Integration of Specification Techniques

Christian Krog Madsen

LYNGBY 2003
EKSAMENSPROJEKT
NR. 2003-66

IMM

Trykt af IMM, DTU

Abstract

Graphical specification notations have gained much popularity in software engineering, as witnessed by the widespread adoption of UML throughout the software industry. Graphical notations are generally characterised by being intuitive to understand, i.e. new users of these notation require little training to become familiar with them. However, few of the graphical notations have a formally defined meaning, so diagrams expressed in such notations are ambiguous – a highly undesirable property for a specification notation. In contrast, formal specification languages have a formally defined mathematical meaning but are comprehensible only to properly educated software engineers. The implication of this is that specifications developed using formal languages are not immediately understandable to the customers and domain experts, and therefore they are difficult to validate. In this thesis we describe the graphical notations of Live Sequence Charts and Statecharts and propose a method using diagrams in these notations for constraining a formal specification expressed in a subset of the RAISE Specification Language. Furthermore, we propose a development method that combines the traditional approach with that of formal development. We give a small example illustrating the application of this method.

Keywords: Specification methods, RAISE Specification Language, graphical specification techniques, Live Sequence Charts, Statecharts.

Resumé

Grafiske specifikationsnotationer har opnået stor popularitet indenfor softwareudvikling, hvilket blandt andet ses i den store udbredelse af UML i softwareindustrien. Grafiske notationer er generelt set karakteriseret ved at være intuitive at forstå, det vil sige at nye brugere af disse notationer har kun behov for kortvarig træning for at bliver fortrolige med dem. Desværre er det kun få grafiske notationer, som har en formelt matematisk defineret betydning. Derfor vil diagrammer udtrykt i sådanne notationer som oftest være tvetydige, hvilket er en uhensigtsmæssig egenskab ved en notation beregnet til specifikation. Modsætningen er formelle specifikationssprog, der har en præcis matematisk defineret betydning, men som kun er forståelige for veluddannede softwareudviklere. Resultatet er, at specifikationer udarbejdet ved brug af formelle notationer ikke umiddelbart kan forstås af kunder og eksperter i domænet, hvilket medfører at disse specifikationer er svære at validere. I denne afhandling beskriver vi de to grafiske notationer Live Sequence Charts og Statecharts og foreslår en metode, der bruger diagrammer udtrykt i disse notationer til at opstille betingelser som skal opfyldes af en formel specifikation skrevet i et uddrag af sproget RAISE Specification Language. Desuden foreslås en udviklingsmetode, som kombinerer den traditionelle metode med den model, der bruges indenfor formel softwareudvikling. Metoden illustreres med et mindre eksempel på dens anvendelse.

Preface

The work reported in this thesis was carried out at the Computer Science and Engineering (CSE) division of the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU) from June through November 2003. The work was supervised by Professor Dines Bjørner.

I would like to thank Professor Dines Bjørner for his valuable guidance during the project and for reading and commenting on draft versions of this thesis. I would also like to thank Associate Professors Martin Fränzle and Michael R. Hansen for the important feedback and advice I received during a colloqium held in September 2003 at IMM/DTU.

Kongens Lyngby, 28 November 2003

Christian Krog Madsen

Contents

Chapter 1

Introduction

In 1968 the first NATO conference on Software Engineering was held in Germany. The agenda was to discuss possible solutions to what was then known as the "software crisis", namely that the discipline of software development was plagued by delays in delivery, budget overruns and faulty software. The term Software Engineering was coined to describe the ideal that should be worked towards in the future. It was defined by Fritz Bauer [48] as

> "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

Since 1968 the field of Software Engineering has evolved at a steady pace. The fundamental problem remains, however, that the increase in system complexity outgrows the increase in the sophistication of our development methodologies. Software developers have tried to keep up with the explosive growth in the possibilities and performance of hardware. The symptoms of this continual crisis are the same as they were 35 years ago: delayed, expensive and faulty software.

At the core of the problem lies the difficulties in forming and communicating abstract ideas. Every software project brings together the knowledge of a range of people, from the prospective users, management of the acquiring organisation, to software architects and developers. The task of the developers is to elicit knowledge of the environment in which the software will function, describe the requirements for the interaction between the system and its environment, and finally implement the requirements as a software system.

In order to communicate decisions and ideas, specifications are used. A specification is (ideally) a precise description that is considered normative.

There exist a wide spectrum of specification methods.

- *Natural language* specifications have the good property that anyone can read them and form an idea of the intention of the specification. This is advantageous in that the domain experts can read it and validate that it gives a suitable and sufficient description of the domain and the system to be built.
  On the negative side, natural language specifications are rarely precise. Extreme care must be taken to avoid ambiguities and omissions. There are no tools to support this process. Another problem is that because natural language does not have a proper semantics, different people may interpret parts of a specification differently.
- *Informal diagrams* share some of the characteristics of natural language descriptions, since they often do not require special training to understand the general idea, while

still being imprecise because they have no semantics. Diagrams tend to emphasise a certain aspect of the system, while ignoring others.
Informal diagrams rely on the intuition that boxes or circles represent some object, entity, concept or situation, while arrows denote a transfer of data, a change in the situation or progression of time.

- *Formalised diagrams* are notations where at least some attempt has been made to define what a diagram means, i.e. to give it a semantics. There are varying degrees of formality. One example of a diagrammatic notation that has both a well-defined syntax and a proper mathematical semantics is Petri Nets [45, 52].
  Compared to informal diagrams, formalised diagrams usually require some familiarisation with the notation before they can be understood.
- *Formal specification languages* are based on concepts from mathematics, such as sets, functions, relations and algebras. They have a well-defined syntax and a complete mathematical semantics. Often, they include a proof system admitting formal reasoning about specifications.
  Formal specification languages are only readable to specially trained professionals with a firm understanding of mathematics and computer science. Another concern is that aspects are hard to separate in a specification. The technique of stepwise refinement, i.e. developing a specification in steps from a very abstract model towards a detailed model, may be applied to at least partly solve this problem.

As the degree of formality increases, the prerequisites for understanding also increase. A possible solution could be to make parallel specifications using both graphical notations and formal notations. Clearly, this raises a new problem, namely that of ensuring consistency between the two. It is exactly that problem, which is the topic of this thesis.

This leads to the main hypothesis of this thesis:

> *By providing a formal link between graphical notations and formal specification languages, we can achieve the best of both worlds.*

There are many candidate graphical notations and specification languages to choose from. We choose the notations of Statecharts and Live Sequence Charts.

Statecharts are specifically designed for specifying the behaviour of reactive systems, i.e. systems which are driven by and respond to external events. Statecharts are derived from state machines and have been incorporated in the UML method. Several tools supporting Statecharts are available, most notably the Statemate tool [24] developed by I-Logix.

Live Sequence Charts are still relatively new, but are derived from Message Sequence Charts which are both standardised by the ITU [9, 30, 31] and enjoy considerable use particularly in the telecommunication industry. Some tool support is available for Live Sequence Charts, for example the Play-Engine developed by David Harel and Rami Marelly [25].

Additionally, Statecharts and Live Sequence Charts complement each other well: Statecharts specify the internal behaviour of a system component and Live Sequence Charts specify the externally observable behaviour of the components and the protocols by which the components communicate.

As for the specification language, we choose the RAISE Specification Language [49, 50], because we are already familiar with that notation, it has tool support and is highly expressive. It allows both algebraic and model-oriented specification styles and supports applicative and imperative, sequential and parallel models. Also, RSL has a proof system for doing provably correct stepwise refinement.

There is currently considerable research effort being directed at integrating different specification techniques. One direction is the Unifying Theories of Programming of Hoare and He [29] and the further development by Woodcock [58]. They attempt to bring together elements of sequential and parallel programming in a framework that links denotational, operational and algebraic semantics based on relational calculus. Other directions are proposals for integrating concurrent process algebras such as CSP or CCS with languages allowing structured data types and values, such as Z, Object-Z or VDM. Examples are the combination of Z and CCS, named ZCCS [15] and the combination of Object-Z with CSP, CSP-OZ [13]. The RAISE Specification Language (RSL) is itself the result of the integration of the algebraic and model-oriented specification styles with facilities for imperative and applicative, sequential and concurrent specifications in a single semantic framework. RSL has also been extended with time by George and Xia [16] and linked to Duration Calculus [59, 10] by Haxthausen and Xia [27].

In the literature, some attempts at integrating graphical notations with formal notations have been reported. Much of this work is related to the quest for providing a formal foundation for the UML method. Weber [57] presents a case study where a system is modelled on an architectural, reactive and functional level using Class Diagrams, Statecharts and Z, respectively. More specifically, the system is divided into objects using the Class Diagram, the reactive behaviour of each object is specified using a Statechart, and the state transitions are specified using Z. Grieskamp *et al* [18] and Büssow *et al* [8] describe the same approach as part of the Express project. A closely related approach is the integration of Statecharts with the algebraic specification language Casl by Reggio and Repetto [51] developed as part of the Common Framework Initiative (CoFI). The combined notation, known as Casl-Chart, uses Statechart notation for specifying states and transitions and CASL for specifying data and the functions describing the state change associated with each transition.

1.1 Thesis Structure

The thesis is structured as follows.

Chapter 2 – Live Sequence Charts
We begin by describing the graphical syntax of Message Sequence Charts (MSC) and Live Sequence Charts (LSC). Using ideas from the official semantics of MSC given in the ITU-T standard Z.120, we develop a process algebra and use this to define the semantics of LSC. Finally, we give an example where the semantics is used to derive a process algebraic term from an LSC diagram.

Chapter 3 – Statecharts
We describe the graphical syntax of Statecharts. We then review a process algebraic semantics of Statecharts proposed in the literature. Finally, we derive the process algebra term from an example Statechart.

Chapter 4 – Relating Diagrams to RSL
We present the syntax and semantics of a subset of RSL and define a way to extract a process algebraic term describing the communication behaviour of a process expressed in RSL. We then define a satisfaction relation, that expresses the conditions under which an RSL implementation correctly implements an LSC. We then define a similar procedure for checking whether an RSL specification correctly implements a Statechart.

Chapter 5 – Development Method
We propose a development method that combines the use of diagrams with RSL and relates the two forms of specification notation using the satisfaction relations of the previous chapter.

Chapter 6 – Example Application of the Development Method
We present an example development – following the method of the previous chapter – of a specification for the Two-Phase Commit Protocol used in implementing distributed transactions.

Chapter 7 – Conclusion
We review the contributions of this thesis and give directions for future research.

Appendix A – A Critique of Live Sequence Charts
During the study of the semantics of Live Sequence Charts, we came upon a number of problematic issues. In this appendix we discuss some of these issues.

Appendix B – Proofs
This appendix contains a proof that is used in Chapter 2.

Chapter 2

Live Sequence Charts

2.1 Introduction

Live Sequence Charts (LSC) is a graphical language introduced by Damm and Harel [11] for specifying interactions between components in a system. It is an extension of the widely used language Message Sequence Charts (MSC). MSCs are frequently used in the specification of telecommunication systems and are closely related to the Sequence Diagrams of UML. Both the graphical and textual syntax of MSCs are standardised by the ITU in Recommendation Z.120 [9, 30, 31]. The standard gives an algebraic semantics of MSCs. LSC extends MSC by promoting conditions to first class elements and providing notations for specifying mandatory and optional behaviour.

Reader's Guide The material on the syntax of MSC and LSC in section Section 2.2 is intended as a quick tutorial as well as for quick reference. The section Section 2.3 on process algebra is rather involved and may seem a bit detached from the context. The Reader is encouraged to refer to the example in Section 2.5 to see how the semantics of a chart is derived.

2.2 Live Sequence Chart Syntax

In this section we describe the components of Live Sequence Charts. We return to the question of the semantics of a subset of LSC in Section 2.4. Since Live Sequence Charts were derived from Message Sequence Charts and share many similarities with them, we begin by describing the syntax of Message Sequence Charts and then proceed to describe the syntax extensions introduced by Live Sequence Charts.

2.2.1 Graphical Syntax of Message Sequence Charts

Message Sequence Charts (MSC) were first standardised by the CCITT (now ITU-T) as Recommendation Z.120 in 1992 [9]. The standard was later revised and extended in 1996 [30] and in 1999 [31]. The original standard specified the components of an MSC. The 1996 standard also specified how several MSCs (called *basic* MSCs) can be combined to form an MSC document, in which the relation between the basic MSCs is defined by a *high-level* MSC. The most recent standard provides additional facilities for specifying the data that is passed in messages and also allows inline expressions. Here we will restrict the description to the 1992 version of the standard.

An MSC consists of a collection of instances. An instance is an abstract entity on which events can be specified. Events are message inputs, message outputs, actions, conditions, timers, process control events and coregions. An instance is denoted by a hollow box with a vertical line extending from the bottom. The vertical line represents a time axis where time runs from top to bottom. Each instance thus has its own time axis and time may progress differently on two axes.

Events specified on an instance are totally ordered in time. Events execute instantaneously and two events cannot take place at the same time. Events on different instances are partially ordered, since the only requirement is that message input by one instance must be preceded by the corresponding message output in another instance.

Actions are events that are local to an instance. Actions are represented by a box on the time line with an action label inside. Actions are used to specify some computation that changes the internal state of the instance.

A *message output* represents the sending of a message to another instance or the environment. A *message input* represents the reception of a message from another instance or the environment. For each message output to another instance there must be a matching message input. A message exchange consists of a message output and a message input. A message exchange is represented as an arrow from the time line of the sending instance to the time line of the receiving instance. In case of messages exchanged with the environment, the side of the diagram can be considered to be the time line of the environment. The arrow is labelled with a message identifier. Message exchange is asynchronous, i.e. message input is not necessarily simultaneous with message output.

Example 2.2.1. Figure 2.1 shows an MSC with two instances, A and B. Instance A sends the message m_1 to instance B followed by message m_2 sent to the environment, B then performs some action, a, and sends the message m_3 to A. □

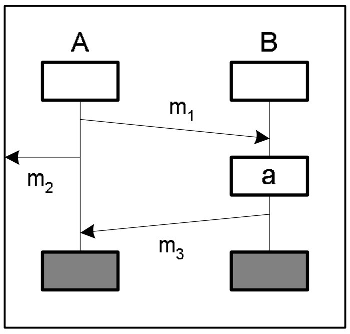


Figure 2.1: Message and action events.

Example 2.2.2. Figure 2.2 shows two situations that violate the partial order induced by message exchange. Thus it is an invalid MSC.

Because events are totally ordered on an instance time line, the reception of message m_1 precedes the sending of m_1. This conflicts with the requirement that message input be preceded by message output.

The exchange of messages m_2 and m_3 illustrates another situation that violates the partial order, as shown by the following informal argument. Let the partial order be denoted $\leq$ and let the input and output of message m be denoted by $in(m)$ and $out(m)$, respectively.

Using the total ordering on events on an instance time line we have

$$in(m_3) \leq out(m_2)$$
$$in(m_2) \leq out(m_3)$$

Using the partial ordering on message events we have

$$out(m_2) \leq in(m_2)$$

Now, by transitivity of $\leq$, $in(m_3) \leq out(m_3)$, thus violating the partial ordering on message events. □

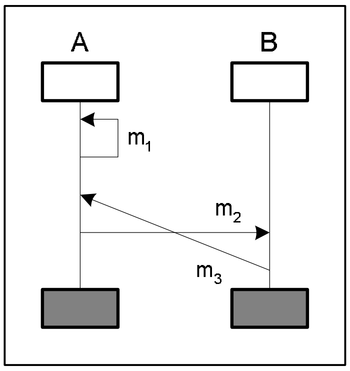


Figure 2.2: Illegal message exchanges.

Conditions describe a state that is common to a subset of instances in an MSC. Conditions have no semantic import and merely serve as documentation. Conditions are represented as a hexagon extending across the time lines of the instances for which the condition applies. The condition text is placed inside the hexagon.

Example 2.2.3. Figure 2.3 illustrates conditions. Condition c_1 is local to instance B. Condition c_2 is a shared condition on instances A and B. Condition c_3 is a shared condition on instances A and C. Note that the time line of B is passed through the hexagon for condition c_3 to indicate that B does not share condition c_3. □

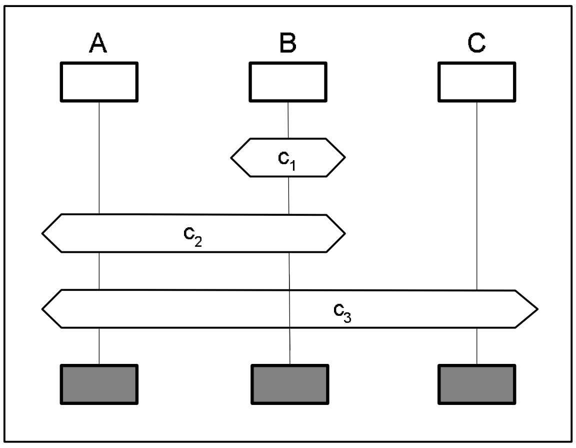


Figure 2.3: Conditions.

There are three *Timer* events: timer set, timer reset and timeout. Timers are local to an instance. The setting of a timer is represented by an hourglass symbol placed next to the instance time line and labelled with a timer identifier. Timer reset is represented by a cross linked by a horizontal line to the time line. Timer timeout is represented by an arrow from the hourglass symbol to the time line. Every timer reset and timeout event must be preceded by the corresponding timer set event. There is no notion of quantitative time in MSC, so timer events are purely symbolic. Extensions of MSC with time have been studied in [4, 40, 41].

Example 2.2.4. Figure 2.4 shows the syntax for timer events. On instance A, the timer T is set and subsequently timeout occurs. On instance B, the timer T' is set and subsequently reset. □

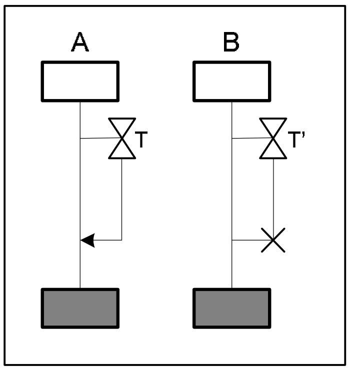


Figure 2.4: Timer events.

An instance may create a new instance. This is called *process creation*. An instance may also cause itself to terminate. This is called *process termination*. Process creation is represented by a dashed arrow from the time line of the creating instance to a new instance symbol with associated time line. Process termination is represented by a cross as the last symbol on the time line of the instance that terminates.

Example 2.2.5. Figure 2.5 shows the creation of instance B by instance A and the subsequent termination of B. □

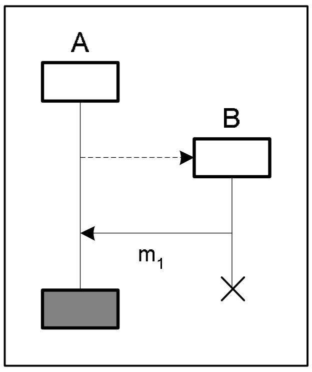


Figure 2.5: Process creation and termination.

Coregions are parts of the time line of an instance where the usual requirement of total ordering is lifted. Within a coregion only message exchange events may be specified and these events may happen in any order, regardless of the sequence in which they are specified. Message exchanges between two instances may be ordered in one instance and unordered in the other instance. Coregions are represented by replacing part of the fully drawn time line with a dashed line.

Example 2.2.6. Figure 2.6 illustrates a coregion in instance B. Because of the coregion, there is no ordering on the input of messages m_1 and m_2 in instance B, so they may occur in any order. □

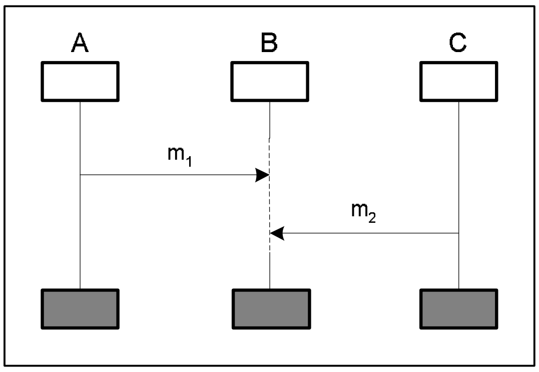


Figure 2.6: Coregion.

In order to increase the readability of complex MSCs, the standard specifies a form of hierarchical decomposition of complex diagrams into a collection of simpler diagrams. This is known as *instance decomposition*. For each decomposed instance there is a sub-MSC, which is itself an MSC. The single instance that is decomposed is represented by more than one instance in the sub-MSC. The behaviour observable by the environment of the sub-MSC should be equivalent to the observable behaviour of the decomposed instance.

Example 2.2.7. In Figure 2.7 instance B is decomposed into two instances, B_1 and B_2 in the sub-MSC. The message events in which B participates are represented as message exchanges with the environment in the sub-MSC. The message m_{int} exchanged between B_1 and B_2 is internal to the decomposed instance, and is thus not visible in the main MSC. □

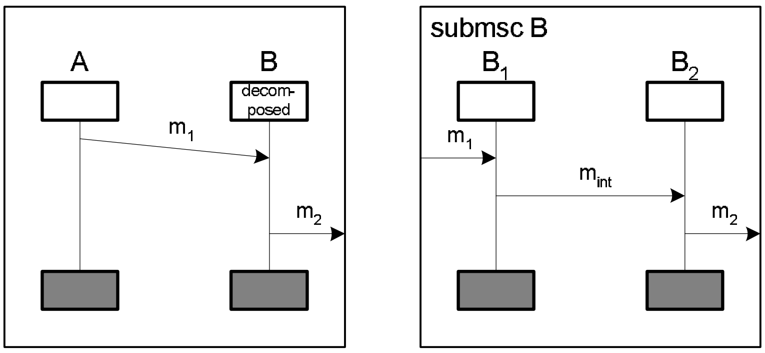


Figure 2.7: Instance decomposition.

2.2.2 Graphical Syntax of Live Sequence Charts

Live Sequence Charts (LSC) were proposed by Damm and Harel [11] as an extension of Message Sequence Charts. They identified a number of shortcomings and weaknesses of the MSC standard and proposed a range of new concepts and notation to overcome these problems.

One of the major problems with the semantics of MSCs is that it is not clear whether an MSC describes all behaviours of a system or just a set of possible behaviours. Typically, the latter view would be used in early stages of development, while the former would apply in later stages when the behaviour is more fixed. Another problem noted by Damm and Harel is the inability of MSCs to specify liveness, i.e. MSCs have no constructions for enforcing progress. They also view the lack of semantics for conditions to be a problem.

The most prominent feature of LSC is the introduction of a distinction between optional and mandatory behaviour. This applies to several elements in charts. A distinction is introduced between *universal* charts and *existential* charts. Universal charts specify behaviour that must be satisfied by every possible run of a system. This may be compared to universal quantification over the runs of the system. On the other hand, existential charts specify behaviour that must be satisfied by at least one run of the system. This is like existential quantification over the runs of the system. The typical application of existential charts would be in the early stages of the development process, particularly in domain modelling. An existential chart specifies a scenario that may be used to describe characteristic behaviours of the domain. Universal charts would typically be used later in the development process, particularly in requirements engineering and in requirements documents. Universal charts are denoted by a fully drawn box around the chart, while existential charts are denoted by a dashed box.

Example 2.2.8. Figure 2.8 shows a universal LSC with two instances, A and B. The four messages are discussed in example 2.2.10 below. The behaviour specified by this chart should be satisfied by every run of the system.

Figure 2.9 shows an existential LSC. This represents a scenario that at least one run of the system must satisfy.

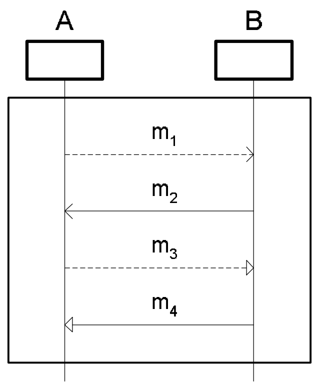


Figure 2.8: Universal chart.

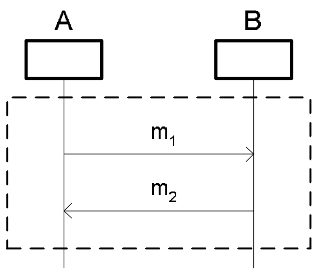


Figure 2.9: Existential chart.

□

LSC introduces the notion of a *prechart* to restrict the applicability of a chart. The prechart is like a precondition that when satisfied activates the main chart. A given system need only satisfy a universal chart whenever it satisfies the prechart. An empty prechart is satisfied by any system. A prechart can be considered as the expression in an IF-statement where the body of the THEN part is the universal chart. The prechart is denoted by a dashed hexagon containing zero, one or more events.

Example 2.2.9. Figure 2.10 shows a universal LSC with a prechart consisting of the single message *activate*. In this case, the behaviour specified in the body of the chart only applies to those runs of the system where the message *activate* is sent from instance A to instance B. □

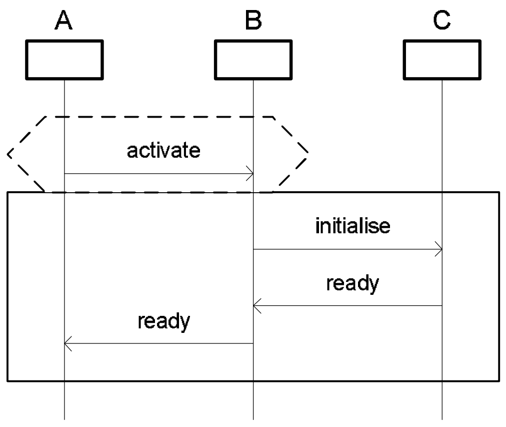


Figure 2.10: Prechart.

LSC allows messages to be "hot" or "cold". A "hot" message is mandatory, i.e. if it is sent then it must be received eventually. This is denoted by a fully drawn arrow. For a "cold" message reception is not required, i.e. it may be "lost". This is denoted by a dashed arrow. Also, a message may be specified as either synchronous or asynchronous. Synchronous messages are denoted by an open arrowhead, while asynchronous messages are denoted by a closed arrowhead.

Example 2.2.10. Figure 2.8 illustrates the four kinds of messages: hot and cold, synchronous and asynchronous. Message m_1 is cold and asynchronous. Message m_2 is hot and asynchronous. Message m_3 is cold and synchronous. Finally, message m_4 is hot and synchronous. □

In LSC conditions are promoted to first-class events. The difference is that conditions now have an influence on the execution of a chart, while in MSC they were merely comments. Again, a distinction is made between a "hot" (mandatory) condition, which, if evaluated to false, causes non-successful termination of the chart, and a "cold" condition (optional) which, if evaluated to false, causes successful termination of the chart. A "hot" condition is like an invariant which must be satisfied. By combining a prechart with a universal chart containing just a single hot condition that always evaluates to false, is is possible to specify forbidden scenarios, since the scenario expressed in the prechart will then always cause non-successful termination. A shared condition forces synchronization among the sharing instances, i.e. the condition will not be evaluated before all instances have reached it and no instance will progress beyond the condition until it has been evaluated.

Example 2.2.11. Figure 2.11 illustrates two conditions. The first is hot, while the second is cold. If the hot condition evaluates to false, the chart is aborted, indicating an erroneous situation. If the second condition evaluates to false, the current (sub)chart is exited successfully. □

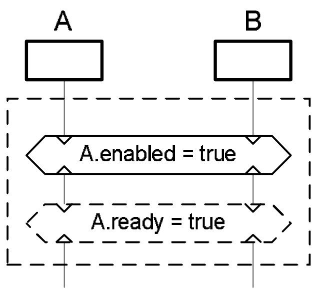


Figure 2.11: Conditions.

Iteration and conditional execution are obtained by means of *subcharts*. Subcharts are LSCs that are specified for a subset of the instances of the containing LSC and possibly additional new instances. Iteration is denoted by annotating the top-left corner of the chart with an integer constant for limited iteration or an asterisk for unlimited iteration. A subchart is exited successfully either when a limited iteration has executed the specified number of times, or when a cold condition evaluates to false. By combining subcharts with cold conditions, WHILE and DO-WHILE loops may be created. Additionally, a special form of subchart with two parts is used to create an IF-THEN-ELSE construct. The first part of the subchart has a cold condition as the first event. If the condition evaluates to true, the first part of the subchart is executed. If the condition evaluates to false, the second part of the subchart is executed.

Example 2.2.12. Figure 2.12 illustrates limited iteration. Instance A will send the message m_1 60 times to instance B.

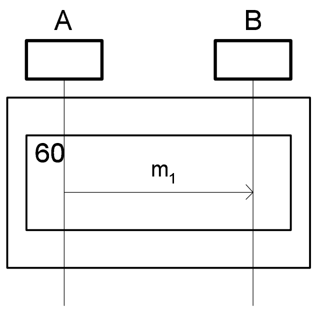


Figure 2.12: Limited iteration.

Figure 2.13 illustrates unlimited iteration with a stop condition, essentially like a DO-WHILE loop. The message m_1 will be sent repeatedly until the condition becomes false. Once that happens, the subchart is exited.

Figure 2.14 is similar to the previous situation, except that the condition is now checked before the first message is sent, thus mimicking a WHILE loop.

Figure 2.15 is like Figure 2.14 except that there is no iteration. Thus, the message m_1 will be sent once if the condition evaluates to true, and it will not be sent if the condition evaluates to false. Therefore, this construction is like an IF-THEN construct.

In Figure 2.16 the special construction for IF-THEN-ELSE is illustrated. The two sub-charts represent the THEN and ELSE branches. If the condition evaluates to true, the

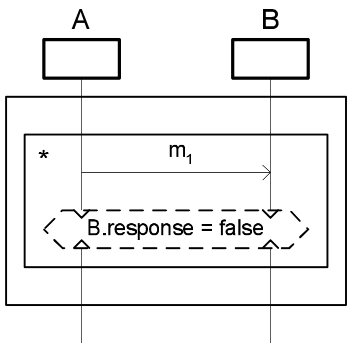


Figure 2.13: DO-WHILE loop.

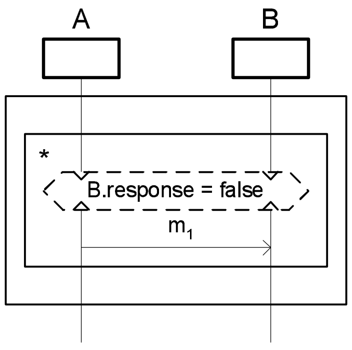


Figure 2.14: WHILE loop.

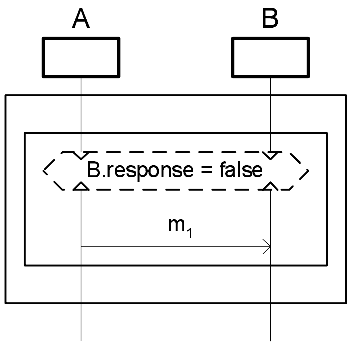


Figure 2.15: IF-THEN conditional.

first subchart is executed, otherwise the second subchart is executed. In either case, the subchart not chosen is skipped entirely.

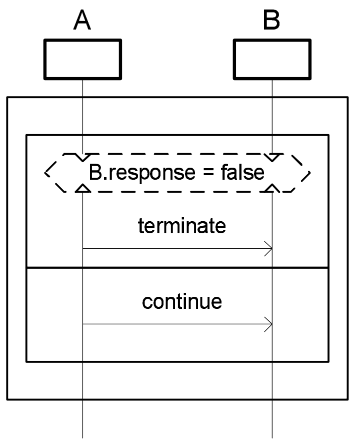


Figure 2.16: IF-THEN-ELSE conditional.

□

The distinction between "hot" and "cold" is also applied to the time line of an instance. Any point where an event is specified on the time line is called a *location*. A location may be "hot" indicating that the corresponding event must eventually take place, or "cold" indicating that event may never occur. A "hot" location is represented by the time line being fully drawn, while a "cold" location is represented by a dashed time line. The time line may alternate between being fully drawn and dashed.

The addition of "cold" location conflicts with the representation of coregions inherited from Message Sequence Charts. For this reason, the syntax for a coregion is modified to be a dashed line positioned next to the part of the time line that the coregion spans.

Example 2.2.13. Figure 2.17 illustrates the syntax for optional progress. The time line is fully drawn at the location where the message m_1 is sent and received, indicating the these events must eventually take place. Thus, this guarantees liveness. At the location where the message m_2 is sent and received, the time line is dashed, indicating that neither instance is required to progress to the sending or receiving of m_2. If an instance does not progress beyond a location l, then no event on the time line of that instance following l will take place. Thus, in this case, if m_2 is never sent, m_3 will never be sent.

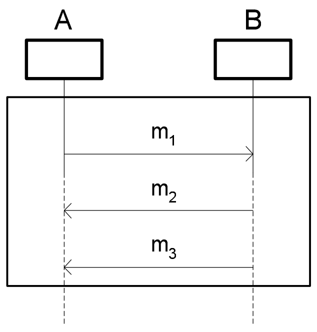


Figure 2.17: Optional progress.

□

2.3 Process Algebra

The ITU standard Z.120 for Message Sequence Charts includes a formal algebraic semantics based on the process algebra PA_ϵ introduced by Baeten and Weijland [3]. In this section we first review the definition of PA_ϵ following [43] and [2] and then present an extension of that algebra (named PAc_ϵ), which will be used for defining the semantics of a subset of Live Sequence Charts in Section 2.4.2 and for expressing communication behaviours of RSL specifications in Section 4.3.2.

2.3.1 The Process Algebra PA_ϵ

The algebraic theory of PA_ϵ is given as an equational specification $(\Sigma_{PA_\epsilon}, E_{PA_\epsilon})$, consisting of the signature, Σ_{PA_ϵ}, and a set of equations, E_{PA_ϵ}. We first define the signature and equations and then give the intuition behind the definitions.

Signature

The one-sorted signature, Σ_{PA_ϵ}, consists of

1. two special constants δ and ϵ
2. a set of unspecified constants A, for which $\{\delta, \epsilon\} \cap A = \emptyset$
3. the unary operator $\surd$
4. the binary operators $+$, $\cdot$, $\|$ and $\lfloor\!\lfloor$.

The unspecified set A is a parameter of the theory. Thus, applications of the theory require the theory to be instantiated with a specific set A. When the theory is applied to Message Sequence Charts, the set A will consist of identifiers for the atomic events of the chart.

For convenience and following tradition, we will apply the binary operators in infix notation, i.e. instead of $+(x, y)$ we will write $x + y$. To reduce the need for parentheses operator precedences are introduced. The $\cdot$ operator binds strongest. The $+$ operator binds weakest.

Let V be a set of variables, then terms over the signature Σ_{PA_ϵ} with variables from V, denoted $T(\Sigma_{PA_\epsilon}, V)$, are given by the inductive definition

1. $v \in V$ is a term;
2. $a \in A$ is a term;
3. δ is a term;
4. ϵ is a term;
5. if t is a term, then $\surd(t)$ is a term;
6. if t_1 and t_2 are terms, then $t_1 op\ t_2$ is a term, for $op \in \{+, \cdot, \|, \lfloor\!\lfloor\}$.

A term is called *closed* if it contains no variables. The set of closed terms over Σ_{PA_ϵ} is denoted $T(\Sigma_{PA_\epsilon})$.

Equations

The equations of PA_ϵ are of the form $t_1 = t_2$, where $t_1, t_2 \in T(\Sigma_{PA_\epsilon}, V)$. For $a \in A$ and $x, y, z \in V$ the equations, E_{PA_ϵ}, are given in Table 2.1.

$$x + y = y + x \tag{A1}$$
$$(x + y) + z = x + (y + z) \tag{A2}$$
$$x + x = x \tag{A3}$$
$$(x + y) \cdot z = x \cdot z + y \cdot z \tag{A4}$$
$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{A5}$$
$$x + \delta = x \tag{A6}$$
$$\delta \cdot x = \delta \tag{A7}$$
$$x \cdot \epsilon = x \tag{A8}$$
$$\epsilon \cdot x = x \tag{A9}$$

$$x \parallel y = x \mathbin{\|\!\!\lfloor} y + y \mathbin{\|\!\!\lfloor} x + \surd(x) \cdot \surd(y) \tag{F1}$$
$$\epsilon \mathbin{\|\!\!\lfloor} x = \delta \tag{F2}$$
$$\delta \mathbin{\|\!\!\lfloor} x = \delta \tag{F3}$$
$$a \cdot x \mathbin{\|\!\!\lfloor} y = a \cdot (x \parallel y) \tag{F4}$$
$$(x + y) \mathbin{\|\!\!\lfloor} z = x \mathbin{\|\!\!\lfloor} z + y \mathbin{\|\!\!\lfloor} z \tag{F5}$$

$$\surd(\epsilon) = \epsilon \tag{T1}$$
$$\surd(\delta) = \delta \tag{T2}$$
$$\surd(a \cdot x) = \delta \tag{T3}$$
$$\surd(x + y) = \surd(x) + \surd(y) \tag{T4}$$

Table 2.1: Equations of PA_ϵ.

Intuition

The special constant δ is called *deadlock*. It denotes the process that has stopped executing actions and can never resume. The special constant ϵ is called the *empty process*. It denotes the process that terminates successfully without executing any actions. The elements of the set A are called *atomic actions*. These represent processes that cannot be decomposed into smaller parts. As mentioned above, the set A is given a concrete definition when the theory is applied. For example, in defining the semantics of Message Sequence Charts, the set A will contain the symbols that identify the events in the chart, such as $in(a, b, m1)$ identifying the event of instance b receiving message $m1$ from instance a.

The binary operators $+$ and $\cdot$ are called *alternative* and *sequential composition*, respectively. The alternative composition of processes x and y is the process that behaves as either x or y, but not both. The sequential composition of processes x and y is the process that first behaves as x until it reaches a terminated state and then behaves as y.

The binary operator $\|$ is called the *free merge*. The free merge of processes x and y is the process that executes an interleaving of the actions of x and y. The unary *termination operator* $\surd$ indicates whether the process it is applied to may terminate immediately. The termination operator is an auxiliary operator needed to define the free merge. The binary operator $⌊\!⌊$ is called the *left merge* and denotes the process that executes the first atomic action of the left operand followed by the interleaving of the remainder of the left operand with the right operand. Like the termination operator, the left merge operator is an auxiliary operator needed to define free merge.

To see why the termination operator is necessary, consider equation F1. What happens in the free merge is that all possible sequences of atomic actions from the two operands are generated. When both operands become the empty process, we want the free merge to be the empty process as well, i.e. we want the equation $\epsilon \parallel \epsilon = \epsilon$ to hold. Because of equation F2, the two first alternatives in F1 become deadlock. However, the last alternative becomes the empty process, because of equation T1. Thus, with A6 we get the desired result. It is possible to give a simpler definition of the free merge without using the empty process or the termination operator, see [2], but for our purposes we need the empty process.

Derivability

We now define what it means for a term to be derivable from an equational specification. First, the two auxiliary notions of a substitution and a context are introduced.

Definition 2.3.1. A substitution $\sigma : V \to T(\Sigma, V)$ replaces variables with terms over Σ. The extension of σ to terms over Σ, denoted $\bar{\sigma} : T(\Sigma, V) \to T(\Sigma, V)$, is given by

1. $\bar{\sigma}(\delta) = \delta$
2. $\bar{\sigma}(\epsilon) = \epsilon$
3. $\bar{\sigma}(a) = a \quad \text{for } a \in A$
4. $\bar{\sigma}(v) = \sigma(v) \quad \text{for } v \in V$
5. $\bar{\sigma}(\surd(x)) = \surd(\bar{\sigma}(x))$
6. $\bar{\sigma}(x \; op \; y) = \bar{\sigma}(x) \; op \; \bar{\sigma}(y) \quad \text{for } op \in \{+, \cdot, \|, ⌊\!⌊\}$

A substitution that replaces all variables with variable-free terms, i.e. closed terms, is called *closed*. □

Definition 2.3.2. A Σ context is a term $C \in T(\Sigma, V \cup \{\Box\})$, containing exactly one occurrence of the distinguished variable $\Box$. The context is written $C[\;]$ to suggest that C

should be considered as a term with a hole in it. Substitution of a term $t \in T(\Sigma, V)$ in $C[\,]$ gives the term $C[\Box \mapsto t]$, written $C[t]$. $\Box$

Definition 2.3.3. Let (Σ, E) be an equational specification and let t, s and u be arbitrary terms over Σ. The derivability relation, $\vdash$, is then given by the following inductive definition.

$$\begin{array}{l}
s = t \in E \quad \Rightarrow \quad (\Sigma, E) \vdash s = t \\
(\Sigma, E) \vdash t = t \\
(\Sigma, E) \vdash s = t \quad \Rightarrow \quad (\Sigma, E) \vdash t = s \\
(\Sigma, E) \vdash s = t \ \wedge \ (\Sigma, E) \vdash t = u \quad \Rightarrow \quad (\Sigma, E) \vdash s = u \\
(\Sigma, E) \vdash s = t \quad \Rightarrow \quad (\Sigma, E) \vdash \bar{\sigma}(s) = \bar{\sigma}(t) \quad \text{for any substitution } \sigma \\
(\Sigma, E) \vdash s = t \quad \Rightarrow \quad (\Sigma, E) \vdash C[s] = C[t] \quad \text{for any context } C[-]
\end{array}$$

If $(\Sigma, E) \vdash s = t$, abbreviated $E \vdash s = t$, then the equation $s = t$ is said to be *derivable* from the equational specification (Σ, E). $\Box$

Reduction to basic terms

We now venture deeper into the theory of process algebra and term rewriting systems. The goal is to show that there exists a model of the equational specification for PA_ϵ and that the equations E_{PA_ϵ} form a complete axiomatisation, i.e. that whenever two terms are equal in the model, then they are provably equal using the equations.

The first step is to show that any PA_ϵ term can be reduced to an equivalent so-called basic term consisting of only atomic actions, δ, ϵ, $+$ and $\cdot$. This result makes subsequent proofs easier, because we need only consider these simpler terms.

Definition 2.3.4. δ and ϵ are basic terms. An atomic action $a \in A$ is a *basic term.* If $a \in A$ and t is a basic term, then $a \cdot t$ is a basic term. If t and s are basic terms, then $t + s$ is a basic term. $\Box$

The next step is to show that any PA_ϵ term can be reduced to a basic term. To do this, a term rewriting system is defined.

Definition 2.3.5. A *term rewriting system* is a pair (Σ, R) of a signature, Σ, and a set, R, of rewriting rules. A rewriting rule is of the form $s \to t$, where $s, t \in T(\Sigma, V)$ are open terms over Σ, such that s is not a variable and $vars(t) \subseteq vars(s)$, where $vars(t)$ denotes the set of variables in the term t.

The one-step reduction relation, $\to$, is the smallest relation containing the rules, R, that is closed under substitutions and contexts. $\Box$

Definition 2.3.6. A term s is in *normal form* if there does not exist a term t, such that $s \to t$. A term s is called *strongly normalising* if there exist no infinite sequences of rewritings starting with s:

$$s \to s_1 \to s_2 \to \ldots$$

A term reduction system is called strongly normalising if every term in the system is strongly normalising. $\Box$

The term rewriting system for PA_ϵ is shown in Table 2.2. Essentially, a term rewriting system is a collection of equations, that can be applied only one way. Compared with

$$x + x \to x \tag{RA3}$$
$$(x + y) \cdot z \to x \cdot z + y \cdot z \tag{RA4}$$
$$(x \cdot y) \cdot z \to x \cdot (y \cdot z) \tag{RA5}$$
$$x + \delta \to x \tag{RA6}$$
$$\delta \cdot x \to \delta \tag{RA7}$$
$$x \cdot \epsilon \to x \tag{RA8}$$
$$\epsilon \cdot x \to x \tag{RA9}$$

$$x \parallel y \to x \mathbin{⫠} y + y \mathbin{⫠} x + \surd(x) \cdot \surd(y) \tag{RF1}$$
$$\epsilon \mathbin{⫠} x \to \delta \tag{RF2}$$
$$\delta \mathbin{⫠} x \to \delta \tag{RF3}$$
$$a \cdot x \mathbin{⫠} y \to a \cdot (x \parallel y) \tag{RF4}$$
$$a \mathbin{⫠} x \to a \cdot x \tag{RF4'}$$
$$(x + y) \mathbin{⫠} z \to x \mathbin{⫠} z + y \mathbin{⫠} z \tag{RF5}$$

$$\surd(\epsilon) \to \epsilon \tag{RT1}$$
$$\surd(\delta) \to \delta \tag{RT2}$$
$$\surd(a \cdot x) \to \delta \tag{RT3}$$
$$\surd(x + y) \to \surd(x) + \surd(y) \tag{RT4}$$

Table 2.2: Term rewriting system for PA_ϵ.

the equations of PA_ϵ in Table 2.1, there are no rewrite rules corresponding to A1 and A2, because these equations have no clear direction. Also, having a rule for A1 would render the rewrite system non-terminating.

A common method for proving normalisation of a term rewriting system is to define a partial ordering on the operators and constants of the signature Σ, and then extend this ordering to terms over Σ. There are several ways to define this extension. For our purposes, the so called lexicographical variant of the recursive path ordering will suffice. The main reference for the following material is [2]. Other references are [3, 37, 34, 12].

Definition 2.3.7. Let $s, t \in T(\Sigma, V)$. We write $s >_{lpo} t$ if $s \to^+ t$, where $\to^+$ is the transitive closure of the reduction relation $\to$ defined by the rules RPO1-5 and LPO in Table 2.3. □

Theorem 2.3.8. (Kamin and Lévy [35]). Let (Σ, R) be a term rewriting system with finitely many rewrite rules and let $>$ be a well-founded partial ordering on Σ. If $s >_{lpo} t$ for each rewriting rule $s \to t \in R$, then the term rewriting system (Σ, R) is strongly normalising.

Proof. See [35]. □

The intuition behind this theorem is that if $x >_{lpo} y$, then y is a less complicated term that x, where we consider basic terms to be the simplest and general terms to be the most complicated. Thus, if all the rules can only make terms less complicated, we are bound to

RPO1. Mark head symbol ($k \geq 0$)

$$H(t_1, \ldots, t_k) \rightarrow H^*(t_1, \ldots, t_k)$$

RPO2. Make copies under smaller head symbol ($H > G$, $k \geq 0$)

$$H^*(t_1, \ldots, t_k) \rightarrow G(H^*(t_1, \ldots, t_k), \ldots, H^*(t_1, \ldots, t_k))$$

RPO3. Select argument ($k \geq 1$, $1 \leq i \leq k$)

$$H^*(t_1, \ldots, t_k) \rightarrow t_i$$

RPO4. Push $*$ down ($k \geq 1$, $l \geq 0$)

$$H^*(t_1, \ldots, G(s_1, \ldots, s_l), \ldots, t_k) \ \rightarrow H(t_1, \ldots, G^*(s_1, \ldots, s_l), \ldots, t_k)$$

RPO5. Handling contexts

$$s \rightarrow t \quad \Rightarrow \quad H(\ldots, s, \ldots) \rightarrow H(\ldots, t, \ldots)$$

LPO. Reduce ith argument ($k \geq 1$, $1 \leq i \leq k$, $l \geq 0$, H has lexicographical status wrt. the ith argument)

Let $t \equiv H^*(t_1, \ldots, t_{i-1}, G(s_1, \ldots, s_l), t_{i+1}, \ldots, t_k)$

then $t \rightarrow H(t, \ldots, t, G^*(s_1, \ldots, s_l), t, \ldots, t)$

Table 2.3: Reduction rules for the lexicographical variant of the recursive partial ordering.

eventually reach a term that can not be simplified.

Lemma 2.3.9. The term rewriting system for PA_ϵ in Table 2.2 is strongly normalizing.

Proof. According to theorem 2.3.8 it is sufficient to define a partial ordering on Σ_{PA_ϵ} and show that each rewriting rule satisfies the extension of the ordering to $T(\Sigma)$. We use the partial order $\| > \mathbin{\|\!\lfloor} > \surd > \cdot > + > \epsilon > \delta$. $\cdot$ has lexicographical status with regard to the first argument. Below, we illustrate the derivation for rewrite rules RA4 and RA5. The remaining derivations are given in Appendix B.1.

$$\begin{array}{rlll}
(x+y)\cdot z & >_{lpo} & (x+y) \cdot^* z & \text{RPO1} \\
 & >_{lpo} & (x+y) \cdot^* z + (x+y) \cdot^* z & \text{RPO2} \\
 & >_{lpo} & (x +^* y) \cdot z + (x +^* y) \cdot z & \text{RPO4, RPO5} \\
 & >_{lpo} & x \cdot z + y \cdot z & \text{RPO3, RPO5}
\end{array}$$

$$\begin{array}{rlll}
(x \cdot y)\cdot z & >_{lpo} & (x \cdot y) \cdot^* z & \text{RPO1} \\
 & >_{lpo} & (x \cdot^* y) \cdot ((x \cdot y) \cdot^* z) & \text{LPO} \\
 & >_{lpo} & x \cdot ((x \cdot^* y) \cdot z) & \text{RPO3, RPO5, RPO5} \\
 & >_{lpo} & x \cdot (y \cdot z) & \text{RPO3, RPO5}
\end{array}$$

Thus, the term rewriting system for PA_ϵ is strongly normalising. □

We are now ready to prove that every PA_ϵ term has an equivalent basic term.

Theorem 2.3.10. *For every PA_ϵ term, s, there is a corresponding basic term, t, such that $PA_\epsilon \vdash s = t$.*

Proof. By theorem 2.3.9 the term rewriting system for PA_ϵ is strongly normalizing. Thus, for every term t, there is a finite sequence of rewritings

$$t \to t_1 \to t_2 \to \cdots \to s$$

where s is in normal form.

We use a proof by contradiction to show that s cannot contain $\|$, $\lfloor\!\lfloor$ or $\surd$. Assume therefore, that s is in normal form and that $s = C[x \parallel y]$. But then the rewriting RF1 can be used, thus contradicting that s is in normal form. Now assume that s is in normal form and that $s = C[x \lfloor\!\lfloor y]$. Then there are three cases

- $x = u \lfloor\!\lfloor w$: in this case we can use the argument recursively to show that u or one of its sub-terms can be reduced by a rewrite rule. This line of reasoning is valid since we deal with finite terms.
- $x = \surd(u)$: in this case either x can be rewritten using one of RT1-4, or we can apply the whole argument to u to show that some sub-term of u can be rewritten.
- in all other cases one of the four rewrite rules RF2-4 may be applied to s, thus forming a contradiction.

Finally, we can use the same argument as above to show that if $s = C[\surd(x)]$ then either we can use one of the rewriting rules RT1-4 on s directly, or some sub-term of x can be reduced using a rewrite rule.

Thus, in all cases we have a contradiction and the theorem follows. □

2.3.2 Semantics of PA_ϵ

We now proceed to define a semantics for PA_ϵ. We use a structural operational semantics in the style of Plotkin [46]. Based on the semantics, we define a behavioural equivalence on PA_ϵ terms, called bisimulation equivalence. We then show that the quotient algebra of PA_ϵ terms under bisimulation equivalence is a model of the equational specification PA_ϵ, which implies soundness of the equations. Finally, we prove completeness of the equations.

A Plotkin-style operational semantics is defined using a set of derivation rules. For our purpose, the premises and conclusion of a derivation rule are formulas of either the form

$$x \xrightarrow{a} x'$$

or of the form

$$x \downarrow$$

Informally, the former formula means that process x can evolve into process x' by performing action a. The latter formula means that process x can terminate immediately and successfully.

A formula ϕ is provable from a set of deduction rules, if there is a rule

$$\frac{\varphi_1 \quad \varphi_2 \quad \cdots \quad \varphi_n}{\varphi}$$

such that there exists a substitution $\sigma : V \to T(\Sigma, V)$ satisfying $\sigma(\varphi) = \phi$ and if $\sigma(\varphi_i)$ is provable from the deduction rules for $i = 1, 2, \ldots, n$.

The deduction rules of the operational semantics for PA_ϵ are shown in Table 2.4.

$\frac{\Box}{a \xrightarrow{a} \epsilon}$	Act	$\frac{\Box}{\epsilon \downarrow}$	EpT
$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	Cho1	$\frac{x \downarrow}{x + y \downarrow}$	ChoT1
$\frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$	Cho2	$\frac{y \downarrow}{x + y \downarrow}$	ChoT2
$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	Seq1	$\frac{x \downarrow \quad y \downarrow}{x \cdot y \downarrow}$	SeqT
$\frac{x \downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$	Seq2	$\frac{x \downarrow \quad y \downarrow}{x \parallel y \downarrow}$	ParT
$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$	Par1	$\frac{x \downarrow \quad y \downarrow \quad x \xrightarrow{a} x'}{x \lfloor\!\lfloor y \downarrow}$	LmeT
$\frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$	Par2	$\frac{x \downarrow}{\surd(x) \downarrow}$	TerT
$\frac{x \xrightarrow{a} x'}{x \lfloor\!\lfloor y \xrightarrow{a} x' \parallel y}$	Lme		

Table 2.4: Structural operational semantics of PA_ϵ.

We seek a means of identifying terms that behave "in the same way". This form of behavioural equivalence is captured in the notion of *bisimulation*. Here, we use the strong formulation of bisimulation, due to Park [44].

Definition 2.3.11. (Bisimulation). *(Strong) Bisimulation equivalence*, $\sim \subseteq T(\Sigma) \times T(\Sigma)$, is the largest symmetric relation, such that for all $x, y \in T(\Sigma)$, if $x \sim y$, then the following conditions hold

1. $\forall x' \in T(\Sigma) : x \xrightarrow{a} x' \Rightarrow \exists y' \in T(\Sigma) : y \xrightarrow{a} y' \wedge x' \sim y'$
2. $x \downarrow \Leftrightarrow y \downarrow$

Two terms, x and y, are called *bisimilar*, if there exists a bisimulation relation, $\sim$, such that $x \sim y$. $\Box$

It follows from the definition that the bisimulation relation is an equivalence relation, since it is reflexive, symmetric and transitive.

The next step is to show that the bisimulation relation is a congruence. Having established this result, it is easy to show that the deduction system in Table 2.4 is a model of the equational specification PA_ϵ. This is the same as saying that the equations for PA_ϵ are sound.

Definition 2.3.12. (Congruence). Let R be an equivalence relation on $T(\Sigma)$. R is called a congruence if for all n-ary function symbols $f \in \Sigma$

$$x_1 R y_1 \wedge \ldots \wedge x_n R y_n \quad \Rightarrow \quad f(x_1, \ldots, x_n) R f(y_1, \ldots, y_n)$$

where $x_1, \ldots, x_n, y_1, \ldots, y_n \in T(\Sigma)$. $\Box$

Definition 2.3.13. (Baeten and Verhoef [1]). Let $T = (\Sigma, D)$ be a term deduction system and let $D = D(T_p, T_r)$, where T_p are the rules for the predicate (here $\downarrow$) and T_r are the rules for the relation (here $\xrightarrow{a}$). Let I and J be index sets of arbitrary cardinality, let $t_i, s_j, t \in T(\Sigma, V)$ for all $i \in I$ and $j \in J$, let $P_j, P \in T_p$ be predicate symbols for all $j \in J$, and let $R_i, R \in T_r$ be relation symbols for all $i \in I$. A deduction rule $d \in D$ is in *path formal* if it has one of the following four forms

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{f(x_1, \ldots, x_n) R t}$$

with $f \in \Sigma$ an n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{x R t}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{P f(x_1, \ldots, x_n)}$$

with $f \in \Sigma$ and n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables or

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\}}{P x}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables.

A term deduction system is said to be in *path* format if all its deduction rules are in *path* format. □

Theorem 2.3.14. (Baeten and Verhoef [1], Fokkink [14]) Let $T = (\Sigma, D)$ be a term deduction system. If T is in *path* format, then strong bisimulation equivalence is a congruence for all function symbols in Σ.

Proof. See [1]. □

Lemma 2.3.15. Let T_{PA_ϵ} be the term deduction system defined in Table 2.4. Then bisimulation equivalence is a congruence on the set of closed PA_ϵ terms.

Proof. We show that the deduction rules EpT and Cho1 are in path format. Writing $\downarrow$ in non-fix notation, deduction rule EpT can be rewritten to

$$\frac{\{\ \}}{\downarrow(\epsilon)}$$

which is in the third form in Definition 2.3.13. Similarly, Cho1 can be rewritten to

$$\frac{\{x \xrightarrow{a} x'\}}{x + y \xrightarrow{a} x'}$$

which is in the first form.

It is easily verified that the remaining deduction rules are also in *path* format, so the lemma follows from theorem 2.3.14. □

Having established that bisimulation equivalence is a congruence, we can construct the term quotient algebra $T(\Sigma_{PA_\epsilon})/\sim$. The reason we want to construct the quotient algebra is that it is an initial algebra, which is characterised by being the smallest algebra that captures the properties of the specification.

Recall that given an algebra A with signature Σ, the quotient algebra under the congruence $\equiv$, written $A/\equiv$ is defined as

- The carrier set of $A/\equiv$ consists of the equivalence classes of the carrier set of A under the equivalence relation $\equiv$, i.e. $|A/\equiv| = \{ [x]_\equiv \mid x \in |A| \}$, where $[x]_\equiv = \{ y \mid y \in |A| \wedge x \equiv y \}$.
- For each n-ary function symbol f_A in A, there is a corresponding n-ary function symbol $f_{A/\equiv}$ in $A/\equiv$, defined by

$$f_{A/\equiv}([x_1]_\equiv, \ldots, [x_n]_\equiv) = [f_A(x_1, \ldots, x_n)]_\equiv$$

Theorem 2.3.16. The set of closed PA_ϵ terms modulo bisimulation equivalence, notation $T(\Sigma_{PA_\epsilon})/\sim$, is a model of PA_ϵ.

Proof. Recall that a Σ-algebra, A, is a model of an equational specification (Σ, E), if $A \models E$, i.e. if every equation derivable from E holds in A.

Because bisimulation equivalence on PA_ϵ terms is a congruence by lemma 2.3.15, it is sufficient to separately verify the soundness of each axiom in E_{PA_ϵ}, i.e. to show if $PA_\epsilon \vdash x = y$, then $x \sim y$.

We illustrate the procedure by verifying equation A1. We have to show that there exists a bisimulation equivalence $\sim_*$ such that $x + y \sim_* y + x$. Let $\sim_*$ be defined as $\{ (x + y, y + x) \mid x, y \in T(\Sigma_{PA_\epsilon}) \} \cup \{ (x, x) \mid x \in T(\Sigma_{PA_\epsilon})\}$. Clearly, $\sim_*$ is symmetric. We now check the first bisimulation condition. $x + y$ can evolve only by following one of the two deduction rules Cho1 and Cho2. Suppose $x \xrightarrow{a} x'$, then $x + y \xrightarrow{a} x'$, but then we also have $y + x \xrightarrow{a} x'$. By definition $x' \sim_* x'$, so the condition is satisfied in this case. The symmetric case $y \xrightarrow{a} y'$ follows from the same argument. Next, the second bisimulation condition must be checked. Suppose $x \downarrow$, then by ChoT1 $x + y \downarrow$. But in that case by ChoT2 $y + x \downarrow$. Again the symmetric case $y \downarrow$ follows immediately.

The above procedure can be applied to the remaining equations to show that equal terms are bisimilar. Thus, the theorem follows. □

Finally, we show that PA_ϵ is a complete axiomatisation of the set of closed terms modulo bisimulation equivalence, i.e. whenever $x \sim y$, then $PA_\epsilon \vdash x = y$.

Theorem 2.3.17. The axiom system PA_ϵ is a complete axiomatisation of the set of closed terms modulo bisimulation equivalence.

Proof. Due to theorem 2.3.16 and 2.3.10 it suffices to prove the theorem for basic terms. The proof for basic terms is given in [2]. □

2.3.3 The Process Algebra PAc_ϵ

The process algebra PA_ϵ introduced in the previous section is sufficiently expressive to define the semantics of Message Sequence Charts. However, the extension to Live Sequence Charts calls for the introduction of an additional operator.

In this subsection we introduce the extended process algebra, called PAc_ϵ, for Process Algebra with conditional behaviour. PAc_ϵ is a conservative extension of PA_ϵ, meaning that the theory of PA_ϵ also holds in PAc_ϵ. We give an axiom system and a model of PAc_ϵ, and show that the axiom system is sound and complete. Our task now is considerably easier, since most of the results for PA_ϵ can be directly transferred to PAc_ϵ.

The signature of PAc_ϵ, Σ_{PAc_ϵ}, consists of

1. two special constants δ and ϵ
2. a set of unspecified constants A, for which $\{\delta, \epsilon\} \cap A = \emptyset$

$\epsilon \triangleright x = x$	C1
$\delta \triangleright x = \epsilon$	C2
$x + y \triangleright z = (x \triangleright z) + (y \triangleright z)$	C3
$a \cdot x \triangleright y = a \cdot (x \triangleright y) + \bar{a}, \quad$ where $\bar{a} \in A \setminus \{a\}$	C4

Table 2.5: Additional equations of PAc_ϵ.

3. the unary operator $\surd$
4. the binary operators $+$, $\cdot$, $\|$, $\llfloor$ and $\triangleright$.

The binary operator $\triangleright$ is the *conditional behaviour* operator. The conditional behaviour of processes x and y is the process that either terminates successfully or executes x followed by y. The other operators and constants have the same meaning as they do in PA_ϵ.

Table 2.5 lists the additional equations E_{PAc_ϵ} for $a \in A$ and $x, y, z \in V$.

$\epsilon \triangleright x \rightarrow x$	RC1
$\delta \triangleright x \rightarrow \epsilon$	RC2
$x + y \triangleright z \rightarrow x \triangleright z + y \triangleright z$	RC3
$a \cdot x \triangleright y \rightarrow a \cdot (x \triangleright y) + \bar{a}$	RC4
$a \triangleright y \rightarrow a \cdot y + \bar{a}$	RC4'

Table 2.6: Additional term rewriting rules for PAc_ϵ.

Theorem 2.3.18. The term rewriting system for PAc_ϵ in Table 2.6 is strongly normalizing.

Proof. The proof is based on the proof of theorem 2.3.9. We add the conditional operator to the partial ordering: $\triangleright > \| > \llfloor > \surd > \cdot > + > \epsilon > \delta$. We now show that the additional rewrite rules for PAc_ϵ satisfy the extension of the partial ordering to terms.

$$\begin{aligned} \epsilon \triangleright x \;&>_{lpo}\; \epsilon \triangleright^* x && \text{RPO1} \\ &>_{lpo}\; \epsilon && \text{RPO3} \end{aligned}$$

$$\begin{aligned} \delta \triangleright x \;&>_{lpo}\; \delta \triangleright^* x && \text{RPO1} \\ &>_{lpo}\; \epsilon && \text{RPO2} \end{aligned}$$

$$\begin{aligned} x + y \triangleright z \;&>_{lpo}\; x + y \triangleright^* z && \text{RPO1} \\ &>_{lpo}\; (x + y \triangleright^* z) + (x + y \triangleright^* z) && \text{RPO2} \\ &>_{lpo}\; (x +^* y \triangleright z) + (x +^* y \triangleright z) && \text{RPO4, RPO5} \\ &>_{lpo}\; (x \triangleright z) + (y \triangleright z) && \text{RPO4, RPO5} \end{aligned}$$

$$\begin{aligned}
a \cdot x \rhd y &>_{lpo} a \cdot x \rhd^* y && \text{RPO1} \\
&>_{lpo} (a \cdot x \rhd^* y) + (a \cdot x \rhd^* y) && \text{RPO2} \\
&>_{lpo} (a \cdot x \rhd^* y) + \bar{a} && \text{RPO2, RPO5} \\
&>_{lpo} (a \cdot x \rhd^* y) \cdot (a \cdot x \rhd^* y) + \bar{a} && \text{RPO2} \\
&>_{lpo} (a \cdot^* x) \cdot (x \rhd y) + \bar{a} && \text{RPO1, RPO3, RPO5} \\
&>_{lpo} a \cdot (x \rhd y) + \bar{a} && \text{RPO1, RPO3}
\end{aligned}$$

$$\begin{aligned}
a \rhd y &>_{lpo} a \rhd^* y && \text{RPO1} \\
&>_{lpo} (a \rhd^* y) + (a \rhd^* y) && \text{RPO2} \\
&>_{lpo} (a \rhd^* y) + \bar{a} && \text{RPO2, RPO5} \\
&>_{lpo} (a \rhd^* y) \cdot (a \rhd^* y) + \bar{a} && \text{RPO1, RPO3} \\
&>_{lpo} a \cdot y + \bar{a} && \text{RPO3, RPO5}
\end{aligned}$$

Thus, the rewrite system for PAc_ϵ is strongly normalizing. □

In theorem 2.3.10 we showed that every PA_ϵ term has an equivalent basic term. With the definition of a basic term from definition 2.3.4, we have the similar result for PAc_ϵ.

Theorem 2.3.19. *For every PAc_ϵ term, s, there is a corresponding basic term, t, such that $PA_\epsilon \vdash s = t$.*

Proof. We have already shown that the subset of PAc_ϵ that corresponds to PA_ϵ can be reduced to basic terms. Thus, we only need to show that terms with the conditional operator can be reduced to basic terms.

Because the term rewriting system for PAc_ϵ is strongly normalizing by theorem 2.3.18, then for every term t, there exists a finite sequence of rewritings

$$t \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow s$$

where s is in normal form.

We use a proof by contradiction to show that s cannot contain $\rhd$. Assume therefore, that s is in normal form and that $s = C[x \rhd y]$.

If $x = C[u \rhd w]$ then the argument can be applied recursively to show that $u \rhd w$ or one of u's sub-terms can be reduced, thus contradicting that s is in normal form. Otherwise, there are five possibilities

- $x = \epsilon$: then s can be reduced by RC1.
- $x = \delta$: then s can be reduced by RC2.
- $x = u + w$: then s can be reduced by RC4.
- $x = a \cdot x'$: then s can be reduced by RC5.
- $x = a$: then s can be reduced by RC5'.

All cases contradict that s is in normal form. Thus, every PAc_ϵ term can be reduced to an equivalent basic term. □

2.3.4 Semantics for PAc_ϵ

The additional semantical rules for PAc_ϵ are shown in Table 2.7.

In order to prove that bisimulation is a congruence on the set of closed PAc_ϵ terms we need to introduce a generalisation of the *path* format used in the previous section. The

$$\frac{x \xrightarrow{a} x'}{x \triangleright y \xrightarrow{\bar{a}} \epsilon}\ \text{Con1} \qquad \frac{x \downarrow \quad y \downarrow}{x \triangleright y \downarrow}\ \text{ConT1}$$

$$\frac{x \xrightarrow{a} x'}{x \triangleright y \xrightarrow{a} x' \triangleright y}\ \text{Con2} \qquad \frac{x \not\downarrow}{x \triangleright y \downarrow}\ \text{ConT2}$$

$$\frac{x \xrightarrow{a} x'}{\epsilon \triangleright x \xrightarrow{a} x'}\ \text{Con3}$$

Table 2.7: Extra semantical rules for PAc_ϵ.

generalisation is known as *panth* format for "predicates and *ntyft/ntyxt* hybrid format". It generalises the *path* format by allowing negative premises in the deduction rules. It is also a generalisation of the *ntyft/ntyxt* of Groote [19], which in turn along with the *path* format is a generalisation of the *tyft/tyxt* format of Groote and Vaandrager [20]. The names of these formats are derived from the format of the premises and conclusion of the deduction rules, see Verhoef [56] for an explanation.

The reference for the following material is Verhoef [56].

Definition 2.3.20. (Verhoef [56]). Let $T = (\Sigma, D)$ be a term deduction system and let $D = D(T_p, T_r)$, where T_p is the set of predicate symbols and T_r is the set of relation symbols. Let I, J, K and L be index sets of arbitrary cardinality, let $s_j, t_i, u_l, v_k, t \in T(\Sigma, V)$ for all $i \in I$, $j \in J$, $k \in K$ and $l \in L$, and let $P_j, P \in T_p$ be predicate symbols for all $j \in J$, and let $R_i, R \in T_r$ be relation symbols for all $i \in I$. A deduction rule $d \in D$ is in *panth format* if it has one of the following four forms

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{f(x_1, \ldots, x_n) R t}$$

with $f \in \Sigma$ an n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{x R t}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables;

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{P f(x_1, \ldots, x_n)}$$

with $f \in \Sigma$ and n-ary function symbol, $X = \{x_1, \ldots, x_n\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables or

$$\frac{\{P_j s_j \mid j \in J\} \cup \{t_i R_i y_i \mid i \in I\} \cup \{\neg P_l u_l \mid l \in L\} \cup \{v_k \neg R_k \mid k \in K\}}{P x}$$

with $X = \{x\}, Y = \{y_i \mid i \in I\}$, and $X \cup Y \subseteq V$ a set of distinct variables.

A term deduction system is said to be in *panth* format if all its deduction rules are in *panth* format. □

Before we can introduce the congruence theorem for the *panth* format we need to define some additional notions.

Definition 2.3.21. Let $T = (\Sigma, D)$ be a term deduction system. The formula dependency graph G of T is a labelled directed graph with the positive formulas of D as nodes. Let $PF(H)$ denoted the set of all positive formulas in H and let $NF(H)$ denote all the negative formulas in H, then for all deduction rules $H/C \in D$ and for all closed substitutions σ we have the following edges in G:

- for all $h \in PF(H)$ there is an edge $\sigma(h) \xrightarrow{p} \sigma(C)$;
- for all $s \neg R \in NF(H)$ there is for all $t \in T(\Sigma)$ an edge $\sigma(sRt) \xrightarrow{n} \sigma(C)$;
- for all $\neg Ps \in NF(H)$ there is an edge $\sigma(Ps) \xrightarrow{n} \sigma(C)$.

An edge labelled with a p is called positive and an edge labelled with an n is called negative. A set of edges is called positive if all its elements are positive and negative if the edges are all negative. □

Definition 2.3.22. A term deduction system is stratifiable if there is no node in its formula dependency graph that is the start of a backward chain of edges containing an infinite negative subset. □

Definition 2.3.23. Let $T = (\Sigma, D)$ be a term deduction system and let F be a set of formulas. The variable dependency graph of F is a directed graph with the variables occurring in F as its nodes. The edge $x \rightarrow y$ is an edge of the variable dependency graph if and only if there is a positive relation $tRs \in F$ with $x \in vars(t)$ and $y \in vars(s)$.

The set F is called well-founded if any backward chain of edges in its variable dependency graph is finite. A deduction rule is called well-founded if its set of premises is so. A term deduction system is called well-founded if all its deduction rules are well-founded. □

We are now ready to formulate the main result of Verhoef [56].

Theorem 2.3.24. (Verhoef [56]). Let $T = (\Sigma, D)$ be a well-founded stratifiable term deduction system in *panth* format, then strong bisimulation is a congruence for all function symbols in Σ.

Proof. See [56]. □

Lemma 2.3.25. Let $T = (\Sigma_{PAc_\epsilon}, D)$ be the term deduction system in Table 2.7, then strong bisimulation is a congruence on the set of closed PAc_ϵ terms.

Proof. The proof relies on theorem 2.3.24.

First, we must check that the term deduction system is well-founded. No variable occurs more than once in the set of premises for any of the deduction rules, so it is clear that there are no cycles in the variable dependency graph. Hence, the term deduction system is well-founded.

Next, we must show that the term deduction system is stratifiable. We use proof by contradiction. Assume the term deduction is not stratifiable. Then, there is some backward chain of edges in the formula dependency graph that contains an infinite negative subset of edges. The only negative edge in the graph is the one that stems from ConT2. Thus, there must be a cycle containing the edge $\sigma(x\downarrow) \xrightarrow{n} \sigma(x \rhd y\downarrow)$. This cycle must also contain at least one edge originating at the node $\sigma(x \rhd y\downarrow)$ and terminating at some node, Z, see Figure 2.18.

$$\begin{tikzcd} \arrow[r, dotted] & \sigma(x\downarrow) \arrow[d, "n"] \\ Z \arrow[u, dash, dotted] & \sigma(x \rhd y\downarrow) \arrow[l] \end{tikzcd}$$

Figure 2.18: Illustration for proof of congruence.

By the definition of the formula dependency graph, the edge $\sigma(x \rhd y\downarrow) \rightarrow Z$ can only be in the graph because there is a deduction rule with $x \rhd y\downarrow$ as one of its premises. However, there is no such rule, and we have a contradiction. Therefore, the term deduction system is stratifiable.

Finally, we must verify that each of the deduction rules are in *panth* format. Since any rule that is in *path* format is also in *panth* format, we only need to check the additional rules for PAc_ϵ, since the remaining rules were shown to be in *path* format in the proof for lemma 2.3.15. The rule Con1 can be trivially rewritten to

$$\frac{\{x \overset{a}{\rightarrow} x'\}}{x \rhd y \overset{\bar{a}}{\rightarrow} \epsilon}$$

which is in the first *panth* form. The rule ConT2 can similarly be rewritten to

$$\frac{\{\neg \downarrow (x)\}}{\downarrow (x \rhd y)}$$

which is in the third *panth* form. The remaining three rules are easily shown to also be in *panth* format.

Thus, all the conditions of theorem 2.3.24 are satisfied and the result follows. □

Theorem 2.3.26. The set of closed PAc_ϵ terms modulo bisimulation equivalence, notation $T(\Sigma_{PAc_\epsilon})/\sim$, is a model of PAc_ϵ.

Proof. Recalling the proof for theorem 2.3.16 we have to show that for each of the equations in E_{PAc_ϵ}, $PAc_\epsilon \vdash x = y$ implies the existence of a bisimulation, $\sim$, such that $x \sim y$.

We give the proof for axiom C4. Let $\sim_*$ be defined by $\{\ (a \cdot x \rhd y, a \cdot (x \rhd y) + \bar{a}) \mid x, y \in T(\Sigma_{PAc_\epsilon}), a \in A\ \} \cup \{\ (x, y) \mid x, y \in T(\Sigma_{PAc_\epsilon})\ \}$. Clearly, $\sim_*$ is symmetric. We first check the termination condition. By ConT1 $a \cdot x \rhd y \downarrow$, since $a \cdot x \not\downarrow$. Similarly, $a \cdot (x \rhd y) + \bar{a} \downarrow$, since $\bar{a} \not\downarrow$ (and actually also $a \cdot (x \rhd y) \not\downarrow$). Thus, the termination condition for bisimulation equivalence is satisfied.

Now, we check the first bisimulation condition. There are two ways $a \cdot x \rhd y$ can evolve:

- $a \cdot x \rhd y \overset{\bar{a}}{\rightarrow} \epsilon$: then we get $a \cdot (x \rhd y) + \bar{a} \overset{\bar{a}}{\rightarrow} \epsilon$ and since $\epsilon \sim_* \epsilon$ by definition, the bisimulation condition is satisfied in this case.
- $a \cdot x \rhd y \overset{a}{\rightarrow} x \rhd y$: similarly, $a \cdot (x \rhd y) + \bar{a} \overset{a}{\rightarrow} x \rhd y$ and again $x \rhd y \sim_* x \rhd y$, so the bisimulation condition is satisfied.

The symmetric case for evolutions of $a \cdot (x \rhd y) + \bar{a}$ is entirely analogous.

The remaining axioms can be checked with the same technique. □

We now come to the final result showing that the axiom system for PAc_ϵ is both sound and complete.

Theorem 2.3.27. The axiom system PAc_ϵ is a complete axiomatisation of the set of closed PAc_ϵ terms modulo bisimulation equivalence.

Proof. Due to theorems 2.3.26 and 2.3.19 it suffices to prove the theorem for basic terms. The proof for basic terms is given in [2]. □

2.4 Algebraic Semantics of Live Sequence Charts

In this section a subset of Live Sequence Charts are given an algebraic semantics using the process algebra PAc_ϵ from the previous section. The presentation here is adapted from the description of the semantics of MSC given by Mauw and Reniers [43].

2.4.1 Textual Syntax of Live Sequence Charts

We give a textual syntax for LSC. The textual syntax is used to define the semantics in the next subsection. The textual syntax is presented as an EBNF grammar below. The nonterminals *lscid*, *msgid* and *inst name* are further unspecified identifiers. The nonterminal *cond* represents a further unspecified conditional expression.

$\langle chart\rangle$ ::= **lsc** $\langle lscid\rangle$; $\langle inst\ def\ list\rangle$ **endlsc**

$\langle inst\ def\ list\rangle$ ::= $\langle inst\ def\rangle$ $\langle inst\ def\ list\rangle$ | $\langle\rangle$

$\langle inst\ def\rangle$ ::= **instance** $\langle inst\ name\rangle$ $\langle prechart\rangle$ $\langle body\rangle$ **endinstance**

$\langle prechart\rangle$ ::= **prechart** $\langle location\rangle$ **endprechart**

$\langle body\rangle$ ::= **body** $\langle location\rangle$ **endbody**

$\langle location\rangle$ ::= **hot** $\langle event\rangle$; $\langle location\rangle$ | **cold** $\langle event\rangle$; $\langle location\rangle$ | $\langle\rangle$

$\langle event\rangle$ ::= $\langle input\rangle$ | $\langle output\rangle$ | $\langle condition\rangle$ | $\langle coregion\rangle$

$\langle input\rangle$ ::= **in** $\langle msgid\rangle$ **from** $\langle address\rangle$ $\langle mode\rangle$

$\langle output\rangle$ ::= **out** $\langle msgid\rangle$ **to** $\langle address\rangle$ $\langle mode\rangle$

$\langle condition\rangle$ ::= **hotcondition** $\langle cond\rangle$ | **coldcondition** $\langle cond\rangle$

$\langle coregion\rangle$ ::= **concurrent** $\langle coeventlist\rangle$ **endconcurrent**

$\langle coeventlist\rangle$ ::= $\langle input\rangle$ $\langle coeventlist\rangle$ | $\langle output\rangle$ $\langle coeventlist\rangle$ | $\langle\rangle$

$\langle address\rangle$ ::= $\langle inst\ name\rangle$ | **env**

$\langle mode\rangle$ ::= **sync** | **async**

Table 2.8: EBNF grammar for the textual syntax of Live Sequence Charts.

We do not explain the mapping from an LSC to the textual syntax further as this is straightforward. The example in Section 2.5 illustrates the mapping.

2.4.2 Semantics of Live Sequence Charts

In order to define the semantics of the subset of LSC, we instantiate the process algebra PAc_ϵ by specifying the set of atomic actions. We assume a set, A_o, of atomic actions representing asynchronous (*out*) and synchronous (*outs*) message output

$$\begin{aligned} A_o =& \{out(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\}\cup \\ & \{outs(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\} \end{aligned}$$

Similarly, we assume a set, A_i, of atomic actions representing asynchronous (*in*) and synchronous (*ins*) message input

$$\begin{aligned} A_i =& \{in(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\}\cup \\ & \{ins(i,j,m) \mid i,j \in \mathcal{L}(\langle inst\ name\rangle), m \in \mathcal{L}(\langle msgid\rangle)\} \end{aligned}$$

Conditions are also viewed as actions, so there is a set of atomic actions representing hot conditions

$$A_{hc} = \{hotcond(c) \mid c \in \mathcal{L}(\langle cond\rangle)\ \}$$

$$\begin{array}{ll}
\lambda_{M,C}(\epsilon) = \epsilon & \text{if } M = \emptyset \\
\lambda_{M,C}(\epsilon) = \delta & \text{if } M \neq \emptyset \\
\lambda_{M,C}(\delta) = \delta & \\
\lambda_{M,C}(a \cdot x) = \delta & \text{if } a \notin A_o \cup A_i \text{ and } C \neq \emptyset \\
\lambda_{M,C}(a \cdot x) = a \cdot \lambda_{M,\emptyset}(x) & \text{if } a \notin A_o \cup A_i \text{ and } C = \emptyset \\
\lambda_{M,C}(\mathit{out}(i, \mathit{env}, m) \cdot x) = \delta & \text{if } C \neq \emptyset \\
\lambda_{M,C}(\mathit{out}(i, \mathit{env}, m) \cdot x) = \mathit{out}(i, \mathit{env}, m) \cdot \lambda_{M,\emptyset}(x) & \text{if } C = \emptyset \\
\lambda_{M,C}(\mathit{out}(i, j, m) \cdot x) = \delta & \text{if } m \in M \text{ or } C \neq \emptyset \\
\lambda_{M,C}(\mathit{out}(i, j, m) \cdot x) = \mathit{out}(i, j, m) \cdot \lambda_{M\cup\{m\},\emptyset}(x) & \text{if } m \notin M \text{ and } C = \emptyset \\
\lambda_{M,C}(\mathit{outs}(i, \mathit{env}, m) \cdot x) = \delta & \text{if } C \neq \emptyset \\
\lambda_{M,C}(\mathit{outs}(i, \mathit{env}, m) \cdot x) = \mathit{outs}(i, \mathit{env}, m) \cdot \lambda_{M,\emptyset}(x) & \text{if } C = \emptyset \\
\lambda_{M,C}(\mathit{outs}(i, j, m) \cdot x) = \delta & \text{if } m \in M \text{ or } C \neq \emptyset \\
\lambda_{M,C}(\mathit{outs}(i, j, m) \cdot x) = \mathit{outs}(i, j, m) \cdot \lambda_{M\cup\{m\},\{m\}}(x) & \text{if } m \notin M \text{ and } C \neq \emptyset \\
\lambda_{M,C}(\mathit{in}(\mathit{env}, j, m) \cdot x) = \delta & \text{if } C \neq \emptyset \\
\lambda_{M,C}(\mathit{in}(\mathit{env}, j, m) \cdot x) = \mathit{in}(\mathit{env}, j, m) \cdot \lambda_{M,\emptyset}(x) & \text{if } C = \emptyset \\
\lambda_{M,C}(\mathit{in}(i, j, m) \cdot x) = \mathit{in}(i, j, m) \cdot \lambda_{M\setminus\{m\},\emptyset}(x) & \text{if } m \in M \text{ and } C = \emptyset \\
\lambda_{M,C}(\mathit{in}(i, j, m) \cdot x) = \delta & \text{if } m \notin M \text{ or } C \neq \emptyset \\
\lambda_{M,C}(\mathit{ins}(\mathit{env}, j, m) \cdot x) = \delta & \text{if } C \neq \emptyset \\
\lambda_{M,C}(\mathit{ins}(\mathit{env}, j, m) \cdot x) = \mathit{ins}(\mathit{env}, j, m) \cdot \lambda_{M,\emptyset}(x) & \text{if } C = \emptyset \\
\lambda_{M,C}(\mathit{ins}(i, j, m) \cdot x) = \mathit{ins}(i, j, m) \cdot \lambda_{M\setminus\{m\},\emptyset}(x) & \text{if } m \in M \text{ and } C = \{m\} \\
\lambda_{M,C}(\mathit{ins}(i, j, m) \cdot x) = \delta & \text{if } m \notin M \text{ or } C \neq \{m\} \\
\lambda_{M,C}(x + y) = \lambda_{M,C}(x) + \lambda_{M,C}(y) & \\
\lambda_{M,C}(x \rhd y) = \lambda_M(x) \rhd \lambda_M(y) &
\end{array}$$

Table 2.9: Definition of the state operator $\lambda_{M,C}$.

and a set of atomic actions representing cold conditions

$$A_{cc} = \{\mathit{coldcond}(c) \mid c \in \mathcal{L}(\langle \mathit{cond} \rangle)\ \}$$

The set of atomic actions, A, of the instantiated process algebra is then

$$A = A_o \cup A_i \cup A_{hc} \cup A_{cc}$$

The process algebra PAc_ϵ defined above does not place any constraints on the order of atomic events. In expressing the semantics of LSC the constraint that message input must follow the corresponding message output has to be expressed. To do this, the state operator $\lambda_{M,C}$ is introduced. It is an instance of the general state operator [3].

For $M \subseteq \mathcal{L}(\langle \mathit{msgid} \rangle)$, $x, y \in V$, $a \in A$, $i, j \in \mathcal{L}(\langle \mathit{inst\ name} \rangle)$ and $m \in \mathcal{L}(\langle \mathit{msgid} \rangle)$, the state operator is defined by the equations in Table 2.9. The subscript M records the message identifiers of messages that have been output, but not yet input. The subscript C records the message identifiers of those synchronous messages that have been output, but not yet input. If C is non-empty and the next event is not the corresponding input event, deadlock occurs. This ensures that no other events can come between the output and input of a synchrous message.

The instantiated process algebra with $\lambda_{M,C}$ will be referred to as PA_{LSC} in the following.

The semantics of LSCs will be defined by semantic functions over the syntactical categories of the textual syntax of LSCs. If $\langle \mathit{cat} \rangle$ denotes a syntactical category (non-terminal) in the ENBF grammar, then $\mathcal{L}(\langle \mathit{cat} \rangle)$ denotes the language of text strings derivable from that syntactical category. The notation $\mathcal{P}X$ denotes the powerset of the set X.

The semantic function for LSCs, $S_{LSC}[\![\,\cdot\,]\!] : \mathcal{L}(\langle \mathit{chart}\rangle) \to T(\Sigma_{PA_{LSC}})$, is defined by

$$S_{LSC}[\![ch]\!] = \lambda_{\emptyset,\emptyset}\left(\left(\|_{i\in \mathit{Inst}_c(ch)}\; S_{\mathit{instpc}}[\![i]\!]\right) \triangleright \left(\|_{i\in \mathit{Inst}_c(ch)}\; S_{\mathit{instbody}}[\![i]\!]\right)\right)$$

where $\mathit{Inst}_c : \mathcal{L}(\langle \mathit{chart}\rangle) \to \mathcal{P}(\mathcal{L}(\langle \mathit{inst\ def}\rangle))$ is the set of instance definitions in the chart. It is defined by

$$\mathit{Inst}_c(\mathbf{lsc}\ \langle \mathit{lscid}\rangle\ ;\ \langle \mathit{inst\ def\ list}\rangle\ \mathbf{endlsc}) = \mathit{Inst}_{\mathit{idl}}(\langle \mathit{inst\ def\ list}\rangle)$$

where in turn $\mathit{Inst}_{\mathit{idl}} : \mathcal{L}(\langle \mathit{inst\ def\ list}\rangle) \to \mathcal{P}(\mathcal{L}(\langle \mathit{inst\ def}\rangle))$ is defined by

$$\mathit{Inst}_{\mathit{idl}}(\langle\rangle) = \emptyset$$
$$\mathit{Inst}_{\mathit{idl}}(\langle \mathit{inst\ def}\rangle\ \langle \mathit{inst\ def\ list}\rangle) = \{\langle \mathit{inst\ def}\rangle\} \cup \mathit{Inst}_{\mathit{idl}}(\langle \mathit{inst\ def\ list}\rangle)$$

The semantic function for instance precharts, $S_{\mathit{instpc}}[\![\,\cdot\,]\!] : \mathcal{L}(\langle \mathit{inst\ def}\rangle) \to T(\Sigma_{PA_{LSC}})$, is defined by

$$S_{\mathit{instpc}}[\![\mathbf{instance}\ \langle \mathit{inst\ name}\rangle\ \langle \mathit{prechart}\rangle\ \langle \mathit{body}\rangle\ \mathbf{endinstance}]\!] = S^{\langle \mathit{inst\ name}\rangle}_{\mathit{prechart}}[\![\,\langle \mathit{prechart}\rangle\,]\!]$$

The semantic function for instance bodies, $S_{\mathit{instbody}}[\![\,\cdot\,]\!] : \mathcal{L}(\langle \mathit{inst\ def}\rangle) \to T(\Sigma_{PA_{LSC}})$, is defined by

$$S_{\mathit{instbody}}[\![\mathbf{instance}\ \langle \mathit{inst\ name}\rangle\ \langle \mathit{prechart}\rangle\ \langle \mathit{body}\rangle\ \mathbf{endinstance}]\!] = S^{\langle \mathit{inst\ name}\rangle}_{\mathit{body}}[\![\,\langle \mathit{body}\rangle\,]\!]$$

For $\mathit{iid} \in \mathcal{L}(\langle \mathit{inst\ name}\rangle$ the semantic function for precharts, $S^{\mathit{iid}}_{\mathit{body}}[\![\,\cdot\,]\!] : \mathcal{L}(\langle \mathit{prechart}\rangle) \to T(\Sigma_{PA_{LSC}})$, is defined by

$$S^{\mathit{iid}}_{\mathit{prechart}}[\![\mathbf{prechart}\ \langle \mathit{location}\rangle\ \mathbf{endprechart}]\!] = S^{\mathit{iid}}_{\mathit{location}}[\![\,\langle \mathit{location}\rangle\,]\!]$$

For $\mathit{iid} \in \mathcal{L}(\langle \mathit{inst\ name}\rangle$ the semantic function for instance bodies, $S^{\mathit{iid}}_{\mathit{body}}[\![\,\cdot\,]\!] : \mathcal{L}(\langle \mathit{body}\rangle) \to T(\Sigma_{PA_{LSC}})$, is defined by

$$S^{\mathit{iid}}_{\mathit{body}}[\![\mathbf{body}\ \langle \mathit{location}\rangle\ \mathbf{endbody}]\!] = S^{\mathit{iid}}_{\mathit{location}}[\![\,\langle \mathit{location}\rangle\,]\!]$$

For $\mathit{iid} \in \mathcal{L}(\langle \mathit{inst\ name}\rangle$ the semantic function for event lists, $S^{\mathit{iid}}_{\mathit{location}}[\![\cdot]\!] : \mathcal{L}(\langle \mathit{location}\rangle) \to T(\Sigma_{PA_{LSC}})$, is defined by

$$S^{\mathit{iid}}_{\mathit{location}}[\![\,\langle\rangle\,]\!] = \epsilon$$
$$S^{\mathit{iid}}_{\mathit{location}}[\![\mathbf{hot}\,\langle \mathit{event}\rangle\ ;\ \langle \mathit{location}\rangle\,]\!] = S^{\mathit{iid}}_{\mathit{event}}[\![\,\langle \mathit{event}\rangle\,]\!] \cdot S^{\mathit{iid}}_{\mathit{location}}[\![\,\langle \mathit{location}\rangle\,]\!]$$
$$S^{\mathit{iid}}_{\mathit{location}}[\![\mathbf{cold}\,\langle \mathit{event}\rangle\ ;\ \langle \mathit{location}\rangle\,]\!] = \epsilon + \left(S^{\mathit{iid}}_{\mathit{event}}[\![\,\langle \mathit{event}\rangle\,]\!] \cdot S^{\mathit{iid}}_{\mathit{location}}[\![\,\langle \mathit{location}\rangle\,]\!]\right)$$

For $\mathit{iid} \in \mathcal{L}(\langle \mathit{inst\ name}\rangle$ the semantic function for events, $S^{\mathit{iid}}_{\mathit{event}}[\![\,\cdot\,]\!] : \mathcal{L}(\langle \mathit{event}\rangle) \to T(\Sigma_{PA_{LSC}})$, is defined by

$$S^{\mathit{iid}}_{\mathit{event}}[\![\mathbf{out}\ \langle \mathit{msgid}\rangle\ \mathbf{to}\ \langle \mathit{address}\rangle\ \mathbf{async}]\!] = \mathit{out}(\mathit{iid}, \langle \mathit{address}\rangle, \langle \mathit{msgid}\rangle)$$
$$S^{\mathit{iid}}_{\mathit{event}}[\![\mathbf{out}\ \langle \mathit{msgid}\rangle\ \mathbf{to}\ \langle \mathit{address}\rangle\ \mathbf{sync}]\!] = \mathit{outs}(\mathit{iid}, \langle \mathit{address}\rangle, \langle \mathit{msgid}\rangle)$$
$$S^{\mathit{iid}}_{\mathit{event}}[\![\mathbf{in}\ \langle \mathit{msgid}\rangle\ \mathbf{from}\ \langle \mathit{address}\rangle\ \mathbf{async}]\!] = \mathit{in}(\langle \mathit{address}\rangle, \mathit{iid}, \langle \mathit{msgid}\rangle)$$
$$S^{\mathit{iid}}_{\mathit{event}}[\![\mathbf{in}\ \langle \mathit{msgid}\rangle\ \mathbf{from}\ \langle \mathit{address}\rangle\ \mathbf{sync}]\!] = \mathit{ins}(\langle \mathit{address}\rangle, \mathit{iid}, \langle \mathit{msgid}\rangle)$$
$$S^{\mathit{iid}}_{\mathit{event}}[\![\mathbf{hotcondition}\ \langle \mathit{cond}\rangle\,]\!] = \mathit{hotcond}(\langle \mathit{cond}\rangle)$$
$$S^{\mathit{iid}}_{\mathit{event}}[\![\mathbf{coldcondition}\ \langle \mathit{cond}\rangle\,]\!] = \mathit{coldcond}(\langle \mathit{cond}\rangle)$$
$$S^{\mathit{iid}}_{\mathit{event}}[\![\mathbf{concurrent}\ \langle \mathit{coeventlist}\rangle\ \mathbf{endconcurrent}]\!] = \|_{e\in \mathit{CoEvents}(\langle \mathit{coeventlist}\rangle)}\; S^{\mathit{iid}}_{\mathit{event}}[\![e]\!]$$

where $CoEvents : \mathcal{L}(\langle eventlist \rangle) \rightarrow \mathcal{P}(\mathcal{L}(\langle event \rangle))$ is defined by

$$CoEvents(\langle\rangle) = \emptyset$$
$$CoEvents(\langle event \rangle \ \langle eventlist \rangle) = \{\langle event \rangle\} \cup CoEvents(\langle eventlist \rangle)$$

2.5 Live Sequence Chart Example

We conclude this chapter with an example illustrating the process of deriving a PA_{LSC} term from an LSC diagram.

Figure 2.19 shows an example LSC with three instances. The first step is to convert the graphical syntax into the textual syntax. The result is shown below.

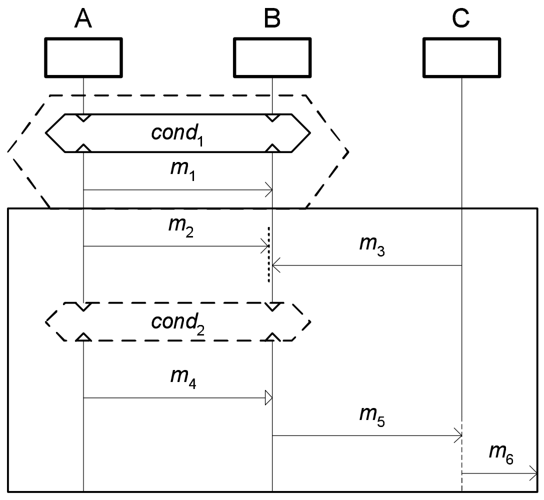


Figure 2.19: Example Live Sequence Chart.

```
lsc Example;
   instance A
      prechart
         hot hotcondition($cond_1$) ;
         hot out $m_1$ to $B$ async ;
      end prechart body hot out $m_2$ to $B$ async ;
         hot coldcondition($cond_2$) ;
         hot out $m_4$ to $B$ sync ;
      end body
   end instance
   instance B
      prechart
         hot hotcondition($cond_1$) ;
         hot in $m_1$ from $A$ async ;
      end prechart
      body
         hot concurrent
            in $m_2$ from $A$ async ;
            in $m_3$ from $C$ async ;
         end concurrent ;
         hot coldcondition($cond_2$) ;
         hot in $m_4$ from $A$ sync ;
```

```
        hot out $m_5$ to $C$ async ;
      end body
   end instance
   instance $C$
      body
         hot out $m_3$ to $B$ async ;
         cold in $m_5$ from $B$ async ;
         cold out $m_6$ to env async ;
      end body
   end instance
end lsc
```

Next, we derive the PA_{LSC} term from the textual syntax by using the semantic function for LSCs. Let the chart be denoted by *ch*, then the semantics of *ch* is given by the PA_{LSC} term below.

$$
\begin{array}{l}
S_{LSC}[\![ch]\!] = \\
\quad \lambda_{\emptyset,\emptyset}((hotcond(cond_1) \cdot out(A,B,m_1) \parallel \\
\qquad hotcond(cond_1) \cdot in(A,B,m_1)) \\
\qquad \triangleright \\
\qquad (out(A,B,m_2) \cdot coldcond(cond_2) \cdot outs(A,B,m_4) \\
\qquad \parallel \\
\qquad (in(A,B,m_2) \parallel in(C,A,m_3)) \cdot coldcond(cond_2) \cdot \\
\qquad ins(A,B,m_4) \cdot out(B,C,m_3) \\
\qquad \parallel \\
\qquad out(C,B,m_3) \cdot (\epsilon + in(B,C,m_5) \cdot (\epsilon + out(C,env,m_6))))) \\
\qquad)
\end{array}
$$

Chapter 3

Statecharts

3.1 Introduction

In this section we describe Statecharts [21, 22] which is a graphical notation tailored for specifying the control flow of reactive systems, i.e. event-driven systems which react to internal and external stimuli. Many electronic devices, such as digital clocks, radios, kitchen appliances, smoke alarms, motion sensors, etc. are reactive systems. Computer programs such as word processors and Internet browsers, that require some form of input from the user during execution are other examples of reactive systems.

The opposite to reactive systems are transformational systems that perform some computation and terminate once the result has been evaluated.

On closer examination, a reactive system actually encompasses several transformational systems, since whenever an event triggers a transition, the resulting change of state may be expressed as a function from states to states, i.e. a transformation on the state.

There are several well established methods for specifying transformational systems, for example a direct definition of a function relating input values to output values, or indirectly through post-conditions stating properties of the output values, assuming the inputs satisfy some pre-conditions.

Statecharts extend conventional state machines and state diagrams with hierarchical states and ways of specifying concurrency and communication. The addition of hierarchy is intended to prevent exponential increase in the number of states required to model complex systems.

The history of state machines or automata is almost as long as the history of computer science. The first reference to automata theory is Kleene's paper from 1956 [36]. Automata play an important role in many areas of computer science, notably in string matching and lexical analysis.

A variant of Harel's Statecharts have been included in the UML suite of diagram types [7, 32, 53].

3.2 Syntax of Statecharts

Like state machines and state diagrams, Statecharts are centered around *states* and *transitions*. A state represents a possible configuration of a system. The behaviour of the system in response to internal and external stimuli depends on the state(s) it is currently in, called the active state(s). A transition describes a change of active state. A transition is triggered by an event or action and may set off other actions.

Statecharts are represented graphically as so-called *higraphs* [22], utilizing area inclusion rather than the more conventional tree or graph structure for representing hierarchy. States are represented as rounded rectangles (for simplicity called boxes in the following). A state, s_c that is fully contained within another state, s_p, is called a sub-state of s_p.

States may be decomposed into sub-states using either AND or XOR decomposition. AND decomposition captures the property that when a system is in a given state, it must also be in all sub-states of that state. Conversely, XOR decomposition captures the property that when a system is in a given state, it must be in exactly one of the sub-states of that state. XOR decomposition is represented by having several sub-states. AND decomposition is represented by subdividing the box of the containing state with a dashed line and placing concurrent sub-states on either side of the line.

Transitions are represented as arrows from states to states. An arrow is labelled with an identifier for the event(s) that triggers the transition and, optionally, a condition enclosed in parentheses. In an extension of the original Statecharts, Pnueli and Shalev [47] allowed negative events to trigger transitions. A negative event is interpreted as the absence of the event itself. The unary logical negation operator, $\neg$, is used to negate events. Typically, a transition will be triggered by both positive and negative events, i.e. it will only occur if all the positive events are offered by the environment, while none of the negative events are offered.

When a transition occurs, control is transferred from the origin state to the destination state. If the origin of a transition is a state with sub-states, control is relinquished by all sub-states. If the destination of a transition is a state that is AND decomposed into sub-states, control is assumed by all sub-states. If the destination is XOR decomposed, control is assumed by the default sub-state. Default states are indicated by a small filled circle with an arrow pointing to the box of the default sub-state. A default state functions like an initial state in a state machine.

Example 3.2.1. Figure 3.1 shows a Statechart with four states, A, B, C and D. State A is XOR decomposed into B and C, with B being the default state.

The Statechart responds to three different events, a, b and c. When the system is in state D it may go to state C upon receiving event c, or it may go to state A upon receiving event a. Since A is XOR decomposed, activating state A leads to activating state B as well, since B is the default sub-state for A. If state A is active and event b occurs, the system will transition to state D, regardless of which sub-state of A is active.

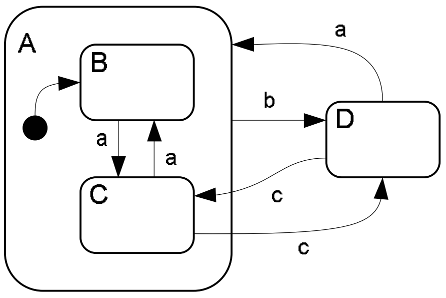


Figure 3.1: Statechart with XOR decomposition.

□

Example 3.2.2. Figure 3.2 shows a Statechart with AND decomposition. The Statechart responds to three events, a, b, and c. When the system is in state G and receives event a, states C and E will be activated concurrently. If either a sub-state of A or a sub-state of B is active, the occurrence of event c will cause G to become the active state. □

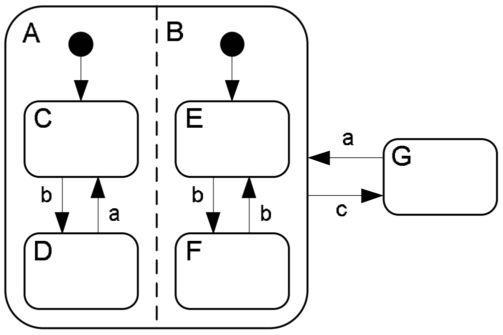


Figure 3.2: Statechart with AND decomposition.

The introduction of the concepts of AND and XOR states are the key to avoiding the exponential blow-up in the number of states as the system being modelled becomes increasingly complex. However, any Statechart including either form of decomposition may be transformed into an equivalent (in a sense to be defined precisely later) Statechart without hierarchical states. The procedure to eliminate an XOR state is to extend every incoming transition to the default sub-state and for every outgoing transition add an outgoing transition with the same event trigger and action and target state to every sub-state. An AND decomposed state may be eliminated by forming new states for every possible combination of concurrent substates.

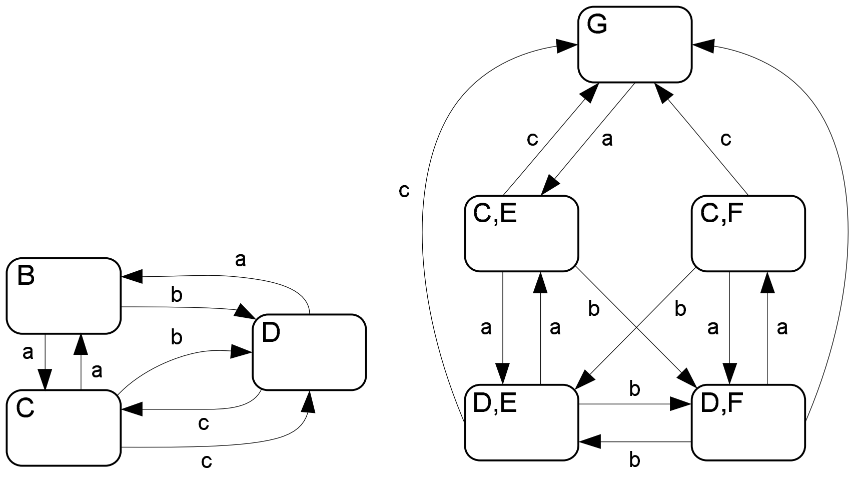
Example 3.2.3. Figure 3.3(a) illustrates the unwinding of the Statechart with AND decomposition in Figure 3.1 into a non-hierarchical Statechart. In this case the unwound Statechart is not more complicated than the original, since it has one less state but one more transition. In general, an unwound AND decomposed Statechart will have at most the same number of states as the original and at least the same number of transitions as the original.

Similarly, Figure 3.3(b) illustrates the unwinding of the Statechart with XOR decomposition in Figure 3.2 into a non-hierarchical Statechart. In this case the resulting Statechart is considerably more complicated that the original. There are only 5 states compared to 6 in the original, but there are 13 transitions compared to 6 in the original.

□

Statecharts support the modelling concepts of abstraction and refinement. Abstraction is the process of extracting common properties from a model. Refinement is the process of adding additional details to a model. In the setting of Statecharts both concepts rely on hierarchical decomposition. Abstraction is supported by moving the common properties (i.e. transitions with the same event trigger and same destination state) of a set of states to a new state that has the original states as substates. Refinement is supported by adding new substates and internal transitions to an existing state.

Example 3.2.4. (Example taken from [21]). Figure 3.4 illustrates the process of abstraction for Statecharts. In the Statechart on the left, the transition on event b from



(a) (b)

Figure 3.3: Unwinding of AND (a) and XOR (b) decomposition into a simple Statechart.

states B and C is a common property of these two state. By introducing a new super-state, these two transitions can be replaced by one common transition, as shown in the Statechart on the right.

Figure 3.5 shows the process of refinement. In the intermediate step in the middle, the state D is refined to show additional details of its internal structure. However, now the two transitions from A to D become under-specified, since it is not clear which of B and C should become active after one of the transitions have occurred. In the Statechart on the right, the transitions have been extended to remove the under-specification. Alternatively, either B or C could have been defined as the default sub-state of D.

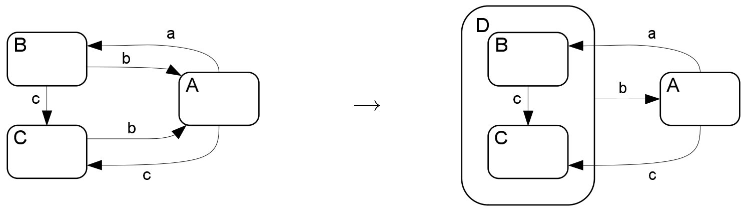


Figure 3.4: Abstraction.

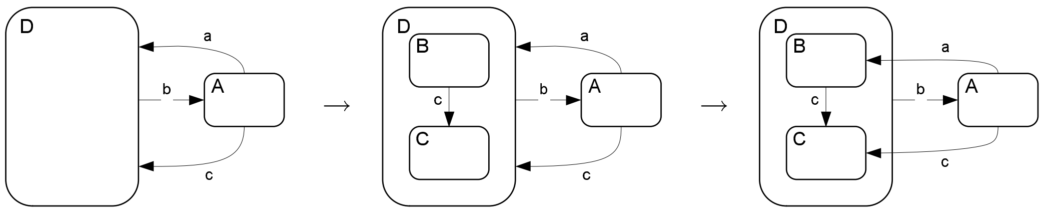


Figure 3.5: Refinement.

□

A special kind of transition causes control to be transferred to the sub-state(s) that most recently had control instead of the default sub-state. This history-dependent type of transition may be related only to the immediate substates, or recursively to substates, substates of substates, etc. In cases where no history is available, i.e. the first time control is transferred to a state, the default substates are used. History-dependent transitions are represented by letting the arrow of the transition point to a symbol composed of an ‘H’ inside a circle. If the transition depends on the recursive history, the H-symbol is decorated with an asterisk.

In some situations, it is convenient to be able to forget the past history. For this purpose, a distinguished action, $clh(S)$, that resets the history of a state S and all substates of S is introduced. Once the history has been reset, the next time a history dependent transition occurs, the default state and not the most recently visited state will become the active state.

Example 3.2.5. Figure 3.6 illustrates a Statechart with history dependent transitions. The first time a state is activated by a history transition, there is no history, so the default sub-state becomes the active state. Thus, in this case, the first time B is activated, F becomes active. Suppose now that E is the active state and that event a occurs, so A becomes the new active state. If event a occurs now, D becomes the active state, since only the first level of the activation history is used. C was the most recently activated sub-state of B, so the default sub-state of C, namely D, is activated. If, on the other hand, event b occurs, E becomes the active state, since in this case the entire history is used. Finally, if B is active and event d occurs, all history is cleared for B, so the next history transition will cause the default sub-state to be activated.

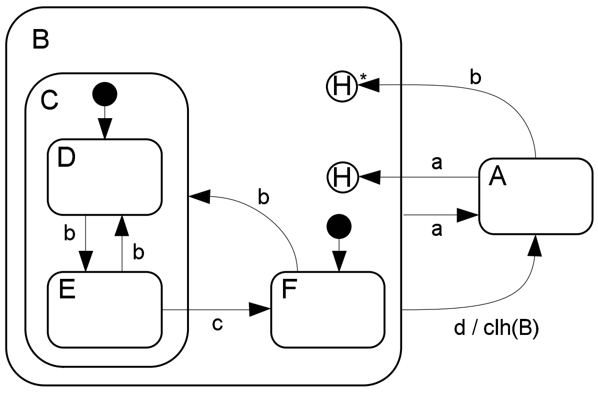


Figure 3.6: Statechart with history and recursive history.

□

The events that trigger transitions are typically the result of external stimuli, but may also be generated by timeouts when control has been in a given state for a predetermined period of time. We do not consider such timeouts here.

Example 3.2.6. Figure 3.7 shows an example Statechart modelling a reactive system that receives four kinds of stimuli from the environment (essentially like four buttons). Each kind of stimuli generates a unique event, called a, b, c and d.

The system is represented by state A. Initially, the system is in state B. When an event d occurs, control is transferred to state F. A b event will now transfer control to state E. An additional b event will transfer control to state G and H, which is the default substate of G, while a c event will transfer control back to state F. When the system is in any of the substates of G, a b event will cause control to be transferred to F.

When the system is in any substate of D an a event will transfer control to both J and K. Similarly, when another b event occurs, control is relinquished by both J and K and all their substates. Control is then transferred to the most recently visited states in D down to the lowest level, i.e. the states from which control was relinquished when the last a event occurred.

The label "/clh(C)" on the transition from B to C is an action that indicates that when this transition occurs, the history of C and its substates is deleted all the way down to the lowest level. Thus, the next time the transition from J/K to D occurs, the default state will be entered.

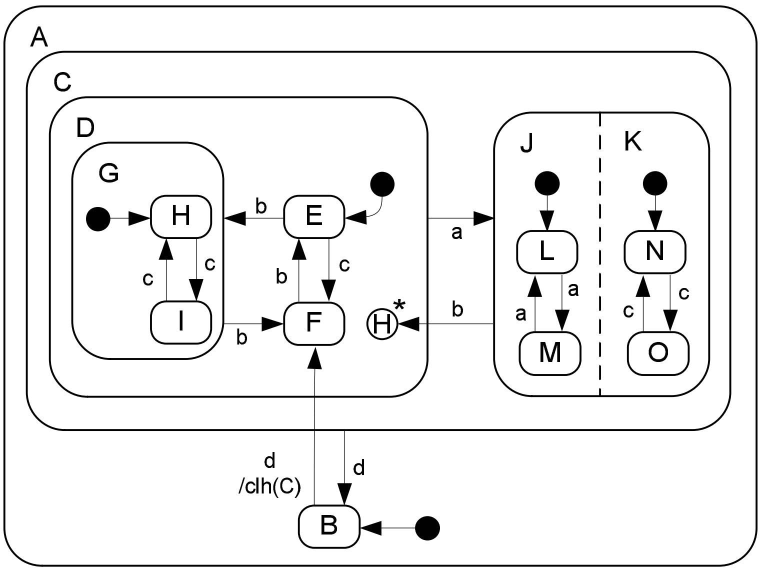


Figure 3.7: Example Statechart.

□

3.3 Process Algebra

Statecharts have proven difficult to give a process algebraic semantics. This difficulty arises partly because of the property that an external event may trigger a transition that produces an event that in turn triggers a transition, etc. Thus one event may start a chain reaction of internal events. Furthermore, if a Statechart is in a given state, it is also in all states enclosing the first state. Therefore, the global state or configuration of a Statechart consists of a variable number of states.

An internal transition is called a micro-step, while the whole chain reaction caused by an external event is called a macro-step.

There are three desirable properties for a semantics for Statecharts: the synchrony hypothesis, compositionality, and causality.

The synchrony hypothesis states that for any set of input events, the reaction of a Statechart must be maximal in the sense that the chain reaction of micro-steps should continue until no further micro-step is possible. This is sometimes referred to as the *maximal progress* assumption. Also, the chain reaction must terminate before the next external event enters the system.

The compositionality property ensures that the behaviour of a system composed from sub-systems is defined in terms of the observable behaviour of the sub-systems. Thus the internal details of the sub-systems need not be known.

The causality property ensures that for every event, there is a chain of events that lead to that event. Thus, no event can occur spontaneously. This property only applies to internal events, since external events – when viewed from the Statechart – will occur spontaneously.

3.3.1 SPL

The process algebraic semantics for Statecharts is presented by Lüttgen, van der Beeck and Cleaveland [42]. It is defined in using the process algebra named Statecharts Process Language (SPL), which is inspired by the Timed Process Language of Hennesy and Regan [28].

The SPL process algebra is defined as a labelled transition system with two types of transitions: action transitions and clock transitions. Action transitions will correspond to events in Statecharts. Clock transitions represent progression of time. The previously discussed micro-steps of a Statechart correspond to action transitions, while clock transitions signal the beginning and end of a macro-step composed of a sequence of micro-steps.

Let Λ be a countable set of events and let $\sigma \notin \Lambda$ be a distinguished event called a *clock event*. *Input actions* are defined as $\langle E, N\rangle$, where $E, N \subseteq \Lambda$. The special case of $\langle\emptyset,\emptyset\rangle$ is called an *unobservable* or *internal* event, also denoted $\bullet$. *Output actions* E are defined as subsets of Λ.

The syntax of SPL is given by the BNF grammar:

$$P ::= \mathbf{0} \mid X \mid \langle E, N\rangle.P \mid [E]\sigma(P) \mid P + P \mid P \rhd P \mid P \rhd_\sigma P \mid P \parallel P \mid P \setminus L$$

where X is a *process variable* that stands for a process term, and L is a *restriction set*, i.e. a set of action identifiers that are hidden from the environment of $X \setminus L$. $\mathbf{0}$ is the empty process, i.e. the process which does not perform any actions. $\langle E, N\rangle.P$ is the *prefix operator* applied to the process P. It represents the instantaneous input of the input action $\langle E, N\rangle$, which can only occur if all the events in E are offered by the environment, and none of the events in N are offered by the environment. The *signal operator* $[E]\sigma(P)$ signals the output of output action E to the environment of process P. The output action is cleared by the next clock transition. The *disabling operator* applied to processes P and Q, written $P \rhd Q$, is the process that either behaves as Q permanently disabling P, or behaves as $P \rhd_\sigma Q$. The *enabling operator* applied to processes P and Q, written $P \rhd_\sigma Q$ behaves as P, disabling Q until the next clock transition. In combination, the disabling and enabling operators serve to define the behaviour when there are enabled transitions on several layers of a hierarchical state.

3.3.2 Semantics of SPL

The semantics of SPL is a Plotkin style operational semantics in the form of a labelled transition system. The labelled transition system is defined as $\langle \mathcal{S}, E, \rightarrow, S\rangle$, where $\mathcal{S}$ is the set of states, $E = A \cup \{\sigma\}$ is the set of actions, including the special clock action, σ, $\rightarrow \in \mathcal{S} \times E \times \mathcal{S}$ is the transition relation, and S is the start state. Following tradition, $P \underset{N}{\overset{E}{\rightarrow}} P'$ will be used as an abbreviation for $(P, \langle E, N\rangle, P') \in\rightarrow$ and $P \overset{\sigma}{\rightarrow} P'$ as an abbreviation for $(P, \sigma, P') \in\rightarrow$. The meaning of $P \underset{N}{\overset{E}{\rightarrow}} P'$ is that process P can evolve to a process P' whenever the environment of P outputs all the actions in E and none of the actions in N.

Before the transition deduction system can be defined, the *initial output action set* must be defined. The initial output action set, notation $\mathbb{I}(P)$ for $P \in \mathcal{S}$, is defined by the equations in Table 3.1. Intuitively, $\bar{\mathbb{I}}(P)$ is the set of actions that process P is immediately ready to output.

$$\begin{array}{ll}
\bar{\mathbb{I}}([E]\sigma(P)) = E & \bar{\mathbb{I}}(X) = \bar{\mathbb{I}}(P),\ \text{if } X \overset{def}{=} P \\
\bar{\mathbb{I}}(P + Q) = \bar{\mathbb{I}}(P) \cup \bar{\mathbb{I}}(Q) & \bar{\mathbb{I}}(P \setminus L) = \bar{\mathbb{I}}(P) \setminus L \\
\bar{\mathbb{I}}(P \parallel Q) = \bar{\mathbb{I}}(P) \cup \bar{\mathbb{I}}(Q) & \bar{\mathbb{I}}(P \rhd_\sigma Q) = \bar{\mathbb{I}}(P) \\
\bar{\mathbb{I}}(P \rhd Q) = \bar{\mathbb{I}}(P) \cup \bar{\mathbb{I}}(Q) &
\end{array}$$

Table 3.1: Initial output action sets.

The term deduction system for action transitions is presented in Table 3.2

$$\begin{array}{llll}
\dfrac{\Box}{\langle E, N\rangle.P \xrightarrow[N]{E} P} & \text{Act} & \dfrac{P \xrightarrow[N]{E} P'}{P + Q \xrightarrow[N]{E} P'} & \text{Sum1} \\
\dfrac{P \xrightarrow[N]{E} P'}{X \xrightarrow[N]{E} P'} \text{ if } X \overset{def}{=} P & \text{Rec} & \dfrac{Q \xrightarrow[N]{E} Q'}{P + Q \xrightarrow[N]{E} Q'} & \text{Sum2} \\
\dfrac{P \xrightarrow[N]{E} P'}{P \rhd_\sigma Q \xrightarrow[N]{E} P' \rhd_\sigma Q} & \text{En} & \dfrac{Q \xrightarrow[N]{E} Q'}{P \rhd Q \xrightarrow[N]{E} Q'} & \text{Dis2} \\
\dfrac{P \xrightarrow[N]{E} P'}{P \rhd Q \xrightarrow[N]{E} P' \rhd_\sigma Q} & \text{Dis1} & &
\end{array}$$

$$\begin{array}{lll}
\dfrac{P \xrightarrow[N]{E} P'}{P \parallel Q \xrightarrow[N]{E \setminus \bar{\mathbb{I}}(Q)} P' \parallel Q} & \text{if } N \cap \bar{\mathbb{I}}(Q) = \emptyset & \text{Par1} \\
\dfrac{Q \xrightarrow[N]{E} Q'}{P \parallel Q \xrightarrow[N]{E \setminus \bar{\mathbb{I}}(P)} P \parallel Q'} & \text{if } N \cap \bar{\mathbb{I}}(P) = \emptyset & \text{Par2} \\
\dfrac{P \xrightarrow[N]{E} P'}{P \setminus L \xrightarrow[N \setminus L]{E} P' \setminus L} & \text{if } E \cap L = \emptyset & \text{Par1}
\end{array}$$

Table 3.2: Action transitions.

The term deduction system for clock transitions is presented in Table 3.3.

3.3.3 Equivalence for SPL terms

We can now define a behavioural equivalence on SPL terms. As we did previously for PA_ϵ, we choose the strong bisimulation equivalence.

Definition 3.3.1. Bisimulation equivalence, $\sim \subseteq \mathcal{S} \times \mathcal{S}$, is the largest symmetric relation

$\dfrac{\Box}{\mathbf{0} \xrightarrow{\sigma} \mathbf{0}}$	tNil	$\dfrac{\Box}{[E]\sigma(P) \xrightarrow{\sigma} P}$	tOut
$\dfrac{\Box}{\langle E,N\rangle.P \xrightarrow{\sigma} \langle E,N\rangle.P}$ if $\langle E,N\rangle \neq \bullet$	tAct	$\dfrac{P \xrightarrow{\sigma} P', Q \xrightarrow{\sigma} Q'}{P + Q \xrightarrow{\sigma} P' + Q'}$	tSum
$\dfrac{P \xrightarrow{\sigma} P', Q \xrightarrow{\sigma} Q'}{P \parallel Q \xrightarrow{\sigma} P' \parallel Q'}$ if $\bullet \notin \mathrm{I}(P \parallel Q)$	tPar	$\dfrac{P \xrightarrow{\sigma} P'}{P \rhd_\sigma Q \xrightarrow{\sigma} P' \rhd Q}$	tEn
$\dfrac{P \xrightarrow{\sigma} P', Q \xrightarrow{\sigma} Q'}{P \rhd Q \xrightarrow{\sigma} P' \rhd Q'}$	tDis	$\dfrac{P \xrightarrow{\sigma} P'}{X \xrightarrow{\sigma} P'}$ if $X \overset{def}{=} P$	tRec
$\dfrac{P \xrightarrow{\sigma} P'}{P \setminus L \xrightarrow{\sigma} P' \setminus L}$ if $\bullet \notin \mathrm{I}(P \setminus L)$	tRes		

Table 3.3: Clock transitions.

such that whenever $P \sim Q$, then the following conditions hold

1. $\bar{\mathbb{I}}(P) \subseteq \bar{\mathbb{I}}(Q)$
2. If $P \xrightarrow[N]{E} P'$, then $\exists Q' \in \mathcal{S} : Q \xrightarrow[N]{E} Q' \wedge Q \sim Q'$.

$\Box$

Note that compared to the bisimulation relations defined for the process algebras PA_ϵ and PAc_ϵ in Section 2.3.2 we have the extra requirement the bisimilar processes have the same initial output sets. This requirement ensures that bisimilar processes have the same observable behaviour in terms of both input and output actions.

3.4 Semantics of Statecharts

We have now presented the tools for expressing the semantics of Statecharts. The next step is to define the correspondence between a Statechart and an SPL term. First, we place some restrictions on the composition of Statecharts by defining a textual syntax, in the form of Statechart terms. Then we define a semantic function that maps Statechart terms to SPL terms.

We need some additional notation. Let $\mathcal{N}$ be a countable set of names for Statechart states, $\mathcal{T}$ be a countable set of names for Statechart transitions and Π a countable set of Statechart events. Every event $e \in \Pi$ has a negated event $\neg e$. By definition $\neg\neg e = e$. If $E \subseteq \Pi \cup \{\neg e \mid e \in \Pi\}$ then $\neg E$ is an abbreviation for $\{\neg e \mid e \in E\}$.

Now, Statechart terms are introduced. In order for a Statechart to be expressible as a Statechart term, it must have exactly one top-level state and it must have no history or inter-level transitions, i.e. transitions that cross the boundary of its containing state. History transitions are disallowed because they make the semantics much more complicated. Inter-level transitions are disallowed because they preclude compositionality in both the syntax and semantics. Note, however, that a Statechart with inter-level transitions can always be transformed into an equivalent Statechart without inter-level transitions.

1. Basic state: If $n \in \mathcal{N}$, then $s = [n]$ is a Statechart term.
2. XOR-state: If $n \in \mathcal{N}$, $s_1, \ldots, s_k$ are Statechart terms for $k > 0$, $T \subseteq \mathcal{T} \times \{1, \ldots, k\} \times \mathcal{P}(\Pi \cup \neg\Pi) \times \mathcal{P}(\Pi) \times \{1, \ldots, k\}$, and $1 \leq l \leq k$, then $s = [n : (s_1, \ldots, s_k), l, T]$ is a Statechart term. Here, $s_1, \ldots, s_k$ are the sub-states of s, l is the index of the

currently active state and T is the set of transitions between the sub-states of s. The default state is defined to be s_1. A transition $\langle t, n_1, E, A, n_2\rangle$ with name t links state s_{n_1} to state s_{n_2}, is triggered by the events in E and produces the actions in A.

3. AND-state: If $n \in \mathcal{N}$, and $s_1, \ldots, s_k$ are Statechart terms for $k > 0$, then $s = [n : (s_1, \ldots, s_k)]$ is a Statechart term.

A Statechart term is considered well-formed, if:

(a) the set of names for states is disjoint from the set of names for transitions, i.e. $\mathcal{N} \cap \mathcal{T} = \emptyset$;
(b) no transition produces an event that contradicts its trigger, i.e. for every transition $\langle t, n_1, E, A, n_2\rangle$, $E \cap \neg A = \emptyset$;
(c) no transition produces an event that is in its trigger, i.e. for every transition $\langle t, n_1, E, A, n_2\rangle$, $E \cap A = \emptyset$.

The set of well-formed Statechart terms is denoted SC.

The function *root* yields the name of the state it is applied to. The function *out* yields the name of the destination state of the transition it is applied to.

Now, the embedding is defined. We give the definition first and then explain it below. The process algebra SPL is instantiated with the set of events $\Lambda = \Pi \cup \neg\Pi$ and the set of process variables $\mathcal{V} = \{\hat{n} \mid n \in \mathcal{N}\}$. Let $\Sigma\ Q$ be the distributed non-deterministic choice between the elements of the set Q, with $\Sigma\{\ \} = \mathbf{0}$, then the embedding function $S_{StC}[\![\,\cdot\,]\!] : SC \to T_{\Sigma_{SPL}}$ is defined as

1. If $s = [n]$, then $S_{StC}[\![s]\!] = \mathbf{0}$.
2. If $s = [n : (s_1, \ldots, s_n), l, T]$, then if $n_l = root(s_l)$, $S_{StC}[\![s]\!] = \hat{n}_l$, where for $1 \leq i \leq n$, $\hat{n}_i = S_{StC}[\![s_i]\!] \rhd \Sigma\{\{\![t]\!\} \mid t \in T \wedge root(out(t)) = n_i\}$ along with the equations produced by $S_{StC}[\![s_1]\!], \ldots, S_{StC}[\![s_n]\!]$. The translation $\{\![t]\!\}$ of a transition t is defined below.
3. If $s = [n : (s_1, \ldots, s_n)]$, then $S_{StC}[\![s]\!] = S_{StC}[\![s_1]\!] \parallel \cdots \parallel S_{StC}[\![s_n]\!]$, along with the equations produced by $S_{StC}[\![s_1]\!], \ldots, S_{StC}[\![s_n]\!]$.

The translation of a transition $t = \langle t, i, E, A, j\rangle$ is defined as $\{\![t]\!\} = \langle E', N'\rangle.[A \cup (E \cap \neg\Pi)]\sigma(\hat{n}_j)$, where $E' = E \cap \Pi$ is the set of positive events in E and $N' = \neg(E \cap \neg\Pi) \cup \neg A$ is the set of negated negative events in E combined with the negated events in A.

The definition of the embedding requires an explanation. First of all, the semantics of a Statechart is expressed as a set of equations rather than a single process term. This allows for recursion. The semantics of a basic state is the inactive process $\mathbf{0}$, since a basic state will not take part in any transitions. The semantics of an AND-state is just the parallel composition of the semantics' of its substates. The semantics of an XOR-state is more involved. First observe that an XOR-state may either stay in the currently active substate, or a transition t may occur, making $out(t)$ the new active substate. This behaviour is modelled by the disabling operator. In the former case the XOR-state behaves like the currently active substate, disabling all transitions until the next clock event. In the latter case, the transition becomes an input prefix handling the triggering events in E, and an output signalling handling the actions in A. For the transition to occur, all the positive events in E and none of the negative events in E must be offered by the environment. This explains E' and partly N'. The reason why $\neg A$ is included in N' is that we must ensure *global consistency*, meaning that no subsequent transition which requires the absence of the events in A fires in the same macro-step. The global consistency requirement also explains why the output includes the negative events in E, since the process is not allowed to produce an event which contradicts its trigger.

3.5 Statecharts Example

In this section the process of deriving an SPL expression from a Statechart is illustrated. The example Statechart is shown in Figure 3.8. In this case, the Statechart is already in a form suitable for conversion to a Statechart term. If this were not the case, the Statechart would first have to be modified to remove inter-level transitions and to have exactly one top-level state.

The corresponding Statechart term s_1 is listed below along with terms for each of the substates of state n_1.

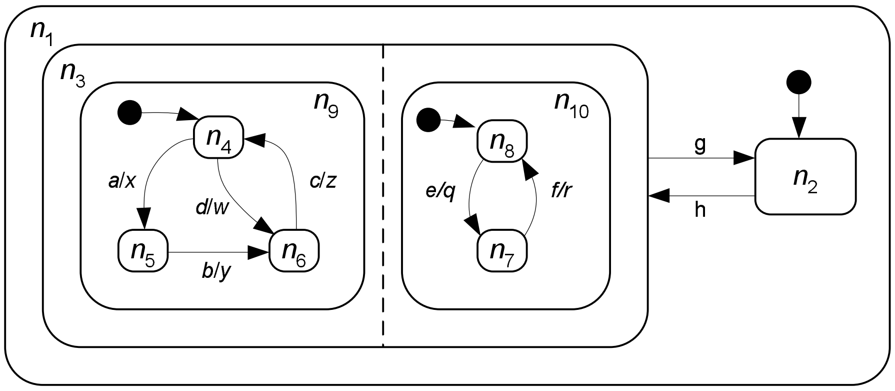


Figure 3.8: Example Statechart.

$$
\begin{aligned}
s_1 &= [n_1 : (s_2, s_3); 1; \{\langle t_1, 3, \{g\}, \emptyset, 2\rangle, \langle t_2, 2, \{h\}, \emptyset, 3\rangle\}] \\
s_2 &= [n_2] \\
s_3 &= [n_3 : (s_9, s_10)] \\
s_4 &= [n_4] \\
s_5 &= [n_5] \\
s_6 &= [n_6] \\
s_7 &= [n_7] \\
s_8 &= [n_8] \\
s_9 &= [n_9 : (s_4, s_5, s_6); 4; \{\langle t_3, 4, \{a\}, \{x\}, 5\rangle, \langle t_4, 6, \{b\}, \{y\}, 5\rangle, \langle 5, s_4, \{c\}, \{z\}, 6\rangle, \\
&\qquad \langle t_6, 4, \{d\}, \{w\}, 6\rangle\}] \\
s_{10} &= [n_{10} : (s_7, s_8); s_7; \{\langle t_6, 8, \{e\}, \{q\}, 7\rangle, \langle t_7, 7, \{f\}, \{r\}, 8\rangle\}]
\end{aligned}
$$

The translation of the Statechart terms into SPL is straightforward. The result is listed below.

$$
\begin{aligned}
&[\![s_1]\!] = \hat{n}_2\\
&\qquad \hat{n}_2 = [\![s_2]\!] \rhd \langle\{h\}, \emptyset\rangle.[\emptyset]\sigma(\hat{n}_3)\\
&\qquad \hat{n}_3 = [\![s_3]\!] \rhd \langle\{g\}, \emptyset\rangle.[\emptyset]\sigma(\hat{n}_2)\\
&[\![s_2]\!] = \mathbf{0}\\
&[\![s_3]\!] = [\![s_9]\!] \parallel [\![s_1 0]\!]\\
&[\![s_4]\!] = \mathbf{0}\\
&[\![s_5]\!] = \mathbf{0}\\
&[\![s_6]\!] = \mathbf{0}\\
&[\![s_7]\!] = \mathbf{0}\\
&[\![s_8]\!] = \mathbf{0}\\
&[\![s_9]\!] = \hat{n}_4\\
&\qquad \hat{n}_4 = [\![s_4]\!] \rhd \langle\{a\}, \{\neg x\}\rangle.[\{x\}]\sigma(\hat{n}_5) + \langle\{d\}, \{\neg w\}\rangle.[\{w\}]\sigma(\hat{n}_6)\\
&\qquad \hat{n}_5 = [\![s_5]\!] \rhd \langle\{b\}, \{\neg y\}\rangle.[\{y\}]\sigma(\hat{n}_6)\\
&\qquad \hat{n}_6 = [\![s_6]\!] \rhd \langle\{c\}, \{\neg z\}\rangle.[\{z\}]\sigma(\hat{n}_7)\\
&[\![s_{10}]\!] = \hat{n}_8\\
&\qquad \hat{n}_8 = [\![s_8]\!] \rhd \langle\{e\}, \{\neg q\}\rangle.[\{q\}]\sigma(\hat{n}_7)\\
&\qquad \hat{n}_7 = [\![s_7]\!] \rhd \langle\{f\}, \{\neg r\}\rangle.[\{r\}]\sigma(\hat{n}_8)
\end{aligned}
$$

Chapter 4

Relating Diagrams to RSL

4.1 Introduction

In this chapter we briefly review a number of ways of integrating different specification notations. We then define a subset of RSL and give an operational semantics, based on the semantics for Timed RSL as defined by George and Xia [16]. We extend the semantic rules with behaviour annotations capturing the communication behaviour of the RSL expression. Utilizing these behaviours, we define three satisfaction relations: one relating a universal LSC to an RSL specification, one relating an existential LSC to an RSL specification and one relating a Statechart to an RSL specification.

4.2 Types of Integration

Haxthausen [26] identifies three approaches to integrating different specification techniques:

- the *unifying, wide-spectrum* approach,
- the *family* approach, and
- the *linking* approach.

The wide-spectrum approach provides a complete semantical integration of the techniques. This was the approach adopted in the development of RSL. The advantage of this approach is that the same language is used throughout the development process. The disadvantage is that this approach results in a complicated semantics.

The idea in the family approach is to define a reasonably expressive base language and then integrating other techniques by defining extension languages. The semantics of the extension languages are required to be consistent with the semantics of the base language. This approach is used in the CoFI project, for which the base language is called Casl. The advantage of the family approach is that the semantics is “only as complicated as it needs to be”, in the sense that for a particular project, one uses the smallest language in the family that has the required facilities.

In the previous two approaches a new semantics that subsumes the semantics’ of the individual techniques is developed. In contrast, in the linking approach, the individual semantics’ are preserved, and the integration instead takes the form of a formal relation between the individual semantics’. This approach is particularly suited for specification techniques that are fundamentally different.

There is also a fourth approach to integration, namely what we call the combination approach. In this approach one notation is embedded in the other to extend its expressiveness. An example is Coloured Petri Nets, which are the result of the combination of classical Petri Nets with an ML-like language [39, 33] used for inscriptions on arcs and type definitions. Other examples are the combinations of Statecharts with Casl and Statecharts with Z mentioned in the introduction.

We believe that of the four approaches described, the linking approach is most suited for our purpose. By using this approach we do not have to "massage" the familiar semantics' of the individual techniques into a new framework. Additionally, all the tools (proof system, syntax checkers, code generators) developed for RSL are immediately available in the integrated method.

What we present in the rest of this chapter is therefore how to link Live Sequence Charts and Statecharts with RSL.

4.3 RSL Subset

4.3.1 Syntax

The subset of RSL defined below is almost the same as the subset defined by George and Xia [16] for Timed RSL. We omit the **wait** construct and use the standard input and output operators from RSL rather than the corresponding operators in Timed RSL. Also, we exclude the special notation for recursive functions. For use in establishing the relation to Live Sequence Charts and Statecharts, we annotate the input and output operators with a message identifier. Similarly, the parallel and interlocking operators are annotated with two process identifiers.

We assume familiarity with RSL and therefore skip an informal description of the operators and constructs of the RSL subset.

The syntactic categories are

- Expressions denoted by E,
- Variables denoted by x,
- Identifiers denoted by id,
- Channels denoted by c,
- Reals denoted by r,
- Types denoted by τ,
- Value definitions denoted by V,
- Message identifiers denoted by *msgid*,
- Process identifiers denoted by n.

The grammar of the subset of RSL is given below.

$\mathrm{V} ::= \mathrm{id} : \tau \mid \mathrm{id} : \tau, \mathrm{V}$

$\mathrm{E} ::= () \mid \textbf{true} \mid \textbf{false} \mid \mathrm{r} \mid \mathrm{id} \mid \mathrm{x} \mid \textbf{skip} \mid \textbf{stop} \mid \textbf{chaos}$
$\quad\mid\quad \mathrm{x} := \mathrm{E} \mid \textbf{if}\ \mathrm{E}\ \textbf{then}\ \mathrm{E}\ \textbf{else}\ \mathrm{E} \mid \textbf{let}\ \mathrm{id} = \mathrm{E}\ \textbf{in}\ \mathrm{E} \mid \mathrm{c}?_{msgid} \mid \mathrm{c}!_{msgid}\mathrm{E}$
$\quad\mid\quad \mathrm{E} \sqcap \mathrm{E} \mid \mathrm{E} \;[\!]\; \mathrm{E} \mid \mathrm{E}\ {}_{n}\|_{n}\ \mathrm{E} \mid \mathrm{E}\ {}_{n}\#_{n}\ \mathrm{E} \mid \mathrm{E} \ ;\ \mathrm{E}$
$\quad\mid\quad \lambda\ \mathrm{id} : \tau \bullet \mathrm{E} \mid \mathrm{E}\ \mathrm{E}$

When in the following we refer to an RSL expression, we mean an expression within the subset of RSL defined here.

4.3.2 Operational Semantics with Communication Behaviour

Before presenting the rules of the operational semantics a number of definitions are needed.

A store s is a finite map from variables (x) to values (v): $s = [x \mapsto v, \ldots]$.

An environment ρ is a finite map from identifiers (id) to values (v) : $\rho = [id \mapsto v, \ldots\ ]$.

A closure is a pair consisting of a lambda expression (λ id : τ • E) and an environment (ρ): $[\![\lambda \text{ id} : \tau \bullet \text{E}, \rho\ ]\!]$.

Compared to George and Xia [16], we modify the notion of a configuration to a triple $< E, s, n >$ where E is an expression, s is a store and n is a process identifier. Moreover, we augment configurations of the form $\alpha\ op\ s\ op\ \beta$ for $op = \|$, $\nparallel$ to include three process identifiers, i.e. $\alpha\ op\ (s, n, n_1, n_2)\ op\ \beta$, where n_1 is the identifier of the process represented by the configuration α, while n_2 is the identifier of the process represented by β.

Inspired by Haxthausen and Xia [27], the rules of the operational semantics are extended to include communication behaviour in the form of a PA_{LSC} term. The transition relation has the form

$$\rho\ \vdash\ \alpha_{\textbf{with}\ \phi}\ \xrightarrow{e}\ \alpha'_{\textbf{with}\ \phi'}$$

where ρ is the environment, α and α' are configurations, ϕ and ϕ' are behaviours and e is an event. The intuition is that the configuration α with the behaviour ϕ can evolve to the configuration α' with behaviour ϕ' by performing the event e.

There are two types of events, silent events and communication events. The silent event, ϵ, denotes an internal change that is not externally visible. Communication events are either input events of the form $c?_{msgid}$ or output events of the form $c!_{msgid}E$. The symbol $\diamond$ is used to denote any event, i.e. a situation where the transition is the same for a silent event and for a communication event.

The only operational rules that change the communication behaviour are the rules for input, output, communication across a parallel or interlocking combinator and merging of two parallel processes. In all other rules, the communication behaviour is preserved.

The process identifier, n, stored in a configuration is used to name processes in PA_{LSC} events. This information is needed to identify the sender and recipient in message input and message output events in the behaviours.

The rules for the parallel and interlocking combinators apply the function *merge* that merges the stores on either side of a parallel composition. It is defined in RSL notation by

$$\text{merge(s, s}', \text{s}'') = \text{s}' \dagger [\text{x} \mapsto \text{s}''(\text{x}) \mid \text{x} \in \textbf{dom}(\text{s}'') \cap \textbf{dom}(\text{s}) \bullet \text{s(x)} \neq \text{s}''(\text{x})\ ]$$

In the rules below we use a notation of the form

$$\frac{C}{\rho\ \vdash\ C_2}$$
$$\qquad\quad C_3$$

as a shorthand for the two rules

$$\frac{C}{\rho\ \vdash\ C_2}$$

and

$$\frac{C}{\rho\ \vdash\ C_3}$$

Also, for rules without premises, i.e. axioms, we write the symbol $\square$ above the line.

Basic Expressions

$$\frac{\square}{\rho\ \vdash\ <\mathbf{skip}, s, n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ <(), s, n>_{\mathbf{with}\ \phi}}$$

$$\frac{\square}{\rho\ \vdash\ <\mathbf{chaos}, s, n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ <\mathbf{chaos}, s, n>_{\mathbf{with}\ \phi}}$$

Configuration Fork

$$\frac{\square}{\rho\ \vdash\ <E_1\ op\ E_2, s, n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ <E_1, s, n>_{\mathbf{with}\ \phi}\ \ op\ \ <E_2, s, n>_{\mathbf{with}\ \phi}}$$

where $op \in \{\sqcap, [\!]\}$

Look up

$$\frac{\square}{\rho \dagger [id \mapsto v]\ \vdash\ <id, s, n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ <v, s, n>_{\mathbf{with}\ \phi}}$$

$$\frac{\square}{\rho\ \vdash\ <id, s \dagger [id \mapsto v], n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ <v, s \dagger [id \mapsto v], n>_{\mathbf{with}\ \phi}}$$

Sequencing

$$\frac{\square}{\rho\ \vdash\ <E_1; E_2, s, n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ (<E_1, s, n>; E_2)_{\mathbf{with}\ \phi}}$$

$$\frac{\rho\ \vdash\ \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond}\ \alpha'_{\mathbf{with}\ \phi'}}{\rho\ \vdash\ (\alpha; E)_{\mathbf{with}\ \phi} \xrightarrow{\diamond}\ (\alpha'; E)_{\mathbf{with}\ \phi'}}$$

$$\frac{\square}{\rho\ \vdash\ (<v, s, n>; E)_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ <E, s, n>_{\mathbf{with}\ \phi}}$$

Assignment

$$\frac{\square}{\rho\ \vdash\ <x := E, s, n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ (x := <E, s, n>)_{\mathbf{with}\ \phi}}$$

$$\frac{\rho\ \vdash\ \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond}\ \alpha'_{\mathbf{with}\ \phi'}}{\rho\ \vdash\ (x := \alpha)_{\mathbf{with}\ \phi} \xrightarrow{\diamond}\ (x := \alpha')_{\mathbf{with}\ \phi'}}$$

$$\frac{\square}{\rho\ \vdash\ <v, s, n>_{\mathbf{with}\ \phi} \xrightarrow{\epsilon}\ <(), s \dagger [x \mapsto v], n>_{\mathbf{with}\ \phi}}$$

Input

$$\frac{\square}{\rho\ \vdash\ <c?_{msgid}, s, n>_{\mathbf{with}\ \phi} \xrightarrow{c?_{msgid}v}\ <v, s, n>_{\mathbf{with}\ \phi\ \cdot\ ins(env,n,msgid)}}$$

Output

$$\frac{\Box}{\rho\ \vdash\ < c!_{msgid}E, s, n >_{\textbf{with}\ \phi} \xrightarrow{\epsilon} (c!_{msgid} < E, s, n >)_{\textbf{with}\ \phi}}$$

$$\frac{\rho\ \vdash\ \alpha_{\textbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with}\ \phi'}}{\rho\ \vdash\ (c!_{msgid}\alpha)_{\textbf{with}\ \phi} \xrightarrow{\diamond} (c!_{msgid}\alpha')_{\textbf{with}\ \phi'}}$$

$$\frac{\Box}{\rho\ \vdash\ (c!_{msgid} < v, s, n >)_{\textbf{with}\ \phi} \xrightarrow{c!_{msgid}v} < (), s, n >_{\textbf{with}\ \phi\ \cdot\ outs(n,env,msgid)}}$$

Internal choice

$$\frac{\Box}{\begin{array}{l}\rho\ \vdash\ (\alpha \sqcap \beta)_{\textbf{with}\ \phi} \xrightarrow{\epsilon} \alpha_{\textbf{with}\ \phi} \\ \qquad\qquad\qquad\quad \xrightarrow{\epsilon} \beta_{\textbf{with}\ \phi}\end{array}}$$

External choice

$$\frac{\rho\ \vdash\ \alpha_{\textbf{with}\ \phi} \xrightarrow{a} \alpha'_{\textbf{with}\ \phi'}}{\begin{array}{l}\rho\ \vdash\ \alpha_{\textbf{with}\ \phi} \, [\!] \, \beta_{\textbf{with}\ \varphi} \xrightarrow{a} \alpha'_{\textbf{with}\ \phi'} \\ \qquad \beta_{\textbf{with}\ \varphi} \, [\!] \, \alpha_{\textbf{with}\ \phi} \xrightarrow{a} \alpha'_{\textbf{with}\ \phi'}\end{array}}$$

$$\frac{\rho\ \vdash\ \alpha_{\textbf{with}\ \phi} \xrightarrow{\epsilon} \alpha'_{\textbf{with}\ \phi'}}{\begin{array}{l}\rho\ \vdash\ \alpha_{\textbf{with}\ \phi} \, [\!] \, \beta_{\textbf{with}\ \varphi} \xrightarrow{\epsilon} \alpha'_{\textbf{with}\ \phi'} \, [\!] \, \beta_{\textbf{with}\ \varphi} \\ \qquad \beta_{\textbf{with}\ \varphi} \, [\!] \, \alpha_{\textbf{with}\ \phi} \xrightarrow{\epsilon} \beta_{\textbf{with}\ \varphi} \, [\!] \, \alpha'_{\textbf{with}\ \phi'}\end{array}}$$

$$\frac{\Box}{\begin{array}{l}\rho\ \vdash\ < v, s, n >_{\textbf{with}\ \phi} \, [\!] \, \alpha_{\textbf{with}\ \phi'} \xrightarrow{\epsilon} < v, s, n >_{\textbf{with}\ \phi} \\ \qquad \alpha_{\textbf{with}\ \phi'} \, [\!] \, < v, s, n >_{\textbf{with}\ \phi} \xrightarrow{\epsilon} < v, s, n >_{\textbf{with}\ \phi}\end{array}}$$

Parallel combinator

$$\frac{\square}{\rho \ \vdash < E_1 \ {}_{n_1}\|_{n_2} \ E_2, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < E_1, s, n_1 >_{\mathbf{with}\ \phi} \| \ (s, n, n_1, n_2) \ \| < E_1, s, n_2 >_{\mathbf{with}\ \phi}}$$

$$\frac{\rho \ \vdash \ \alpha_{\mathbf{with}\ \phi} \xrightarrow{c!_{msgid}v} \alpha'_{\mathbf{with}\ \phi'} \qquad \rho \ \vdash \ \beta_{\mathbf{with}\ \varphi} \xrightarrow{c?_{msgid}v} \beta'_{\mathbf{with}\ \varphi'}}{\begin{array}{ll} \rho \ \vdash & \alpha_{\mathbf{with}\ \phi} \ \| \ (s, n, n_1, n_2) \ \| \ \beta_{\mathbf{with}\ \varphi} \ \xrightarrow{\epsilon} \ \alpha'_{\mathbf{with}\ \phi \ \cdot \ out(n_1, n_2, id)} \ \| \ (s, n, n_1, n_2) \\ & \qquad\qquad\qquad\qquad \| \ \beta'_{\mathbf{with}\ \varphi \ \cdot \ in(n_1, n_2, msgid)} \\ & \beta_{\mathbf{with}\ \varphi} \ \| \ (s, n, n_1, n_2) \ \| \ \alpha_{\mathbf{with}\ \phi} \ \xrightarrow{\epsilon} \ \beta'_{\mathbf{with}\ \varphi \ \cdot \ in(n_2, n_1, id)} \ \| \ (s, n, n_1, n_2) \\ & \qquad\qquad\qquad\qquad \| \ \alpha'_{\mathbf{with}\ \phi \ \cdot \ out(n_2, n_1, msgid)} \end{array}}$$

$$\frac{\rho \ \vdash \ \alpha_{\mathbf{with}\ \phi} \ \xrightarrow{\diamond} \ \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{ll} \rho \ \vdash & \alpha_{\mathbf{with}\ \phi} \ \| \ (s, n, n_1, n_2) \ \| \ \beta_{\mathbf{with}\ \varphi} \ \xrightarrow{\diamond} \ \alpha'_{\mathbf{with}\ \phi'} \ \| \ (s, n, n_1, n_2) \ \| \ \beta_{\mathbf{with}\ \varphi} \\ & \beta_{\mathbf{with}\ \varphi} \ \| \ (s, n, n_1, n_2) \ \| \ \alpha_{\mathbf{with}\ \phi} \ \xrightarrow{\diamond} \ \beta_{\mathbf{with}\ \varphi} \ \| \ (s, n, n_1, n_2) \ \| \ \alpha'_{\mathbf{with}\ \phi'} \end{array}}$$

$$\frac{\square}{\begin{array}{ll} \rho \ \vdash & \alpha_{\mathbf{with}\ \phi} \ \| \ (s, n, n_1, n_2) \ \| < v, s', n_2 >_{\mathbf{with}\ \varphi} \xrightarrow{\epsilon} \ \alpha_{\mathbf{with}\ \phi} \ \| \ (s, n, n_1, n_2) \ \| \ s'_{\mathbf{with}\ \varphi} \\ & < v, s', n_2 >_{\mathbf{with}\ \varphi} \| \ (s, n, n_1, n_2) \ \| \ \alpha_{\mathbf{with}\ \phi} \ \xrightarrow{\epsilon} \ s'_{\mathbf{with}\ \varphi} \ \| \ (s, n, n_1, n_2) \ \| \ \alpha_{\mathbf{with}\ \phi} \end{array}}$$

$$\frac{\rho \ \vdash \ \alpha_{\mathbf{with}\ \phi} \ \xrightarrow{\diamond} \ \alpha'_{\mathbf{with}\ \phi'}}{\begin{array}{ll} \rho \ \vdash & \alpha_{\mathbf{with}\ \phi} \ \| \ (s, n, n_1, n_2) \ \| \ s'_{\mathbf{with}\ \varphi} \ \xrightarrow{\diamond} \ \alpha'_{\mathbf{with}\ \phi'} \ \| \ (s, n, n_1, n_2) \ \| \ s'_{\mathbf{with}\ \varphi} \\ & s'_{\mathbf{with}\ \varphi} \ \| \ (s, n, n_1, n_2) \ \| \ \alpha_{\mathbf{with}\ \phi} \ \xrightarrow{\diamond} \ s'_{\mathbf{with}\ \varphi} \ \| \ (s, n, n_1, n_2) \ \| \ \alpha'_{\mathbf{with}\ \phi'} \end{array}}$$

$$\frac{\square}{\begin{array}{ll} \rho \ \vdash & < v, s'', n_1 >_{\mathbf{with}\ \phi} \| \ (s, n, n_1, n_2) \ \| \ s'_{\mathbf{with}\ \varphi} \ \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\mathbf{with}\ \phi \| \varphi} \\ & s'_{\mathbf{with}\ \varphi} \ \| \ (s, n, n_1, n_2) \ \| < v, s'', n_1 >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\mathbf{with}\ \phi \| \varphi} \end{array}}$$

Interlocking combinator

$$\dfrac{\square}{\rho \ \vdash < E_1 \ {}_{n_1}\!⫲_{n_2} E_2, s, n >_{\textbf{with } \phi} \xrightarrow{\epsilon} < E_1, s, n_1 >_{\textbf{with } \phi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ < E_1, s, n_2 >_{\textbf{with } \phi}}$$

$$\dfrac{\rho \ \vdash \ \alpha_{\textbf{with } \phi} \xrightarrow{c!_{msgid}v} \alpha'_{\textbf{with } \phi'} \qquad \rho \ \vdash \ \beta_{\textbf{with } \varphi} \xrightarrow{c?_{msgid}v} \beta'_{\textbf{with } \varphi'}}{\begin{array}{ll} \rho \ \vdash & \alpha_{\textbf{with } \phi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \beta_{\textbf{with } \varphi} \ \xrightarrow{\epsilon} \ \alpha'_{\textbf{with } \phi \ \cdot \ out(n_1,n_2,id)} \ ⫲ \ (s, n, n_1, n_2) \\ & \qquad\qquad ⫲ \ \beta'_{\textbf{with } \varphi \ \cdot \ in(n_1,n_2,msgid)} \\ & \beta_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \alpha_{\textbf{with } \phi} \ \xrightarrow{\epsilon} \ \beta'_{\textbf{with } \varphi \ \cdot \ in(n_2,n_1,id)} \ ⫲ \ (s, n, n_1, n_2) \\ & \qquad\qquad ⫲ \ \alpha'_{\textbf{with } \phi \ \cdot \ out(n_2,n_1,msgid)} \end{array}}$$

$$\dfrac{\rho \ \vdash \ \alpha_{\textbf{with } \phi} \ \xrightarrow{\epsilon} \ \alpha'_{\textbf{with } \phi'}}{\begin{array}{ll} \rho \ \vdash & \alpha_{\textbf{with } \phi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \beta_{\textbf{with } \varphi} \ \xrightarrow{\epsilon} \ \alpha'_{\textbf{with } \phi'} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \beta_{\textbf{with } \varphi} \\ & \beta_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \alpha_{\textbf{with } \phi} \ \xrightarrow{\epsilon} \ \beta_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \alpha'_{\textbf{with } \phi'} \end{array}}$$

$$\dfrac{\square}{\begin{array}{ll} \rho \ \vdash & \alpha_{\textbf{with } \phi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ < v, s', n_2 >_{\textbf{with } \varphi} \xrightarrow{\epsilon} \ \alpha_{\textbf{with } \phi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ s'_{\textbf{with } \varphi} \\ & < v, s', n_2 >_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \alpha_{\textbf{with } \phi} \ \xrightarrow{\epsilon} \ s'_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \alpha_{\textbf{with } \phi} \end{array}}$$

$$\dfrac{\rho \ \vdash \ \alpha_{\textbf{with } \phi} \ \xrightarrow{\diamond} \ \alpha'_{\textbf{with } \phi'}}{\begin{array}{ll} \rho \ \vdash & \alpha_{\textbf{with } \phi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ s'_{\textbf{with } \varphi} \ \xrightarrow{\diamond} \ \alpha'_{\textbf{with } \phi'} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ s'_{\textbf{with } \varphi} \\ & s'_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \alpha_{\textbf{with } \phi} \ \xrightarrow{\diamond} \ s'_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ \alpha'_{\textbf{with } \phi'} \end{array}}$$

$$\dfrac{\square}{\begin{array}{ll} \rho \ \vdash & < v, s'', n_1 >_{\textbf{with } \phi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ s'_{\textbf{with } \varphi} \ \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\textbf{with } \phi \| \varphi} \\ & s'_{\textbf{with } \varphi} \ ⫲ \ (s, n, n_1, n_2) \ ⫲ \ < v, s'', n_1 >_{\textbf{with } \phi} \xrightarrow{\epsilon} < v, merge(s, s', s''), n >_{\textbf{with } \phi \| \varphi} \end{array}}$$

Function

$$\frac{\square}{\rho\ \vdash < E_1\ E_2, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} (< E_1, s, n > E_2)_{\mathbf{with}\ \phi}}$$

$$\frac{\rho\ \vdash\ \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\rho\ \vdash\ (\alpha\ E)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} (\alpha'\ E)_{\mathbf{with}\ \phi'}}$$

$$\frac{\square}{\rho\ \vdash < \lambda\, id : \tau \bullet E, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < [\![\lambda\, id : \tau \bullet E, \rho]\!], s, n >_{\mathbf{with}\ \phi}}$$

$$\frac{\square}{\rho\ \vdash\ (< [\![\lambda\, id : \tau \bullet E_1, \rho_1]\!], s, n >\ \ E_2)_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} ([\![\lambda\, id : \tau \bullet E_1, \rho_1]\!] < E_2, s, n >)_{\mathbf{with}\ \phi}}$$

$$\frac{\rho\ \vdash\ \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\rho\ \vdash\ ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ \alpha)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ \alpha')_{\mathbf{with}\ \phi'}}$$

$$\frac{\square}{\rho\ \vdash\ ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ \ < v, s, n >)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} ([\![\lambda\, id : \tau \bullet E, \rho_1]\!]\ v)_{\mathbf{with}\ \phi}}$$

$$\frac{\rho_1 \dagger [id \mapsto v]\ \vdash\ \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\rho\ \vdash\ ([\![\lambda\, id : \tau \bullet \alpha, \rho_1]\!]\ v)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} ([\![\lambda\, id : \tau \bullet \alpha', \rho_1]\!]\ v)_{\mathbf{with}\ \phi'}}$$

$$\frac{\rho_1 \dagger [id \mapsto v]\ \vdash\ \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} < v', s, n >_{\mathbf{with}\ \phi'}}{\rho\ \vdash\ ([\![\lambda\, id : \tau \bullet \alpha, \rho_1]\!]\ v)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} < v', s, n >_{\mathbf{with}\ \phi'}}$$

Let expression

$$\frac{\square}{\rho\ \vdash < \mathbf{let}\ id = E_1\ \mathbf{in}\ E_2, s, n >_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} (\mathbf{let}\ id =< E_1, s,\ > \mathbf{in}\ E_2)_{\mathbf{with}\ \phi}}$$

$$\frac{\rho\ \vdash\ \alpha_{\mathbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\mathbf{with}\ \phi'}}{\rho\ \vdash\ (\mathbf{let}\ id = \alpha\ \mathbf{in}\ E)_{\mathbf{with}\ \phi} \xrightarrow{\diamond} (\mathbf{let}\ id = \alpha'\ \mathbf{in}\ E)_{\mathbf{with}\ \phi'}}$$

$$\frac{\square}{\rho\ \vdash\ (\mathbf{let}\ id =< v, s, n >\ \ \mathbf{in}\ E)_{\mathbf{with}\ \phi} \xrightarrow{\epsilon} < E[v/id], s, n >_{\mathbf{with}\ \phi}}$$

If expression

$$\frac{\Box}{\rho\ \vdash\ <\textbf{if}\ E\ \textbf{then}\ E_1\ \textbf{else}\ E_2, s, n>_{\textbf{with}\ \phi} \xrightarrow{\epsilon} (\textbf{if}\ <E, s, n>\ \textbf{then}\ E_1\ \textbf{else}\ E_2)_{\textbf{with}\ \phi}}$$

$$\frac{\rho\ \vdash\ \alpha_{\textbf{with}\ \phi} \xrightarrow{\diamond} \alpha'_{\textbf{with}\ \phi'}}{\rho\ \vdash\ (\textbf{if}\ \alpha\ \textbf{then}\ E_1\ \textbf{else}\ E_2)_{\textbf{with}\ \phi} \xrightarrow{\diamond} (\textbf{if}\ \alpha'\ \textbf{then}\ E_1\ \textbf{else}\ E_2)_{\textbf{with}\ \phi'}}$$

$$\frac{\Box}{\rho\ \vdash\ (\textbf{if}\ <\textbf{true}, s, n>\ \textbf{then}\ E_1\ \textbf{else}\ E_2)_{\textbf{with}\ \phi} \xrightarrow{\epsilon} <E_1, s, n>_{\textbf{with}\ \phi}}$$

$$\frac{\Box}{\rho\ \vdash\ (\textbf{if}\ <\textbf{false}, s, n>\ \textbf{then}\ E_1\ \textbf{else}\ E_2)_{\textbf{with}\ \phi} \xrightarrow{\epsilon} <E_2, s, n>_{\textbf{with}\ \phi}}$$

4.4 Relating Live Sequence Charts to RSL

4.4.1 Syntactical Restrictions

There are a number of problematic issues with conditions in LSCs as discussed in Appendix A. For that reason we choose to omit hot and cold conditions when relating an RSL specification to an LSC. This is done by removing all condition events from the PA_{LSC} term prior to checking satisfaction.

Since RSL only supports synchronous communication on channels, we restrict the relation to cover synchronous messages only. More specifically, if an LSC contains asynchronous messages no RSL specification can satisfy it.

4.4.2 Satisfaction Relation

Before we can define what it means for an RSL expression to satisfy a Live Sequence Chart, we introduce some auxiliary notions.

In most cases we do not want an LSC to constrain all parts of an RSL specification. Typically, we only want to constrain the sequence of a limited number of messages. For this reason we label each LSC with the set of events it constrains. We allow this set to contain events that are not mentioned in the chart. For an LSC *ch* this set is denoted $\mathcal{C}_{ch}$.

Below we need an event extraction function that yields the set of those event identifiers that occur in the PA_{LSC} term for an LSC. The event extraction function, $events : T(\Sigma_{PAc_\epsilon}) \rightarrow$

$\mathcal{P}Event$ is defined as

$$
\begin{aligned}
&events(\epsilon) = \emptyset\\
&events(in(n_1, n_2, m)) = \{m\}\\
&events(out(n_1, n_2, m)) = \{m\}\\
&events(ins(n_1, n_2, m)) = \{m\}\\
&events(outs(n_1, n_2, m)) = \{m\}\\
&events(hotcondition(cond)) = \emptyset\\
&events(coldcondition(cond)) = \emptyset\\
&events(X \cdot Y) = events(X) \cup events(Y)\\
&events(X + Y) = events(X) \cup events(Y)\\
&events(X \parallel Y) = events(X) \cup events(Y)\\
&events(X \rhd Y) = events(X) \cup events(Y)
\end{aligned}
$$

As explained above we do not check LSC conditions when making the relation to RSL. The function removing conditions, $clean : T(\Sigma_{PAc_\epsilon}) \to T(\Sigma_{PAc_\epsilon})$ is defined as

$$
\begin{aligned}
&remcond(\epsilon) = \epsilon\\
&remcond(in(n_1, n_2, m)) = in(n_1, n_2, m\\
&remcond(out(n_1, n_2, m)) = out(n_1, n_2, m)\\
&remcond(ins(n_1, n_2, m)) = in(n_1, n_2, m\\
&remcond(outs(n_1, n_2, m)) = out(n_1, n_2, m)\\
&remcond(hotcondition(cond)) = \epsilon\\
&remcond(coldcondition(cond)) = \epsilon\\
&remcond(X \cdot Y) = remcond(X) \cdot remcond(Y)\\
&remcond(X + Y) = remcond(X) + remcond(Y)\\
&remcond(X \parallel Y) = remcond(X) \parallel remcond(Y)\\
&remcond(X \rhd Y) = remcond(X) \rhd remcond(Y)
\end{aligned}
$$

Definition 4.4.1. A PA_{LSC} term, x, can *simulate* a PAc_ϵ term, y, notation $x \succeq y$, if

$$y \downarrow \Rightarrow x \downarrow \;\wedge \forall y' : y \;\xrightarrow{a}\; y' \Rightarrow \exists x' : x \;\xrightarrow{a}\; x' \wedge x' \succeq y'$$

□

Definition 4.4.2. A PA_{LSC} formula *cbh* is called a *communication behaviour* of an RSL expression E wrt. an initial store s_0, if and only if there exists a configuration α, such that

$$[\,] \;\vdash < E, s_0, n >_{\textbf{with}\ \epsilon} \quad (\overset{\diamond}{\rightarrow})^* \quad \alpha_{\textbf{with}\ cbh}$$

where $(\overset{\diamond}{\rightarrow})^*$ denotes the transitive closure of the transition relation. If α is of the form $< v, s, n >$, where v is a value literal or a lambda expression, *cbh* is called a *terminated behaviour.* □

We are now ready to define the satisfaction relations for universal and existential LSCs.

Definition 4.4.3. (Satisfaction for universal LSC) An RSL expression E *satisfies* a universal LSC, *ch*, if for any initial store, s_0, for any terminated behaviour, *cbh*, of E

there exists a PA_{LSC} term ϕ_{prefix} and a PA_{LSC} term ϕ_{suffix}, such that

$$events(\phi_{\text{prefix}}) \cap \mathcal{C}_{ch} = \emptyset$$
$$events(\phi_{\text{suffix}}) \cap \mathcal{C}_{ch} = \emptyset$$

and

$$\phi_{\text{prefix}} \cdot remcond(S_{LSC}[\![ch]\!]) \cdot \phi_{\text{suffix}} \succeq cbh$$

□

Definition 4.4.4. (Satisfaction for existential LSC) An RSL expression E *satisfies* an existential LSC, ch, if for any initial store, s_0, there exists a terminated behaviour, cbh, of E, a PA_{LSC} term ϕ_{prefix} and a PA_{LSC} term ϕ_{suffix},such that

$$events(\phi_{\text{prefix}}) \cap \mathcal{C}_{ch} = \emptyset$$
$$events(\phi_{\text{suffix}}) \cap \mathcal{C}_{ch} = \emptyset$$

and

$$\phi_{\text{prefix}} \cdot remcond(S_{LSC}[\![ch]\!]) \cdot \phi_{\text{suffix}} \succeq cbh$$

□

4.5 Relating Statecharts to RSL

4.5.1 Syntactical Restrictions

In Statecharts negative events, i.e. the absence of events, can be part of the trigger of a transition. In RSL there is no way of checking whether a message is available on a channel without actually performing an input. Thus, the absence of an event can not be detected. We therefore restrict the relation between Statecharts and RSL to cover only triggers with all positive events. Specifically, if a Statechart contains a negative event in a trigger, no RSL specification can satisfy it.

4.5.2 Satisfaction Relation

Similar to the approach for Live Sequence Charts, we now want a method of extracting from an RSL specification its communication behaviour in the form of an SPL term. We do this in two steps: first we extract the communication behaviour as a PA_{LSC} term using the procedure defined for LSCs and then apply a function translating a PA_{LSC} expression into an SPL expression.

Definition 4.5.1. Let $translate : PA_{LSC} \rightarrow SPL$ be the function defined by

$$translate(\epsilon) = \mathbf{0}$$
$$translate(in(s,r,m)) \cdot X) = \mathbf{0} \triangleright \langle\{m\},\emptyset\rangle.[\emptyset]\sigma(translate(X))$$
$$translate(ins(s,r,m)) \cdot X) = \mathbf{0} \triangleright \langle\{m\},\emptyset\rangle.[\emptyset]\sigma(translate(X))$$
$$translate(out(s,r,m)) \cdot X) = [\{m\}]\sigma(translate(X))$$
$$translate(outs(s,r,m)) \cdot X) = [\{m\}]\sigma(translate(X))$$
$$translate(X \parallel Y) = translate(X) \parallel translate(Y)$$
$$translate(X + Y) = translate(X) + translate(Y)$$

□

The result of the *translate* function may not be in a convenient form, so we define an additional function, *normalise*, that simplifies an SPL term.

Definition 4.5.2. Let $\mathit{normalise} : SPL \rightarrow SPL$ be the function defined by

$$\mathit{normalise}(\mathbf{0}) = \mathbf{0}$$
$$\mathit{normalise}(\mathbf{0} \triangleright \langle m, \emptyset \rangle.(X)) = (\mathbf{0} \triangleright \langle m, \emptyset \rangle.\mathit{normalise}(X)$$
$$\mathit{normalise}([m]\sigma(X)) = \begin{cases} [m]\sigma(\mathit{normalise}(X)) & \text{if } X \neq [n]\sigma(Y) \text{ for every } n \text{ and } Y \\ \mathit{normalise}([m \cup n]\sigma(Y)) & \text{if } X = [n]\sigma(Y) \text{ for some } n \text{ and } Y \end{cases}$$
$$\mathit{normalise}(X \parallel Y) = \mathit{normalise}(X) \parallel \mathit{normalise}(Y)$$
$$\mathit{normalise}(X + Y) = \mathit{normalise}(X) + \mathit{normalise}(Y)$$

□

We can now define the satisfaction relation for a Statechart. Unlike for LSC we do not allow prefixes and suffixes, since the single Statechart is supposed to provide the full specification of the communication behaviour of the object.

Definition 4.5.3. (Satisfaction for Statechart) An RSL expression E *satisfies* a Statechart, ch, if for any initial store, s_0, for any terminated behaviour, cbh, of E

$$S_{StC}[\![ch]\!] \succeq \mathit{normalise}(\mathit{translate}(cbh))$$

□

4.6 Checking Satisfaction

The two satisfaction criteria defined in Definition 4.4.3 and 4.5.3 both require checking that all behaviours of the RSL expression can be simulated by the semantics of the corresponding chart. In some situations the RSL expressions may have infinitely many behaviours, so in that case, this simple form of checking is not possible. We do not have a solution for this problem at present, but the idea of *common histories* of Haxthausen and Xia [27] may be a solution.

The idea behind common history is to derive a process algebraic term that captures all the possible behaviours of an RSL expression.

Another problem arises when processes are recursive as is often the case for Statecharts. In this case, it is not enough to simply perform the transitions to check satisfaction. If the processes eventually terminate, an inductive proof on the number of recursions may be used to prove satisfaction. If the processes are non-terminating there is no base case, so induction can not be used. In this case the more powerful principle of co-induction may be used.

4.7 Tool Support

Actually checking an RSL specification against a behavioral specification in the form of LSCs and Statecharts can be very tedious. For that reason, the methods defined above are of limited applicability without tool support.

Tools should be developed to extract the semantic terms from diagrams and RSL specifications and for checking the satisfaction relations.

It would also be convenient to have a way of translating an LSC or Statechart into a skeleton RSL specification. Such a procedure was studied at the beginning of the present project, but was deemed to be too restrictive as to the style of the RSL specification. The problem is that because of the expressiveness of RSL, there are many ways to specify essentially the same behaviour. An automatic conversion would force the software engineer to use one particular style.

Chapter 5

Development Method

5.1 Introduction

In this section we place the use of graphical and formal specification notations in the larger context of a development method covering the phases of software engineering from domain analysis to testing. This discussion is based loosely on elements from Sommerville [55] and Bjørner [6].

5.2 Domain Analysis

Domain Analysis is the study of the environment or context in which the system to be developed will operate. In this phase it is important for the software engineers to familiarise themselves with the processes and terms of the domain. To achieve this, the engineers talk with those people whose work will be affected by the new system, or who has other interests in the system, i.e. those people collectively referred to as the stake-holders. It is also important to establish if and how the system will interact with existing systems.

In this phase of development, descriptions are going to be written in natural language supplemented by informal diagrams. UML Use Case diagrams may also be employed in this phase to characterise the main scenarios of use.

5.3 Initial Requirements Development

The goal in initial requirements development is to identify the required interactions between the system and its environment and to divide the system into its constituent components.

As the first step in specifying the requirements, Live Sequence Charts defining the patterns of interaction, i.e. the protocols between the environment and the system should be developed. In the next step, Live Sequence Charts defining the internal communication between components in the system should be prepared. Once these charts are ready, the internal behaviour of the components should be explained using Statecharts. Here, the Live Sequence Charts are the glue that binds together the Statecharts of different components.

In order to ensure consistency, verification tools such as Rhapsody developed by I-Logix, should be run to check that the Live Sequence Charts and the Statecharts are compatible.

5.4 Formalisation of Requirements

By now the overall subdivision of the system and its behaviours are fixed. The formalisation of the requirements serves the purpose of making sure that the final requirements are precise and complete in terms of the functional requirements. This phase will add additional details to the initial specification from the previous phase.

In this phase, the requirements are formally specified using RSL. The development of the specification will be guided by the diagrams from the previous phase. Once the specification is finished, it must be checked against the Live Sequence Charts and Statecharts. This is done using the satisfaction relations defined in the previous chapter. As is discussed elsewhere in this report, tool support is necessary for this step to be feasible in practise.

The RSL specification is most easily made by defining one function – possibly calling other auxiliary function – for each component, i.e. for each Statechart. The system is then the parallel composition of the component functions.

5.5 Refinement/Design

Next, one or more steps of refinement may be carried out. Refinement will transform the formal specification into a style that is close to the implementation language. Typically, this will involve translating an applicative specification into an imperative one. The refinement step(s) may also introduce design and architecture decisions.

The refinement process is an integrated part of the RAISE method [50], which includes the necessary proof system and tools to prove correctness of a refinement step. Even if the correctness of refinement is not formally proved, it is still worth spending the extra time on the refinement steps, since flaws in the specification may be uncovered and it will ease the implementation.

5.6 Implementation

The implementation phase is concerned with going from specifications to software. The time spent doing refinements in the previous phase will typically be rewarded when the implementation is developed, since many of the implementation choices have already been made. Also, the closer the final specification is to the implementation language, the fewer mistakes are likely to be made.

In the case of RSL, and indeed other similar specification languages, tools exist that allow a specification, that is sufficiently concrete, to be automatically translated into a programming language.

5.7 Testing

In the first stage of testing, each component of the system should be tested individually, i.e. component-level black-box testing. The LSCs developed for the initial requirements are a valuable tool for this purpose, since they describe the required interactions between components. Therefore, the overall test scenarios can be identified from the charts. The particular test vectors to use must be identified from the formal specification.

In integration testing, i.e. system-level black-box testing, the LSCs are again valuable. In this case, it is the interactions between the environment and the system that are in focus. Again, overall test scenarios are identified from the charts, while the exact test vectors are elicited from the formal specification.

Chapter 6

Example Application: Two-Phase Commit Protocol

6.1 Introduction

We present an example of the application of the central parts of the development method discussed in the previous chapter. We include only the phases of initial requirements specification and formalisation, since the aim is to show how to develop an RSL specification from diagrams and how to check the satisfaction relations.

The example is the Two-Phase Commit protocol of Gray [17]. The description below is based on Sharp [54].

6.2 Description

In many forms of distributed systems, the need arises for a group of parties to reach an agreement to perform some action. Each party has the option of vetoing the action, in which case all the other parties must not perform the action. Another possibility is that one or more parties fail before either committing or vetoing the action. In that case, the action must also be aborted by all parties.

One application of this protocol is to implement distributed transactions. In this case, the parties must agree whether to commit or roll back the transaction, such that it is either performed by all parties or by none.

The protocol described here is centralised, since a single distinguished party acts as the coordinator. The remaining parties are slaves.

6.2.1 Protocol

We start by describing the communication that takes place in the protocol.

Figure 6.2 illustrates the situation in which both parties agree to commit the transaction. After the coordinator receives confirmation from both parties, they are informed of the outcome.

Figure 6.3 and 6.4 illustrate the two situations in which slave 1, respectively slave 2, causes the transaction to be aborted. Since the party that aborts already knows the transaction will be aborted, the subsequent abort message is only sent to the other party by the

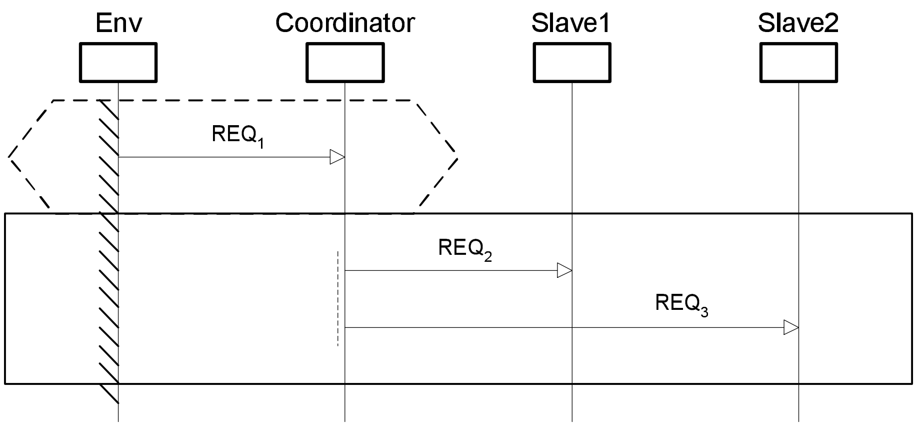


Figure 6.1: LSC for the initiation of the protocol.

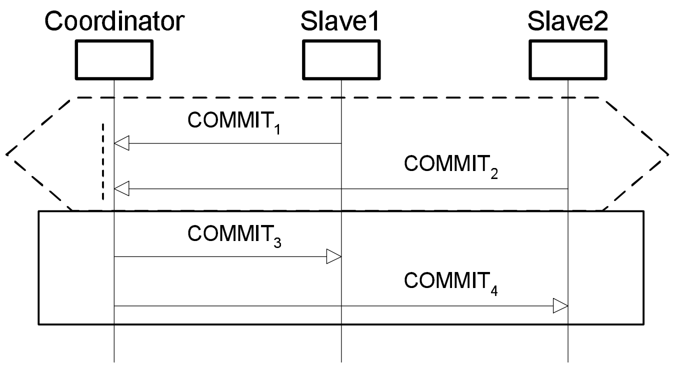


Figure 6.2: LSC commit.

coordinator. In case both processes abort, see 6.5, no confirmation messages are sent by the coordinator.

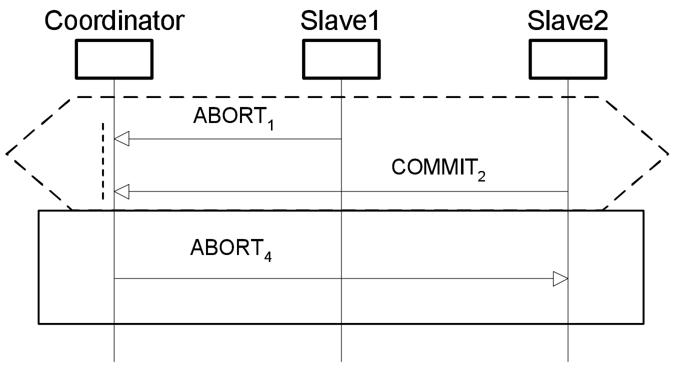


Figure 6.3: LSC abort by slave 1.

6.2.2 Internal Behaviour

Next, the internal states of the coordinator are described. In the initial state, the coordinator will wait for the user to request some action to be performed as a distributed transaction. The requested action is transmitted to the other parties. If both parties responds with commit, the transaction is committed. If at least one party responds with abort, the transaction is aborted.

The Statechart for the coordinator is shown in Figure 6.6.

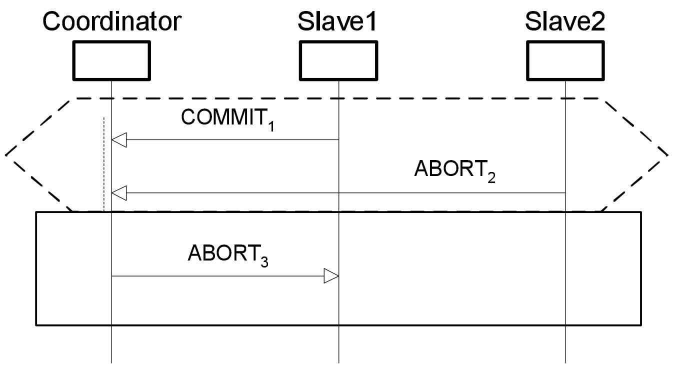


Figure 6.4: LSC abort by slave 2.

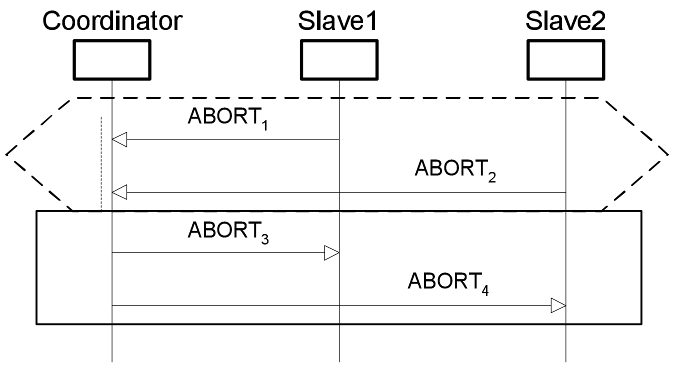


Figure 6.5: LSC abort by both slaves.

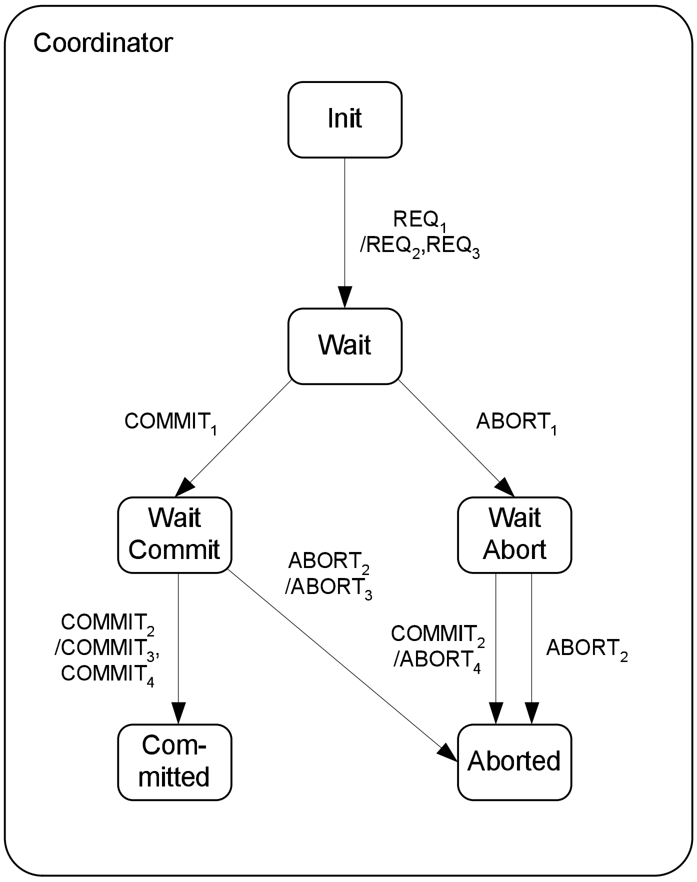


Figure 6.6: Statechart for the coordinator.

6.2.3 RSL Model

Now, we model the protocol in RSL. More specifically, we define processes for the system, the coordinator and the two slaves. The system process is just the parallel composition of the coordinator process and the two slave processes.

The coordinator process will wait to be invoked by inputting a request from the user. The requested action is transmitted to the two slaves. Next, the coordinator will input the responses from first slave 1 and then slave 2. If both choose to commit, they are informed that agreement has been reached to commit. Note how the function *commit* is supposed to abstract the actual action to be performed. If either slave responds with abort, the other slave is informed that the transaction is aborted and the coordinator performs the necessary clean-up, abstracted by the function *abort*.

The slave processes are entirely analogous. They first wait for a request to be received from the coordinator. Upon receipt, they decide – non-deterministically – to commit or abort. In the latter case, they tell the coordinator to abort and perform the necessary clean-up, abstracted by *abort*. In the former case, they tell the coordinator to commit and await the response. Based on the coordinator's response, they either commit or abort the transaction. The non-deterministic choice is an abstraction of the process used to decide whether to commit or abort. The details of this decision depends on the exact application of the protocol.

```
scheme TwoPhaseCommit =
  class
    type Req == REQ, Commit == COMMIT, Abort == ABORT

    channel uc, rcs1, rcs2 : Req, cs1c, cs2c : Commit, cs1a, cs2a : Abort

    variable c : Commit, a : Abort

    value
      System : Unit → in any out any write a, c Unit
      System() ≡ Coord() $_{Coordinator}$ ╫ $_{Slaves}$ (Slave1() $_{Slave1}$ ╫ $_{Slave2}$ Slave2()),

      Coord : Unit → in any out any write a, c Unit
      Coord() ≡
        let req = uc?$_{REQ_1}$ in
          rcs1!$_{REQ_2}$req ;
          rcs2!$_{REQ_3}$req ;
          ((c := cs1c?$_{COMMIT_1}$ ;
              ((c := cs2c?$_{COMMIT_2}$ ; cs1c!$_{COMMIT_3}$COMMIT
                  ; cs2c!$_{COMMIT_4}$COMMIT) ; commit())
              []
              (a := cs2a?$_{ABORT_2}$ ; cs1a!$_{ABORT_3}$ABORT ; abort()))
           []
           (a := cs1a?$_{ABORT_1}$ ;
              ((c := cs2c?$_{COMMIT_2}$ ; cs2a!$_{ABORT_4}$ABORT ; abort())
              []
              (a := cs2a?$_{ABORT_2}$ ; abort()))))
        end,

      Slave1 : Unit → in rcs1, cs1c, cs1a out cs1c, cs1a write a, c Unit
      Slave1() ≡
        let req = rcs1?$_{REQ_2}$ in
          (cs1c!$_{COMMIT_1}$COMMIT ; (c := cs1c?$_{COMMIT_3}$ ; commit())
                                $\square$ (a := cs1a?$_{ABORT_3}$ ; abort()))
          $\sqcap$
          (cs1a!$_{ABORT_1}$ABORT ; abort())
        end,

      Slave2 : Unit → in rcs2, cs2c, cs2a out cs2c, cs2a write a, c Unit
      Slave2() ≡
        let req = rcs2?$_{REQ_3}$ in
          (cs2c!$_{COMMIT_2}$COMMIT ; (c := cs2c?$_{COMMIT_4}$ ; commit())
                                $\square$ (a := cs2a?$_{ABORT_4}$ ; abort()))
          $\sqcap$
          (cs2a!$_{ABORT_2}$ABORT ; abort())
        end,

      commit : Unit → Unit,
      abort : Unit → Unit
  end
```

6.2.4 Checking Satisfaction

We will illustrate the process of checking satisfaction for the chart where the transaction is committed. The procedure for checking the remaining charts is entirely analogous.

The first step is to extract the semantics of the chart.

$$
\begin{aligned}
&[\![ch]\!] = \\
&\quad \lambda_{\emptyset,\emptyset}(((\mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \parallel \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2)) \\
&\qquad\quad \parallel \\
&\qquad\quad \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \\
&\qquad\quad \parallel \\
&\qquad\quad \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2)) \\
&\qquad \triangleright \\
&\qquad\quad ((\mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \parallel \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4)) \\
&\qquad\quad \parallel \\
&\qquad\quad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \\
&\qquad\quad \parallel \\
&\qquad\quad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4)) \\
&\qquad)
\end{aligned}
$$

Using the definition of $\lambda_{\emptyset,\emptyset}$ (Section 2.4.2, page 31) and the axioms for PAc_ϵ (Table 2.5, page 25) this can be rewritten to basic terms.

$$
\begin{aligned}
&[\![ch]\!] = \\
&\qquad\quad \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \\
&\qquad\quad \cdot\, \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \\
&\qquad\quad + \\
&\qquad\quad \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \\
&\qquad\quad \cdot\, \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \\
&\qquad \triangleright
\end{aligned}
$$

$$\begin{aligned}
&\mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3)\\
&\cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4)\\
&+\\
&\mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4)\\
&\cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3)\ )
\end{aligned}$$

Next, the behaviours of the RSL specification are extracted. It turns out there are four possible terminated behaviours, corresponding to the four possibilities COMMIT-COMMIT, ABORT-COMMIT, COMMIT-ABORT, ABORT-ABORT.

$$\begin{aligned}
&\mathit{cbh}_1 =\\
&\quad \lambda_{\emptyset,\emptyset}(\mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ}_1) \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2)\\
&\qquad \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1)\\
&\qquad \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3)\\
&\qquad \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2) \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1)\\
&\qquad \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2)\\
&\qquad \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_4)\\
&\qquad)
\end{aligned}$$

$$\begin{aligned}
&\mathit{cbh}_2 =\\
&\quad \lambda_{\emptyset,\emptyset}(\mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ}_1) \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2)\\
&\qquad \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1)\\
&\qquad \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT}_1) \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{ABORT}_2)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2) \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1)\\
&\qquad \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{ABORT}_2)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT}_1)\\
&\qquad)
\end{aligned}$$

$$\begin{aligned}
&\mathit{cbh}_3 =\\
&\quad \lambda_{\emptyset,\emptyset}(\mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ}_1) \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2)\\
&\qquad \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT}_3)\\
&\qquad \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{ABORT}_4)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2) \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT}_3)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2)\\
&\qquad \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{ABORT}_4)\\
&\qquad)
\end{aligned}$$

$$\begin{aligned}
&\mathit{cbh}_4 =\\
&\quad \lambda_{\emptyset,\emptyset}(\mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ}_1) \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2)\\
&\qquad \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT}_3)\\
&\qquad \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT}_1)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ}_2) \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT}_3)\\
&\qquad \|\\
&\qquad \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ}_3) \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT}_1)\\
&\qquad)
\end{aligned}$$

Using the definition of $\lambda_{\emptyset,\emptyset}$ (Section 2.4.2, page 31) and the axioms for PAc_ϵ (Table 2.5, page 25) the behaviours are rewritten to basic terms.

$$\begin{aligned} cbh_1 = {} & \mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ_1}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \\ & \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT_1}) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT_1}) \\ & \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT_2}) \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT_2}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT_3}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT_3}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT_4}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT_4})) \end{aligned}$$

$$\begin{aligned} cbh_2 = {} & \mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ_1}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \\ & \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT_1}) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT_1}) \\ & \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT_1}) \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT_1}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{ABORT_2}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{ABORT_2}) \end{aligned}$$

$$\begin{aligned} cbh_3 = {} & \mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ_1}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \\ & \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT_3}) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT_3}) \\ & \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT_2}) \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT_2}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{ABORT_4}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{ABORT_4})) \end{aligned}$$

$$\begin{aligned} cbh_4 = {} & \mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ_1}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \\ & \cdot \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT_3}) \cdot \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{ABORT_3}) \\ & \cdot \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT_1}) \cdot \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT_1}) \end{aligned}$$

Recall that by Definition 4.4.3 (page 56) we now have to find a prefix and a suffix to add to the semantics of the chart.

Starting with cbh_1, we find that the prefix should be

$$\begin{aligned} \phi_{\text{prefix}} = {} & \\ & \mathit{ins}(\mathit{env}, \mathit{Coordinator}, \mathit{REQ_1}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{REQ_2}) \\ & \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{REQ_3}) \end{aligned}$$

and that the suffix should be

$$\phi_{\text{suffix}} = \epsilon$$

Thus, we have to check that

$$\phi_{\text{prefix}} \cdot S_{LSC}[\![ch]\!] \cdot \phi_{\text{suffix}} \succeq cbh_1$$

It is clear that if two terms are equal, then one simulates the other. Since ϕ_{prefix} is a prefix of cbh_1 we can therefore remove ϕ_{prefix} from the front of both expressions. We denote cbh_1 without prefix as $\widehat{cbh_1}$. We are now ready to check that $S_{LSC}[\![ch]\!]$ simulates $\widehat{cbh_1}$. The procedure is illustrated in Figure 6.7. The aliases s_i and t_i for $1 \leq i, j \leq 8$ are defined below. On the right side of the diagram is the sequence of transitions that $\widehat{cbh_1}$ can evolve by. On the left is the matching sequence of transitions that $S_{LSC}[\![ch]\!]$ can evolve by. The labels in the middle are the actions for the transitions.

$$\begin{tikzcd}
S_{LSC}[\![ch]\!] \cdot \phi_{\text{suffix}} \arrow[dd] & \succeq & \widehat{cbh_1} \arrow[dd] \\
& \mathit{outs}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) & \\
s_7 \arrow[dd] & \succeq & t_7 \arrow[dd] \\
& \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) & \\
s_6 \arrow[dd] & \succeq & t_6 \arrow[dd] \\
& \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) & \\
s_5 \arrow[dd] & \succeq & t_5 \arrow[dd] \\
& \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) & \\
s_4 \arrow[dd] & \succeq & t_4 \arrow[dd] \\
& \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) & \\
s_3 \arrow[dd] & \succeq & t_3 \arrow[dd] \\
& \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) & \\
s_2 \arrow[dd] & \succeq & t_2 \arrow[dd] \\
& \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) & \\
s_1 \arrow[dd] & \succeq & t_1 \arrow[dd] \\
& \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) & \\
\epsilon & \succeq & \epsilon
\end{tikzcd}$$

Figure 6.7: Checking simulation for cbh_1.

$$\begin{aligned}
s_1 &= \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \\
s_2 &= \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \cdot s_1 \\
s_3 &= \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \cdot s_2 \\
s_4 &= \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \cdot s_3 \\
s_5 &= s_5' \rhd s_4 + s_4' \\
s_6 &= s_6' \rhd s_4 + s_4' \\
s_7 &= (\mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \cdot s_6') \rhd s_4 + s_4' \\
s_4' &= \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \\
&\quad \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \\
&\quad \cdot \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \\
&\quad \cdot \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \\
s_5' &= \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \\
s_6' &= \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \cdot s_5'
\end{aligned}$$

$$\begin{aligned}
t_1 &= \mathit{ins}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \\
t_2 &= \mathit{outs}(\mathit{Coordinator}, \mathit{Slave2}, \mathit{COMMIT}_4) \cdot t_1 \\
t_3 &= \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \cdot t_2 \\
t_4 &= \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{COMMIT}_3) \cdot t_3 \\
t_5 &= \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \cdot t_4 \\
t_6 &= \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{COMMIT}_2) \cdot t_5 \\
t_7 &= \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT}_1) \cdot t_6
\end{aligned}$$

$$\begin{tikzcd}
S_{LSC}\llbracket ch \rrbracket \cdot \phi_{\text{suffix}} \arrow[dd] & \succeq & \widehat{cbh_2} \arrow[dd] \\
 & outs(Slave1, Coordinator, COMMIT_1) & \\
u_5 \arrow[dd] & \succeq & v_5 \arrow[dd] \\
 & ins(Slave1, Coordinator, COMMIT_1) & \\
u_4 \arrow[dd] & \succeq & v_4 \arrow[dd] \\
 & outs(Slave2, Coordinator, ABORT_1) & \\
u_3 \arrow[dd] & \succeq & v_3 \arrow[dd] \\
 & ins(Slave2, Coordinator, ABORT_1) & \\
u_2 \arrow[dd] & \succeq & v_2 \arrow[dd] \\
 & outs(Coordinator, Slave1, ABORT_2) & \\
u_1 \arrow[dd] & \succeq & v_1 \arrow[dd] \\
 & ins(Coordinator, Slave1, ABORT_2) & \\
\epsilon & \succeq & \epsilon
\end{tikzcd}$$

Figure 6.8: Checking simulation for cbh_2.

We now repeat the procedure for cbh_2. The prefix is the same as for cbh_1. The suffix is

$$\begin{aligned}
\phi_{\text{suffix}} = {} & \\
& ins(Slave2, Coordinator, ABORT_1) \\
& \cdot outs(Coordinator, Slave1, ABORT_2) \cdot ins(Coordinator, Slave1, ABORT_2)
\end{aligned}$$

Again, let $\widehat{cbh_2}$ be the result of removing ϕ_{prefix} from the front of cbh_2.

Thus, we have to check that

$$S_{LSC}\llbracket ch \rrbracket \cdot \phi_{\text{suffix}} \succeq \widehat{cbh_2}$$

Figure 6.8 illustrates the procedure. The aliases u_i and v_i for $1 \leq i \leq 5$ are defined below.

$$\begin{aligned}
u_1 &= ins(Coordinator, Slave1, ABORT_2) \\
u_2 &= outs(Coordinator, Slave1, ABORT_2) \cdot u_1 \\
u_3 &= ins(Slave2, Coordinator, ABORT_1) \cdot u_2 \\
u_4 &= (u'_4 \rhd u') \cdot u_3 \\
u_5 &= (ins(Slave1, Coordinator, COMMIT_1) \cdot u'_4 \rhd u') \cdot u_3 \\
u'_4 &= outs(Slave2, Coordinator, COMMIT_2) \\
&\quad \cdot ins(Slave2, Coordinator, COMMIT_2) \\
u' &= outs(Coordinator, Slave1, COMMIT_3) \\
&\quad \cdot ins(Coordinator, Slave1, COMMIT_3) \\
&\quad \cdot outs(Coordinator, Slave2, COMMIT_4) \\
&\quad \cdot ins(Coordinator, Slave2, COMMIT_4) \\
&\quad + \\
&\quad outs(Coordinator, Slave2, COMMIT_4) \\
&\quad \cdot ins(Coordinator, Slave2, COMMIT_4) \\
&\quad \cdot outs(Coordinator, Slave1, COMMIT_3) \\
&\quad \cdot ins(Coordinator, Slave1, COMMIT_3)
\end{aligned}$$

$$
\begin{aligned}
v_1 &= \mathit{ins}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{ABORT_2})\\
v_2 &= \mathit{outs}(\mathit{Coordinator}, \mathit{Slave1}, \mathit{ABORT_2}) \cdot v_1\\
v_3 &= \mathit{ins}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT_1}) \cdot v_2\\
v_4 &= \mathit{outs}(\mathit{Slave2}, \mathit{Coordinator}, \mathit{ABORT_1}) \cdot v_3\\
v_5 &= \mathit{ins}(\mathit{Slave1}, \mathit{Coordinator}, \mathit{COMMIT_1}) \cdot v_4
\end{aligned}
$$

The two remaining behaviours, cbh_3 and cbh_4, can be checked using the same technique, but we omit the details.

Now, we must verify that the specification implements the Statechart.

First, the semantics is extracted from the Statechart.

$$
\begin{aligned}
&[\![\mathit{Coordinator}]\!] = \widehat{\mathit{Init}}\\
&\qquad \widehat{\mathit{Init}} = [\![\mathit{Init}]\!] \triangleright \langle\{REQ_1\}, \emptyset\rangle.[\{REQ_2, REQ_3\}]\sigma(\widehat{\mathit{Wait}})\\
&\qquad \widehat{\mathit{Wait}} = [\![\mathit{Wait}]\!] \triangleright \langle\{COMMIT_1\}, \emptyset\rangle.[\emptyset]\sigma(\widehat{\mathit{WaitCommit}})\\
&\qquad\qquad\qquad + \langle\{ABORT_3\}, \emptyset\rangle.[\emptyset]\sigma(\widehat{\mathit{WaitAbort}})\\
&\qquad \widehat{\mathit{WaitCommit}} = [\![\mathit{WaitCommit}]\!] \triangleright \langle\{COMMIT_2\}, \emptyset\rangle.[\{COMMIT_3, COMMIT_4\}]\sigma(\widehat{\mathit{Comitted}})\\
&\qquad\qquad\qquad + \langle\{ABORT_1\}, \emptyset\rangle.[\{ABORT_2\}]\sigma(\widehat{\mathit{Aborted}})\\
&\qquad \widehat{\mathit{WaitAbort}} = [\![\mathit{WaitAbort}]\!] \triangleright \langle\{COMMIT_2\}, \emptyset\rangle.[\{ABORT_2\}]\sigma(\widehat{\mathit{Aborted}})\\
&\qquad\qquad\qquad + \langle\{ABORT_1\}, \emptyset\rangle.[\{ABORT_4\}]\sigma(\widehat{\mathit{Aborted}})\\
&\qquad \widehat{\mathit{Committed}} = [\![\mathit{Committed}]\!]\\
&\qquad \widehat{\mathit{Aborted}} = [\![\mathit{Aborted}]\!]\\
&[\![\mathit{Init}]\!] = \mathbf{0}\\
&[\![\mathit{Wait}]\!] = \mathbf{0}\\
&[\![\mathit{WaitCommit}]\!] = \mathbf{0}\\
&[\![\mathit{WaitAbort}]\!] = \mathbf{0}\\
&[\![\mathit{Committed}]\!] = \mathbf{0}\\
&[\![\mathit{Aborted}]\!] = \mathbf{0}
\end{aligned}
$$

By substituting process variables with their values, we get the more compact expression

$$
\begin{aligned}
[\![\mathit{Coordinator}]\!] = \mathbf{0} \triangleright &\langle\{REQ_1\}, \emptyset\rangle.[\{REQ_2, REQ_3\}]\sigma(\\
&\mathbf{0} \triangleright \langle\{COMMIT_1\}, \emptyset\rangle.[\emptyset]\sigma(\\
&\qquad \mathbf{0} \triangleright \langle\{COMMIT_2\}, \emptyset\rangle.[\{COMMIT_3, COMMIT_4\}]\sigma(\mathbf{0})\\
&\qquad\quad + \langle\{ABORT_1\}, \emptyset\rangle.[\{ABORT_2\}]\sigma(\mathbf{0}))\\
&\quad + \langle\{ABORT_3\}, \emptyset\rangle.[\emptyset]\sigma(\\
&\qquad \mathbf{0} \triangleright \langle\{COMMIT_2\}, \emptyset\rangle.[\{ABORT_2\}]\sigma(\mathbf{0})\\
&\qquad\quad + \langle\{ABORT_1\}, \emptyset\rangle.[\{ABORT_4\}]\sigma(\mathbf{0})))
\end{aligned}
$$

We now extract the SPL expressions that describes the behaviour of the RSL specification. In this case we are not interested in the behaviour of the whole system, since the Statechart only describes the behaviour of the Coordinator. Therefore, we extract the behavioural SPL term by first finding the terminated behaviours of the *Coordinator* process. As was the case above for the system as a whole, there are four possible behaviours. We will only show the procedure for the first of these, named cbh_5.

$$\begin{aligned}
&cbh_5 = \\
&\quad ins(env, Coordinator, REQ_1) \cdot outs(Coordinator, Slave1, REQ_2) \\
&\quad \cdot outs(Coordinator, Slave2, REQ_3) \cdot ins(Slave1, Coordinator, COMMIT_1) \\
&\quad \cdot ins(Slave2, Coordinator, COMMIT_2) \cdot outs(Coordinator, Slave1, COMMIT_3) \\
&\quad \cdot outs(Coordinator, Slave2, COMMIT_4)
\end{aligned}$$

Next, we translate the PA_{LSC} term into SPL using the functions *translate* (Definition 4.5.1) and *normalise* (Definition 4.5.2).

$$\begin{aligned}
translate(cbh_5) = \mathbf{0} \triangleright \langle\{REQ_1\}, \emptyset\rangle.[\emptyset]\sigma([\{REQ_2\}]\sigma([\{REQ_3\}]\sigma(\\
\mathbf{0} \triangleright \langle\{COMMIT_1\}, \emptyset\rangle.[\emptyset]\sigma(\mathbf{0} \triangleright \langle\{COMMIT_2\}, \emptyset\rangle.[\emptyset]\sigma(\\
[\{COMMIT_3\}]\sigma([\{COMMIT_4\}]\sigma(\mathbf{0})))))))
\end{aligned}$$

$$\begin{aligned}
\widehat{cbh}_5 = normalise(translate(cbh_5)) = \mathbf{0} \triangleright \langle\{REQ_1\}, \emptyset\rangle.[\{REQ_2, REQ_3\}]\sigma(\\
\mathbf{0} \triangleright \langle\{COMMIT_1\}, \emptyset\rangle.[\emptyset]\sigma(\mathbf{0} \triangleright \langle\{COMMIT_2\}, \emptyset\rangle.[\{COMMIT_3, COMMIT_4\}]\sigma(\mathbf{0})))
\end{aligned}$$

It is easy to see that if all the nondeterministic choices in the SPL expression for the Coordinator Statechart are resolved in favour of the left operand, then the resulting SPL term is identical to $\widehat{cbh}_5$. Therefore, the RSL specification satisfies the Statechart.

Chapter 7

Conclusion

The aim of this MSc project was to study ways of integrating graphical specification techniques with formal specification languages. We have presented the syntax and semantics of two such graphical specification techniques, namely Live Sequence Charts and Statecharts. We have defined a method of using diagrams expressed in these graphical notations to constrain a formal specification given in a subset of the RAISE Specification Language (RSL), and proposed a development method applying these principles in combination with the recommended development method for RSL.

For both Live Sequence Charts and Statecharts the semantics is traditionally defined using Büchi automata. However, in this report we have presented semantics' based on process algebra. The justification for this choice is that the process algebras provide a more compact notation while offering an established technique for checking behavioural equivalence.

The semantics for Live Sequence Charts is an extension of the existing standardised semantics of Message Sequence Charts, from which Live Sequence Charts are derived. To the best of our knowledge, this work represents the first attempt to define the semantics of Live Sequence Charts using a process algebra. The distinction between universal and existential charts is not incorporated into the semantics. Rather, this distinction is made when the diagrams are related to an RSL specification.

The semantics for Statecharts is taken from the literature. Unlike many other semantics' for Statecharts it is characterised by possessing the three desirable properties of synchrony, compositionality and causality.

Several approaches to the integration of Live Sequence Charts and Statecharts with RSL were investigated before the linking approach as described in this thesis was adopted. The linking approach means that there is no direct translation between the notations. Instead, the diagrams are considered as constraints against which the RSL specification should be checked. This approach seems to give the software engineer the greatest freedom in choosing the style of specification to use. Moreover, the idea of using diagrams as constraints on the specification mimics the way requirements are used in the development process: they guide the development of the design and implementation and are used to verify that the implementation behaves in the way it is required to do.

The proposed development method combines elements of a typical informal development method oriented towards graphical notations, such as the widely used UML method, with the formal development method of RAISE. Essentially, the diagrams replace the initial, very abstract specifications in the RAISE method.

Referring back to the hypothesis stated in the introduction the obvious question is: to what extent has the hypothesis been confirmed? Clearly, the techniques presented in this work are not complete and mature enough to be directly employed in software projects of realistic scale. There are still several open questions and unsolved problems regarding the relations between diagrams and formal specifications. However, this work has shown a promising direction that warrants further research. The main benefit of combining diagrams with formal specifications is that domain experts, who are not software experts, can validate the requirements, since the graphical notations are rather easy to become familiar with, as opposed to formal specification languages, which require extensive training. Also, the diagrams help highlight important aspects of a prospective system, while abstracting other aspects away. Ideally, methods similar to the one described in this thesis will lead to more widespread use of formal methods in software engineering, offering some hope of countering the so-called software crisis.

There are several topics for future work. The satisfaction relations between Live Sequence Charts and RSL and between Statecharts and RSL are somewhat problematic, since checking them can be difficult. When processes are recursive, a simple evaluation is not sufficient. If the processes terminate, the satisfaction relation can be proved using an inductive proof on the number of recursions. If the processes are non-terminating, induction is not strong enough. Possibly, the dual notion of co-induction may be used in this case. The notion of common histories may also be of use in alleviating this problem.

A wholly different approach to the integration of graphical notations with formal specification, is to develop new types of diagrams tailored specifically to coexist with formal methods.

A prerequisite for any kind of industrial applications of this method is the development of tool support. These tools should automate the translation of the diagrams into corresponding process algebra terms, extract communication behaviours of the RSL specification and check the satisfaction relation between the diagrams and the specification. Furthermore, tool support may be provided for converting a (collection of) diagram(s) into a skeleton specification.

Appendix A

A Critique of Live Sequence Charts

The definition of Live Sequence Charts [11] leaves a number of question open for interpretation. These questions must be answered if LSCs are to be used for software development based on formal methods or tool support.

Damm and Harel propose Live Sequence Charts as a notation for specifying inter-object or inter-process behaviour. Their development method combines LSCs with Statecharts, which specify the intra-object or intra-process behaviour. A complete systems specification will thus contain a number of LSCs and a Statechart for each instance referenced in the LSCs.

The semantics for LSC given by Damm and Harel treats conditions as events just like message input or output events. Thus, it is not specified when or how conditions should be evaluated. Specifically, it is not clear whether conditions are evaluated once or whether they somehow wait for the condition to become satisfied. Furthermore, since local actions that may change local variables are allowed anywhere between non-local events (i.e. conditions, message events, etc.), the Boolean expression of a condition is not guaranteed to hold as an invariant for the next event. In principle, a condition could hold at the moment it is evaluated, but be false when the next event happens. Therefore, the meaning of conditions is unclear.

Another problem regarding conditions is that cold conditions allow the system to terminate successfully without completing the rest of the chart. Essentially, this means that what follows a cold condition places no constraints on a system implementing the specification.

The last issue concerning conditions is the use of simultaneous conditions on more than one instance. Because the condition is a Boolean expression over the visible variables of the chart, some form of global variables is presupposed. This would not seem to match the reality of a distributed system, which is an important application area for LSCs. Indeed, in such a system all communication between processes or objects would be via messages. In an example, Damm and Harel [11] use LSC for specifying a driver-less Rail-car system originally proposed by Harel and Gery [23]. In their example a proximity sensor is supposed to notify the driver-less car that it is approaching a station. To do this, they place a simultaneous condition on the proximity sensor and the car, stating that the car should be cruising. In the interest of encapsulation and separation of concerns, the behaviour of the proximity sensor should not be guided by the internal state of the rail-car. A more realistic model for this situation is that the car sends a signal to the proximity sensor. This more closely matches the physics of the situation: the magnetic field of the rail-car induces a current in the sensor, which is interpreted as the car passing by. However, one may argue that replacing simultaneous conditions with message passing clutters the specification.

Another problematic issue of LSCs is the use of mandatory versus optional progress. Similar to the issue with cold conditions discussed above, optional progress allows a correct implementation to omit the rest of the specification.

The inclusion of timers inherited from Message Sequence Charts seems to oppose the inter-object orientation of LSC. Timers specify (non-quantified) timing constraints on a single instance. Because there is no notion of quantifiable time, these timing constraints have no semantical import. If the definition of LSC were extended to include quantifiable time (as is done by Klose and Wittke [38] for example) more general timing constraints on the occurrence of causally ordered events within an instance and between instances would be more useful.

Finally, the ability to specify hot and cold messages, i.e. whether a message is required to be received or may disappear, is redundant because of the facility for describing hot and cold locations. Essentially, the temperature of the locations take precedence over the temperature of the message, so whether or not the message is received is determined entirely by the temperature of the message input. This questionable feature of LSC is recognised by Harel and Marelly [25] who list the possible cases and conclude that the temperature of the message has no semantical meaning.

Appendix B

Proofs

B.1 Proof of Theorem 2.3.9 – Termination of PA_ϵ

In the proof of theorem 2.3.9 on page 20 we showed that the rewriting rules RA4 and RA5 satisfy the lexicographical recursive path ordering. We now give the remaining derivations.

RA3

$$\begin{aligned} x + x &>_{lpo} x +^* x && \text{RPO1} \\ &>_{lpo} x && \text{RPO2} \end{aligned}$$

RA6

$$\begin{aligned} x + \delta &>_{lpo} \delta \lfloor\!\lfloor^* x && \text{RPO1} \\ &>_{lpo} \delta && \text{RPO2} \end{aligned}$$

RA7

$$\begin{aligned} \delta \cdot x &>_{lpo} \delta \cdot^* x && \text{RPO1} \\ &>_{lpo} \delta && \text{RPO2} \end{aligned}$$

RA8

$$\begin{aligned} x \cdot \epsilon &>_{lpo} x \cdot^* \epsilon && \text{RPO1} \\ &>_{lpo} x && \text{RPO2} \end{aligned}$$

RA9

$$\begin{aligned} \epsilon \cdot x &>_{lpo} \epsilon \cdot^* x && \text{RPO1} \\ &>_{lpo} x && \text{RPO2} \end{aligned}$$

RF1

$$\begin{aligned} x \parallel y &>_{lpo} x \parallel^* y && \text{RPO1} \\ &>_{lpo} (x \parallel^* y) + (x \parallel^* y) && \text{RPO2} \\ &>_{lpo} (x \parallel^* y) + ((x \parallel^* y) + (x \parallel^* y)) && \text{RPO2} \\ &>_{lpo} ((x \parallel^* y) \lfloor\!\lfloor (x \parallel^* y)) + (((x \parallel^* y) \lfloor\!\lfloor (x \parallel^* y)) + ((x \parallel^* y) \cdot (x \parallel^* y))) && \text{RPO2} \\ &>_{lpo} x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + \surd((x \parallel^* y)) \cdot \surd((x \parallel^* y)) && \text{RPO2, RPO3} \\ &>_{lpo} x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + \surd(x) \cdot \surd(y) && \text{RPO3} \end{aligned}$$

RF2

$$\begin{aligned} \epsilon \mathbin{\lfloor\!\lfloor} x &>_{lpo} \epsilon \mathbin{\lfloor\!\lfloor}^{*} x && \text{RPO1} \\ &>_{lpo} \delta && \text{RPO2} \end{aligned}$$

RF3

$$\begin{aligned} \delta n \mathbin{\lfloor\!\lfloor} x &>_{lpo} \epsilon \mathbin{\lfloor\!\lfloor}^{*} x && \text{RPO1} \\ &>_{lpo} \delta && \text{RPO3} \end{aligned}$$

RF4: this case cannot be proved with the partial ordering used so far. We skip this rewrite rule for now and return to it below.

RF4'

$$\begin{aligned} a \mathbin{\lfloor\!\lfloor} x &>_{lpo} a \mathbin{\lfloor\!\lfloor}^{*} x && \text{RPO1} \\ &>_{lpo} (a \mathbin{\lfloor\!\lfloor}^{*} x) \cdot (a \mathbin{\lfloor\!\lfloor}^{*} x) && \text{RPO2} \\ &>_{lpo} a \cdot x && \text{RPO3,RPO5} \end{aligned}$$

RF5

$$\begin{aligned} (x+y) \mathbin{\lfloor\!\lfloor} z &>_{lpo} (x+y) \mathbin{\lfloor\!\lfloor}^{*} z && \text{RPO1} \\ &>_{lpo} ((x+y) \mathbin{\lfloor\!\lfloor}^{*} z) + ((x+y) \mathbin{\lfloor\!\lfloor}^{*} z) && \text{RPO2} \\ &>_{lpo} ((x +^{*} y) \mathbin{\lfloor\!\lfloor} z) + ((x +^{*} y) \mathbin{\lfloor\!\lfloor} z) && \text{RPO4} \\ &>_{lpo} x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z && \text{RPO3} \end{aligned}$$

RT1

$$\begin{aligned} \surd(\epsilon) &>_{lpo} \surd^{*}(\epsilon) && \text{RPO1} \\ &>_{lpo} \epsilon && \text{RPO3} \end{aligned}$$

RT2

$$\begin{aligned} \surd(\delta) &>_{lpo} \surd^{*}(\delta) && \text{RPO1} \\ &>_{lpo} \delta && \text{RPO3} \end{aligned}$$

RT3

$$\begin{aligned} \surd(a \cdot x) &>_{lpo} \surd^{*}(a \cdot x) && \text{RPO1} \\ &>_{lpo} \delta && \text{RPO2} \end{aligned}$$

RT4

$$\begin{aligned} \surd(x+y) &>_{lpo} \surd^{*}(x+y) && \text{RPO1} \\ &>_{lpo} (\surd^{*}(x+y)) + (\surd^{*}(x+y)) && \text{RPO2} \\ &>_{lpo} (\surd(x +^{*} y)) + (\surd(x +^{*} y)) && \text{RPO4} \\ &>_{lpo} \surd(x) + \surd(y) && \text{RPO2} \end{aligned}$$

We now return to the case of RF4. The problem in this case is that we would like to have $\mathbin{\lfloor\!\lfloor} > \|$ so that we can use RPO2 to introduce $\|$, but this would conflict with RF1 where we want $\| > \mathbin{\lfloor\!\lfloor}$.

The solution found by Bergstra and Klop [5] is to adorn the operators $\|$ and $\mathbin{\lfloor\!\lfloor}$ with a rank. The intuition is that we should allow *parallel* to be introduced using RPO2, but only if the complexity of the arguments are reduced at the same time. The rank is a measure of the complexity of the arguments. First, weights of terms are introduced.

Definition B.1.1. Let x and y be terms in PA_ϵ and let a be an atomic action. The weight of a term x, denoted $|x|$, is defined inductively by

$$\begin{aligned}
&|a| = 1\\
&|\delta| = 1\\
&|\epsilon| = 1\\
&|\surd(x)| = |x|\\
&|x + y| = \max\{|x|, |y|\}\\
&|x \cdot y| = |x| + |y|\\
&|x \parallel y| = |x| + |y|\\
&|x \mathbin{\lfloor\!\lfloor} y| = |x| + |y|
\end{aligned}$$

□

Definition B.1.2. The rank of an operator $\parallel$ or $\lfloor\!\lfloor$ is the weight of subterm of which it is the leading operator. In the definition of terms over PA_ϵ the $\parallel$ and $\lfloor\!\lfloor$ operators are removed and the following families of operators added.

- $\{\parallel_n | n \geq 2\}$
- $\{\lfloor\!\lfloor_n | n \geq 2\}$

where in both cases n is the sum of the weights of the two arguments to the operator. □

We now extend the partial order on the signature Σ_{PA_ϵ} to

$$\delta < \epsilon < \surd < + < \cdot < \lfloor\!\lfloor_2 < \parallel_2 < \lfloor\!\lfloor_3 < \parallel_3 < \dots$$

Let $n = |x| + |y|$, then

$$\begin{aligned}
a \cdot x \mathbin{\lfloor\!\lfloor}_{n+1} y &>_{lpo} a \cdot x \mathbin{\lfloor\!\lfloor}^*_{n+1} y && \text{RPO1}\\
&>_{lpo} (a \cdot x \mathbin{\lfloor\!\lfloor}^*_{n+1} y) \cdot (a \cdot x \mathbin{\lfloor\!\lfloor}^*_{n+1} y && \text{RPO2}\\
&>_{lpo} (a \cdot x) \cdot ((a \cdot x \mathbin{\lfloor\!\lfloor}^*_{n+1} y) \parallel_n (a \cdot \lfloor\!\lfloor^*_{n+1} y)) && \text{RPO2, RPO3}\\
&>_{lpo} (a \cdot^* x) \cdot (a \cdot^* x \parallel_n y) && \text{RPO1, RPO3}\\
&>_{lpo} a \cdot (x \parallel_n y) && \text{RPO3}
\end{aligned}$$

Bibliography

[1] Baeten, J. C. M., and Verhoef, C. A congruence theorem for structured operational semantics with predicates. In *Proceedings CONCUR 93, Hildesheim, Germany* (1993), vol. 715 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 477–492.

[2] Baeten, J. C. M., and Verhoef, C. Concrete process algebra. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 4: Semantic Modelling. Oxford University Press, 1995, ch. 2. .

[3] Baeten, J. C. M., and Weijland, W. P. *Process Algebra.* No. 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[4] Ben-Abdallah, H., and Leue, S. Expressing and analyzing timing constraints in Message Sequence Chart specifications. Tech. Rep. 97-04, Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, 1997.

[5] Bergstra, J. A., and Klop, J. W. Algebra of communicating processes with abstraction. *Theoretical Computer Science 37*, 1 (1985), 77–121.

[6] Bjørner, D. *Software Engineering*, vol. 3: From Domains via Requirements to Software. 2003–2004. To be published.

[7] Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[8] Büssow, R., Geisler, R., and Klar, M. Specifying safety-critical embedded systems with Statecharts and Z: A case study. In *Fundamental Approaches to Software Engineering: First International Conference, FASE'98, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March/April 1998* (1998), E. Astesiano, Ed., vol. 1382 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 71–87.

[9] CCITT. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.

[10] Chaochen, Z., and Hansen, M. R. *Duration Calculus: A formal approach to real–time systems.* Monographs in Theoretical Computer Science. Springer–Verlag, 2003.

[11] Damm, W., and Harel, D. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design 19* (2001), 45–80. Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems* (FMOODS'99), Kluwer, 1999, pp. 293–312.

[12] Dershowitz, N., and Jouannaud, J.-P. Rewrite systems. In *Handbook of Theoretical Computer Science* (1990), J. van Leeuwen, Ed., vol. B: Formal Models and Semantics, Elsevier, pp. 243–320.

[13] Fischer, C. CSP-OZ: A combination of Object-Z and CSP. Tech. Rep. TRCF-97-2, Universität Oldenburg, 1997.

[14] Fokkink, W. J. The *tyft/tyxt* format reduces to tree rules. In *Proceedings 2nd International Symposium on Theoretical Aspects of Computer Science (TACS'94), Sendai, Japan* (1994), vol. 789 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 440–453.

[15] Galloway, A. *Integrated Formal Methods*. PhD thesis, University of Teeside, 1996.

[16] George, C. W., and Xia, Y. An Operational Semantics for Timed RAISE. In *FM'99 — Formal Methods* (1999), J. M. Wing, J. Woodcock, and J. Davies, Eds., FME, Springer–Verlag, pp. 1008–1027.

[17] Gray, J. Notes on database operating systems. In *Operating Systems – An Advanced Course*, R. Bayer et al., Eds., vol. 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978, pp. 393–481.

[18] Grieskamp, W., Heiseland, M., and Dörr, H. Specifying embedded systems with Statecharts and Z: An agenda for cyclic software components. In *Fundamental Approaches to Software Engineering: First International Conference, FASE'98, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March/April 1998* (1998), E. Astesiano, Ed., vol. 1382 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 88–106.

[19] Groote, J. F. Transistion systems specification with negative premises. Tech. rep., CWI, Amsterdam, 1990. An extended abstract appeared in J. C. M. Baeten and J. W. Klop, editors, Proceedings CONCUR 90, Amsterdam, Lecture Notes in Computer Science no. 458, pp. 332–314, Springer-Verlag, 1990.

[20] Groote, J. F., and Vaandrager, F. W. Structured operational semantics and bisimulation as a congruence. *Information & Computation 100*, 2 (1992), 202–260.

[21] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (1987), 231–274.

[22] Harel, D. On visual formalisms. *Communications of the ACM 33*, 5 (514–530 1988).

[23] Harel, D., and Gery, E. Executable object modeling with Statecharts. *IEEE Computer 30*, 7 (1997), 31–42.

[24] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., and Trakhtenbrot, M. B. STATEMATE: A working environment for the development of complex reactive systems. *Software Engineering 16*, 4 (1990), 403–414.

[25] Harel, D., and Marelly, R. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[26] Haxthausen, A. Some approaches for integration of specification techniques (invited extended abstract), 2000.

[27] Haxthausen, A., and Xia, Y. Linking DC together with TRSL. In *Proceedings of 2nd International Conference on Integrated Formal Methods (IFM'2000), Schloss Dagstuhl, Germany, November 2000* (2000), no. 1945 in Lecture Notes in Computer Science, Springer-Verlag, pp. 25–44.

[28] Hennessy, M., and Regan, T. A process algebra for timed systems. *Information and Computation 117* (1995), 221–239.

[29] Hoare, C. A. R., and He, J. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[30] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[31] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.

[32] Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process*. Addison-Wesley, 1999.

[33] Jensen, K. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use, Volume 1 Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[34] Jouannaud, J.-P. Rewrite proofs and computations. In *Proof and Computation*, H. Schwichtenberg, Ed., vol. 139 of *Computer and Systems Sciences*. Springer Verlag, 1995.

[35] Kamin, S., and Lévy, J.-J. Two generalizations of the recursive path ordering.

Unpublished manuscript, 1980.

[36] Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata Studies*, C. Shannon and I. McCarthy, Eds. Princeton University Press, 1956, pp. 3–41.

[37] Klop, J. W. Term rewriting systems. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds., vol. 2: Background: Computational Structures. Oxford University Press, 1992, ch. 1, pp. 1–116.

[38] Klose, J., and Wittke, H. An automata based interpretation of Live Sequence Charts. In *TACAS 2001* (2001), T. Margaria and W. Yi, Eds., LNCS 2031, Springer-Verlag, pp. 512–527.

[39] Kristensen, L. M., Christensen, S., and Jensen, K. The practitioner's guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer 2*, 2 (1998), 98–132.

[40] Ladkin, P. B., and Leue, S. Analysis of Message Sequence Charts. Tech. Rep. IAM 92-013, Institute for Informatics and Applied Mathematics, University of Berne, Bern, Switzerland, 1992.

[41] Leue, S. *Methods and Semantics for Telecommunications Systems Engineering.* PhD thesis, Philosophisch-naturwissenschaftlichen Fakultät, University of Berne, Bern, Switzerland, 1995.

[42] Lüttgen, G., van der Beeck, M., and Cleaveland, R. Statecharts via process algebra. Tech. Rep. ICASE Report No. 99-42, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia, USA, 1999.

[43] Mauw, S., and Reniers, M. A. An algebraic semantics of basic Message Sequence Charts. *The Computer Journal 37*, 4 (1994), 269–277.

[44] Park, D. M. R. Concurrency and automata on infinite sequences. In *5th GI Conference* (1981), P. Deussen, Ed., vol. 104 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 167–183.

[45] Petri, C. A. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[46] Plotkin, G. D. Structural operational semantics. Lecture notes, Aarhus University, 1981. DAIMI FN-19. Reprinted 1991.

[47] Pnueli, A., and Shalev, M. What is a step: on the semantics of Statecharts. In *Theoretical Aspects of Computer Software (TACS'91)*, T. Ito and M. A, Eds., vol. 526 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991, pp. 244–264. Sendai, Japan.

[48] Pressman, R. S. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 1997.

[49] RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall Int., 1992.

[50] RAISE Method Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall Int., 1995.

[51] Reggio, G., and Repetto, L. Casl-Chart: A combination of Statecharts and of the algebraic specification language Casl. Tech. Rep. DISI-TR-00-2, DISI, Università di Genova, 2000.

[52] Reisig, W. *A Primer in Petri Net Design.* Springer-Verlag, 1992.

[53] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[54] Sharp, R. *Principles of Protocol Design*, second ed. IMM-DTU, 2001.

[55] Sommerville, I. *Software Engineering*, sixth ed. Addison-Wesley, 2001.

[56] Verhoef, C. A congruence theorem for structured operational semantics with pred-

icates and negative premises. *Nordic Journal of Computing 2*, 2 (1995), 274–302.

[57] Weber, M. Combining Statecharts and Z for the design of safety-critical control systems. In *FME 96: Industrial Benefit and Advances in Formal Methods* (1996), M. Gaudel and J. Woodcock, Eds., vol. 1051 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 307–326.

[58] Woodcock, J. C. P., and Hughes, A. Unifying theories of parallel programming. In *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China* (October 21–25 2002), C. George and H. Miao, Eds., vol. 2495 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 24–37.

[59] Zhou, C., Hoare, C. A. R., and Ravn, A. P. A calculus of durations. *Information Processing Letters 40*, 5 (1991), 269–276.