (1)*. It is the resultant NAC from the first rule* p'_c*, such that* $g'_c \models NAC_{L'_c} \Leftrightarrow g_c \models D_{m_c}(NAC_{L'_c})$ *[12].*

The other resultant NAC, $DL_{p_c}(NAC(n_j))$*, is transformed from the* $NAC(n_j)$ *of second rule* p_n*. In fact* $NAC(n'_j)$ *is a postcondition for the production rule* $L_c \leftarrow C_c \rightarrow E$.

$$\begin{tikzcd}
N''_j & Z \arrow[l] \arrow[r] & N'_j \\
L_c \arrow[u, "n''_j"] \arrow[ur, phantom, "(5)"] & C_c \arrow[l] \arrow[r] \arrow[u] \arrow[ur, phantom, "(4)"] & E \arrow[u, "n'_j"']
\end{tikzcd}$$

As defined in construction of NACs on L_c *from NACs on* L_1, $DL_{p_c}(NAC_{n_j}) = L_p(D_{e_1}(NAC(n_j)))$ *such that* $g_n \models NAC_{L_n} \Leftrightarrow g_c \models DL_{p_c}(NAC_{L_n})$ *[12].*

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* $g_c : L_c \rightarrow G_0$*, comatch* $h_c : R_c \rightarrow G_{n+1}$*) and* $NAC_{p_c} = DL_{pc}(NAC_{L_n}) \cup D_{m_c}(NAC_{L'_c}))$*. It is a valid direct graph transformation and* g_c *satisfies* NAC_{p_c} *[12].*

The first component of NAC_{p_c} *is the concurrent NAC of the first* n *transformation. We have to show that for an arbitrary* $k_0 : G_0 \rightarrow H_0$, $k_0 \circ g_c \models D_{m_c}(NAC_{L'_c})$.

As introduced, $D_{e_n}(NAC(n_j))$ *is the gluing of* E *and* N_j *as the right-hand NAC of production* $L_c \overset{l}{\leftarrow} C_c \rightarrow E$*. As we assumed that the NACs only contains stable elements, and* p_0 *creates the first stable elements,* L_c*, the LHS of the concurrent production* p_c *contains no stable elements. Thus according to Corollary 8.2.0.1 the pushout complement* (4) *does not exists, and* $NAC(n''_j)$ *is satisfied by arbitrary* g_c *and* k_0*. Thus, the extension diagram exists, and from Theorem 8.1.1 the compositionality follows.*

Case 2: *A rule pair* $p_i, p_{i+1} \in P$ *exists such that all stable elements in* NAC_{i+1} *are gluing points in* p_i*. In this case the Corollary 8.2.0.1 does not* *apply for this rule pair.*

We create the concurrent rule p_{c_i} for p_i, p_{i+1} with concurrent NACs $D_m(NAC_{L_i})$ and $DLp_c(NAC_{L_{i+1}})$. From Concurrency Theorem with NACs *if follows that we can use the concurrent rule p_{c_i} in transformation sem instead p_i, p_{i+1}.*

If other rule pairs $p_j, p_{j+1} \in P$ exists that NAC_{j+1} contains only gluing points from p_j we perform the previous step until there will be no such rule. This is possible, since G_0 does not contain stable elements, since all stable elements are created at a certain p_k production rule, and then they will be created points in the NAC_{k+1}. Then compositionality follows from Case 1.

The conditions in Theorem 8.2.1 are naturally satisfied in the case of triple graph grammars [18]: Created types are elements of the target and relational metamodels, hence they do not occur in source models. The only real restriction is that no negative application conditions are allowed on the source model. Our mapping from UML architectural models to CSP satisfies these restrictions and is thus compositional.

Chapter 9

Conclusion and Future Work

The results presented in this paper are spanning two levels of abstraction. At the level of architectural refactoring, we have developed a method for verifying transformations of UML architectural models based on a semantic mapping into CSP processes. More generally, we have shown that the correctness of such an approach depends on the compositionality of the semantic mapping, and that this property can be guaranteed by a structural condition on the form of the mapping rules which is easily satisfied, for example, by triple graph grammars.

Future work will continue to address both levels. At the concrete level we hope to be able to come up with a catalogue of verified refactoring rules, rather than relying on the extraction of rules from individual transformations as in this paper. It remains to be seen if a general catalogue comparable to OO refactorings is possible. In general, the approach of rule extraction needs to be supported by heuristics about which elements of a model, apart from those that are changed, should be included into the rule in order to verify its semantic compatibility.

Acknowledgements

This work is funded by the IST-FET IST-2005-16004 project SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

Bibliography

[1] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382, 1998.

[2] 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.

[3] 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.

[4] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[5] G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proc. European Software Engineering Conference (ESEC/FSE 01), Vienna, Austria*, volume 1301, pages 327–343. Springer Verlag, 2001.

[6] Formal Systems Europe Ltd. *FDR2 User Manual*, 2005. http://www.fsel.com/documentation/fdr2/html/index.html.

[7] 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.

[8] Dan Hirsch, Paolo Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *ISAW '98:*

Proceedings of the third international workshop on Software architecture, pages 69–72, New York, NY, USA, 1998. ACM Press.

[9] Charles Antony Richard Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice Hall, April 1985.

[10] Diogenes Laertius. *Lives of Eminent Philosophers*, volume 2. Loeb Classical Library, January 1925.

[11] Leen Lambers. Adhesive high-level replacement systems with negative application conditions. Technical report, Technische Universität Berlin, 2007.

[12] Leen Lambers, Hartmut Ehrig, Fernando Orejas, and Ulrike Prange. Adhesive high-level replacement systems with negative application conditions. In *Proceedings of Applied and Computational Category Theory Workshop*. Kluwer Academic, 2007.

[13] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[14] 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.

[15] Tiago Massoni, Rohit Gheyi, and Paulo Borba. An approach to invariant-based program refactoring. In *Software Evolution through Transformations 2006*. Electronic Communications of the EASST, 2006.

[16] 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.

[17] OMG. *Unified Modeling Language, version 2.1.1*, 2006. http://www.omg.org/technology/documents/formal/uml.htm.

[18] 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.

[19] Bran Selic. Using uml for modeling complex real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[20] Gerson Suny, D. Pollet, Y. Le Traon, and J.-M. Jzquel. Refactoring uml models, 2001.

[21] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 179–193, London, UK, 2000. Springer-Verlag.

[22] Tiger Developer Team. *Tiger EMF Transformer*, 2007. `http://www.tfs.cs.tu-berlin.de/emftrans`.

[23] 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.

[24] Michel Wermelinger and José Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.*, 44(2):133–155, 2002.

[25] 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.