Constructive Tool Design for Formal Languages

from semantics to executing models

L.J. van Bokhoven

Constructive Tool Design for Formal Languages
from semantics to executing models

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 28 november 2002 om 16.00 uur

door

Leonardus Jakobus van Bokhoven
geboren te Eindhoven

Dit proefschrift is goedgekeurd door de promotoren:

prof.ir. M.P.J. Stevens
en
prof.dr.ir. J.E. Rooda

Copromotor:

dr.ir. J.P.M. Voeten

Print: Universiteitsdrukkerij Eindhoven
Cover design: L.J. van Bokhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Bokhoven, Leonardus J. van

Constructive tool design for formal languages : from semantics to executing models / by Leonardus J. van Bokhoven. – Eindhoven : Technische Universiteit Eindhoven, 2002.
Proefschrift. – ISBN 90-386-1960-X
NUR 965
Trefw.: discrete simulatie / procesalgebra / software-ontwikkeling ; prototypes / stochastische processen / programmeertalen ; real-time / CAD.
Subject headings: discrete event simulation / process algebra / software prototyping / probabilistic automata / real-time systems / high level synthesis.

To my loving parents and brother.

Contents

List of Figures and Tables

Acknowledgements

In 1998, I was appointed as PhD student at the Technische Universiteit Eindhoven, starting the four-year research project that led to this doctoral dissertation. For this opportunity I would like to thank prof. Stevens, who, holding the chair of the Information and Communication Systems (ICS) department, became my supervisor.

In the past few years I have had the privilege of being coached by Jeroen Voeten. His conviviality and encouraging style of working have constituted an eminent incentive to complete this project.

I already miss the pleasant, sometimes absurd, but mostly interesting discussions on sundry topics with my fellow room-mates: Marc Geilen (thanks for proofreading), Frank van Wijk and Bart Theelen. The latter, just like Mark Verhappen, produced models of unprecedented monstrosity — ostensibly only designed to expose problems in early versions of my execution engine... Well, actually they were quite invaluable beta-testers, whom I am absolutely grateful to — juncta juvant.

My other colleagues (and this is an inconclusive list) include Piet van der Putten, who, with Jeroen Voeten, laid the basis for this research, Ad Verschueren who builds amazing processor-driven contraptions, Ad Chamboné, present from the very beginning, Max Bonsel, for a helping hand, and Jos van Itegem, our system manager. Special thanks go to Rian van Gaalen, our group secretary. Although she always stays at the background, her presence can hardly go unnoticed. It is she who takes care of improving the quality of life for the other ICS members by decorating the department on festive occasions, providing a listening ear, and preparing tea and coffee for guests.

The critical remarks of the members of the committee, Mario Stevens, Koos Rooda, Jeroen Voeten, Martin Rem, Thijs Krol, Jos Baeten, Jef van Meerbergen, Henk Corporaal, René Segers and Ton Engbersen, have been valuable and allowed me to view my own work from different perspectives. I thank my new colleagues at Magma Design Automation for their patience whilst my completing this dissertation.

Working in the ICS group has not only extended my knowledge on the academic level, but has also offered an outlook over the world — even beyond the vista of Eindhoven and its surroundings. Of the people I have met, I would especially like to thank Lucia Lobont and Daniel Domilescu for their hospitality.

Last but not least, I am greatly indebted to my parents and brother — their unqualified support absolutely meant a great deal.

Summary

Embedded, distributed, real-time, electronic systems are becoming more and more dominant in our lives. Hidden in cars, televisions, mp3-players, mobile phones and other appliances, these hardware/software systems influence our daily activities. Their design can be a huge effort and has to be carried out by engineers in a limited amount of time. Computer-aided modelling and design automation shorten the design cycle of these systems enabling companies to deliver their products sooner than their competitors.

The design process is divided into different levels of abstraction, starting with a vague product idea (abstract) and ending up with a concrete description ready for implementation. Recently, research has started to focus on the system level, being a promising new area at which the product design could start.

This dissertation develops a constructive approach to building tools for system-level design/description/modelling/specification languages, and shows the applicability of this method to the system-level language POOSL (Parallel Object-Oriented Specification Language). The formal semantics of this language is redefined and partly redeveloped, adding probabilistic features, real-time, inheritance, concurrency within processes, dynamic ports and atomic (indivisible) expressions, making the language suitable for performance analysis/modelling. The semantics is two-layered, using a probabilistic denotational semantics for stating the meaning of POOSL's data layer, and using a probabilistic structural operational semantics for the process layer and architecture layer.

The constructive approach has yielded the system-level simulation tool *rotalumis*, capable of executing large industrial designs, which has been demonstrated by two successful case studies — an ATM-packet switch (in conjunction with IBM Research at Zürich) and a packet routing switch for the Internet (in association with Alcatel/Bell at Antwerp). The more generally applicable optimisations of the execution engine (rotalumis) and the decisions taken in its design are discussed in full detail.

Prototyping, where the system-level model functions as a part of the prototype implementation of the designed product, is supported by rotalumis-rt, a real-time variant of the execution engine. The viability of prototyping is shown by a case study of a learning infrared remote control, partially realised in hardware and completed with a system-level model.

Keywords formal languages / formal specification / modelling languages / system-level design / embedded systems / real-time systems / performance analysis / discrete event simulation / probabilistic process algebra / design automation / prototyping / simulation tool.

Samenvatting

Ingebedde, gedistribueerde, real-time, elektronische systemen nemen in ons leven een steeds belangrijkere plaats in. Deze verborgen hardware-/softwaresystemen in auto's, televisies, mp3-spelers, mobiele telefoons en andere apparaten beïnvloeden onze dagelijkse bezigheden. Het ontwerpen van deze systemen kan een flinke inspanning vereisen van ingenieurs, die hun taak binnen beperkte tijd moeten hebben afgerond. Modellering met behulp van computers en ontwerpautomatisering versnellen de ontwerpcyclus van deze systemen en stellen bedrijven in staat om produkten eerder dan hun concurrenten op de markt te brengen.

Het ontwerpproces kan worden onderverdeeld in verschillende niveaus van abstractie, beginnend bij een vaag omschreven produktidee (abstract) en eindigend bij een concrete beschrijving die de implementatie van het produkt mogelijk maakt. Recentelijk begint onderzoek zich te richten op het systeemniveau, een nieuw en veelbelovend vertrekpunt voor produktontwerp.

Dit proefschrift ontwikkelt een constructieve aanpak voor het bouwen van gereedschappen voor systeemniveau ontwerp-/beschrijvings-/modellerings- en specificatietalen, en laat de praktische toepasbaarheid van deze methode zien voor de taal POOSL (Parallel Object-Oriented Specification Language). De formele semantiek van deze taal wordt opnieuw gedefinieerd en uitgebreid met probabilistische eigenschappen, real-time, inheritance, parallellisme binnen processen, dynamische poorten en atomaire (ondeelbare) expressies, zodat de taal geschikt wordt voor prestatieanalyse/ prestatiemodellering. De semantiek bestaat uit twee lagen: een probabilistische denotationele semantiek om de betekenis van POOSL's datalaag te beschrijven en een structurele operationele semantiek voor de proces- en architectuurlaag.

De constructieve benadering heeft het systeemniveau simulatiegereedschap *rotalumis* opgeleverd, dat in staat is om grote industriële ontwerpen te executeren, hetgeen gedemonstreerd is door twee succesvolle casussen — een ATM-packet switch (in samenwerking met IBM Research te Zürich) en een packet routing switch voor het internet (samen met Alcatel/Bell te Antwerpen). De algemener toepasbare optimalisaties voor de executiemachine (rotalumis) en de daarbij genomen beslissingen worden gedetailleerd beschreven.

Rotalumis-rt, de real-time variant van de executiemachine ondersteunt *prototyping*, waarbij het systeemniveau model deel uitmaakt van het prototype van het te ontwerpen produkt. De uitvoerbaarheid van prototyping wordt aangetoond met een casus van een lerende infrarood afstandbediening, deels gerealiseerd in hardware en gecompleteerd met een systeemniveau model.

Trefwoorden: formele talen / formele specificatie / modelleringstalen / ontwerpen op systeemniveau / ingebedde systemen / real-time systemen / prestatieanalyse / discrete simulatie / probabilistische procesalgebra / CAD / prototyping / simulatiegereedschap.

Chapter 1

Introduction

The design of embedded electronic systems can be an intricate task. To appreciate the difficulties designers have to face, we will make a short anatomical study of an ordinary device: a mobile telephone. Figure 1.1 shows a dissected telephone, revealing its inner parts. Several elements are encountered inside the protective housing; next to the familiar things like a liquid crystal display, keyboard, microphone, loudspeaker and a battery, is a part that is unknown to most people — a printed circuit board with electronic chips mounted on either side. The chips on this board implement several subsystems. To give an idea of the functionality implemented by these chips, we consider what happens during an incoming telephone call.

An incoming call is relayed by a base station, and received by the antenna of the telephone. The antenna is connected to a radio frequency (RF) section, which analyses incoming signals and retrieves the (encrypted) information stream they carry. An error-correction system tries to remove the errors from the bitstream, after which a decryption unit unravels the original information stream. Besides digitised speech, the stream also contains control packets, for instance telling the caller's telephone number. The digitised speech is sent to a digital-to-analog converter before it is made audible on the loudspeaker.

For answering the caller, sound picked up by the microphone is sent to an analog-to-digital converter and then to a digital signal processor. Finally, the digitised speech is encrypted and, via the RF-section, transmitted by the antenna to the base station that relays the signal to the caller.

Most of these systems are coordinated and controlled by a microprocessor that executes a multitasking operating system. The software applications that run on this operating system perform various chores. Next to handling the interface to the user of the telephone (display and keyboard), they also handle the communication protocol with the base station, power management and so forth.

A mobile telephone is an example of a contraption containing an embedded, distributed, real-time system made up of software and hardware. Other examples are (digital) photo cameras, television sets, dvd/cd/mp3-players, printers and —although you might not expect it— cars; in fact, a modern car contains several microprocessors communicating with and possibly reacting to each other: an engine management system, anti-lock braking system (ABS), airbag system, climate control system and a cruise control system. We are surrounded by embedded systems!

The design of embedded systems can be very complicated. To reason about them and make well-founded decisions along their design trajectory, models are created that describe the system's behaviour, which is not necessarily transformational, but can also be dependent on time, next to the stimuli from the environment. Describing the behaviour of these *reactive systems* that continuously interact with their environment, requires a language that can capture timing aspects, parallelism, communication, selection, conditional execution, interrupts, watchdogs and more. These features are missing in traditional languages such as C, Fortran, Smalltalk and Pascal, but can be found in so-called design/description/modelling/specification languages like POOSL, ESTEREL, SystemC and VHDL.

Only a few of these languages are really fit for describing the complex systems being designed today. To better understand a system, there should not only be a rigourous specification of it on paper, but also an adequate executable model. Tools for executable models enable simulation, testing, verification, performance analysis and so

Figure 1.1: Disassembled mobile telephone.

on, without the need to actually build (hardware) prototypes. Unfortunately, current tools cannot keep up with the increase in complexity of designs, and, consequently, engineers are having difficulties to finish their products in time. Tools may fail because they cannot render their results within reasonable time or because the size or complexity of the model exceeds their limits.

In order to reduce the design cycle, engineers can start with a simplified model and gradually fill in more specific details. This technique of concentrating on important aspects only and disposing of other issues is called *abstraction*. At the *system level*, an engineer reasons about a system by concentrating on the interacting concurrent subsystems it is composed of. The subsystems themselves may also be a compound or *cluster* of subsystems. Indivisible subsystems are called *processes*. A system-level model not only describes how these processes behave (regardless of whether that behaviour will be realised in hardware or in software), but also how they are grouped and interconnected by communication channels — the hierarchical architecture.

An example of such a system-level model for the mobile telephone described above, might contain a process that models the decryption of an incoming data stream. After unravelling the original information stream, the packets containing speech can be sent to another process that simulates converting them into analog signals and making them audible, while control packets might go to a process that represents the controlling microprocessor. Notice that if the actual content of the information stream is not used, it can be left out without any repercussions. For example, an abstraction of the information stream can be a sequence of messages simply stating the presence of a control packet or a data packet.

Abstraction can be advantageous for several reasons. The decrease of detail simplifies the model, taking less time to execute. A compact and concise model is also easier to understand and reason about with colleagues in the design team. Abstracting should, however, be performed with great care: if relevant aspects are left out, the model may yield incorrect results and is then called *inadequate*.

Only models that adequately reflect the actual system enable the engineer to examine various implementations and to make well-founded design decisions. The explored implementation alternatives are part of the *design space*. Not all implementations fulfill the design requirements equally well — some may be cheap to build, others may be hard to manufacture or even infeasible and still others leave the possibility open of additional features. The designer is expected to find, within the limited time-to-market, a profitable design that meets its requirements. It is here that the engineer can benefit from the system-level approach: because only the key elements of a design are taken into account, it is easier (compared to more detailed models) to reason about the model. Decisions taken at this high level of abstraction can have a significant impact on the realisation's quality, performance, cost, et cetera. After deciding upon the global behaviour and architecture of the design, the engineer can concentrate on the subsystems one by one, describe them in more detail at a lower level of abstraction, while using the results of the system-level model as a reference. Design automation tools finally help the designer to actually build software and hardware to realise the system.

1.1 Objectives

The heart of the problem of designing embedded, distributed, real-time software/ hardware systems lies in their increasing complexity, which can hardly be handled by current design tools. A solution that could alleviate this problem, at least for a while, is the use of more abstract models that reason about a system in terms of the subsystems it is composed of. Such a system-level approach requires at least the following:

- a system-level design method;
- a system-level specification and modelling language;
- simulation, verification and performance analysis tools capable of handling large industrial models.

The research performed by the Information and Communication Systems group (ICS) has already resulted in the system-level design method SHE (Software/Hardware Engineering) and the system-level specification and modelling language POOSL (Parallel Object-Oriented Specification Language) [41]. The accompanying techniques and tools for verification [18] and performance analysis [54, 53, 52, 55] are being developed.

The objective of this thesis is to provide a constructive approach to building tools for system-level design languages based on their formal semantics. The applicability of this approach should be demonstrated by a simulation tool for POOSL, capable of dealing with large industrial problems, yet the approach should not be restricted to this particular tool and language. Features missing in POOSL, but required to support performance analysis or easier modelling should be included and supplied with a formal semantics. Proving and verifying desired properties of this semantics is beyond the scope of this thesis and will be future research — emphasis lies on the method of implementing the language. The simulator (or *execution engine*) should enable design space explorations, performance studies, validation and prototyping. For prototyping, the system-level model acts as a virtual prototype of (part of) the design, interacting with actual processes in the real world. This requires the model time to be synchronised with the real time and the generating of and reacting to external events; this extension should be described by a formal semantics too. The execution engine should also be able to function as a slave-component in other tools.

1.2 Related Research

Other approaches to the execution of formal specifications exist for languages such as LOTOS (Language of Temporal Ordering Specifications) [13], CCS (Calculus of Communicating Systems) [35], μCRL (based on Common Representation Language)[20], SDL (Specification and Description Language) [49], χ [8] or TyCO (Typed Concurrent Objects) [29].

[14, 12] discusses work on executing CCS/LOTOS specifications. The simulator tool Hippo executes LOTOS specifications. The implementation of that tool has been derived by a stepwise transformation of the LOTOS inference rules. It employs a 'menu' function to compute the available transitions given the current execution state, and a 'next' function to move to a subsequent state for a given available transition

and execution state. [12] also suggests to increase the efficiency of computing state representations by factoring out the dynamic part from the part that is unchanged during a transition.

LOTOS uses abstract data types (ADTs), requiring a nonconstructive implementation of the data part. The need for constructive approaches towards implementing data for process calculi is for instance presented in [9], giving a redefinition of Milner's value-passing calculus [35] to prevent that for each value a different message should be constructed.

In [30] and [29] a virtual machine is discussed for the execution of a polymorphic, strongly typed, concurrent and object-oriented programming language based on a process calculus called TyCO (a variant of the π-calculus [36]). Specifications are compiled to an intermediate assembly language representation, which is then translated to byte-code that the virtual machine can run. The language does not have provisions to handle real-time aspects.

Besides the research into simulation presented in this dissertation, the Information and Communication Systems group is working on the following related subjects in the field of system-level design. The foundation for the Software/Hardware Engineering (SHE) methodology and POOSL are laid in [41]. [18] contributes a number of formal techniques (both exhaustive and non-exhaustive) in the area of verification of distributed real-time systems. Further, techniques are introduced for automatic verification of requirements formalised in linear temporal logic. In [55] a theory for performance analysis is developed.

The tools based on this work on simulation, verification and performance modelling, are applied to investigate techniques for design space exploration in the field of digital video/multimedia systems [56], and have been used for industrial case studies in the area of telecommunication systems [47, 48] (nonconfidential excerpts are [46, 45]) for gaining insight in performance modelling, that is, techniques for building adequate models for performance analysis.

Part of the work for this thesis is supported as part of the STW Progress Research Project EES5202 on "Modelling and Performance Analysis of Telecommunication Systems".

1.3 Thesis Overview

The organisation of this thesis is as follows. After this introduction, Chapter 2 discusses the mathematical preliminaries required for understanding the semantics of POOSL. The three conceptual layers of POOSL are discussed in Chapters 3, 4 and 5, each containing two parts: a specification of the semantics of the language primitives for that layer, followed by a technique for implementing those primitives. The resulting implementation will execute the language constructs while respecting their meaning as specified by the formal semantics. These techniques have been applied to build the basic framework for the execution engine *rotalumis* and its precursors. To make the engine capable of supporting large industrial designs, it requires optimisation. Chapter 6 analyses the source of potential problems in large-scale models and presents countermeasures to reduce their effect (if possible). The optimised engine has been put to the test in several case studies from the industry, which have shown that the engine is capable of successfully handling very large models. Chapter 7 discusses an extension for prototyping, accompanied by a case study of a learning infrared remote control that has partly been realised in hardware. Chapter 8 concludes this thesis and provides directions for future work. Additional content is gathered in Appendix A, defining the formal semantics of POOSL, and in Appendix B, listing the concrete syntax of POOSL.

1.4 Reading Instructions

The part at the back of this thesis contains a glossary of symbols that may help you in finding the definition of symbols and their meaning; the index is added for quick retrieval of text related to a specific subject. People only interested in techniques for constructing execution engines, can concentrate on Sections 2.3, 2.4, 3.3, 4.3, 5.3 and Chapter 6. If you want to model in POOSL, you can download SHESim or rotalumis and use Appendix B to learn the syntax of POOSL. When a simulation run takes unaccountably much time for a large-scale model, it might help to read Chapter 6 for avoiding known bottlenecks — the expressive power of POOSL introduces enough freedom to model a problem in various ways. Persons with an affinity to mathematics are invited to read Chapter 2, Sections 3.2, 4.2, 5.2 and Appendix A. For the diehards among us: continue and do not stop reading before you have finished the final page!

It is recommended to read the subsequent sections. The first one gives a global overview of POOSL, stating the key elements of this system-level specification and modelling language. It is succeeded by an explanation of the global structure of rotalumis, which may serve as a guide to the upcoming implementation details in Sections 3.3, 4.3, 5.3 and Chapter 6.

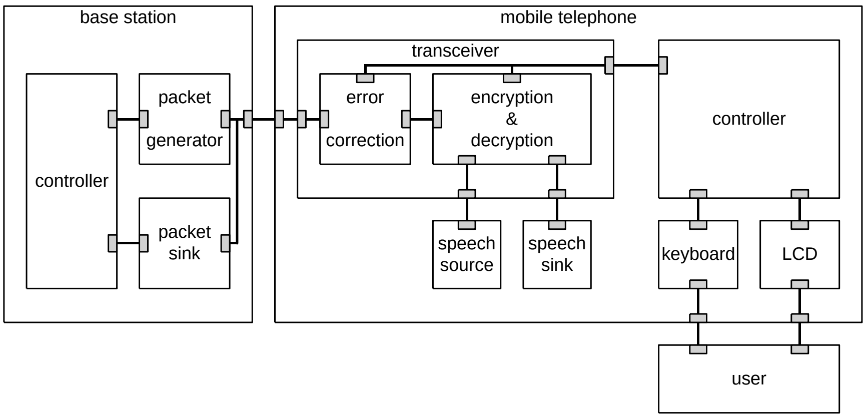


Figure 1.2: System-level model of a mobile telephone.

1.5 System-Level Modelling Language POOSL

The formal specification language POOSL is part of the system-level design method SHE and was introduced and formally defined in [41]. Real-time concepts for POOSL have been studied in [17]. POOSL is used for stating a rigourous specification of a system; this specification can directly function as an executable model of the design. This thesis will add inheritance, immediate data, dynamic port passing (as in the π-calculus [36]), parallelism and probabilistic features to the language and develop a new semantics for the full language, which unifies these concepts.

Because POOSL plays a key role in this dissertation, we will give an overview of the language, following its three conceptual layers, describing architecture, processes and data respectively.

1.5.1 Architecture Layer

Reactive systems can often be decomposed into several concurrent, communicating subsystems that may also be decomposable. The architecture layer of POOSL offers:

- *processes* to represent subsystems that are considered as single entities;
- *clusters* to denote a group composed of processes and clusters;
- *channels* to symbolise the communication paths between processes and clusters.

Clusters introduce hierarchy in the model. The architecture of a model is static; it does not change over time. It may reflect the actual functional blocks of a system as it will be realised, or it follows a different decomposition — one that matches a conceptual design.

Without going into all the details, Figure 1.2 gives an example of a system-level model of the mobile telephone discussed in the introduction of this chapter. On the highest hierarchical level are two clusters (base station and mobile telephone) and a process (user). The base-station cluster contains[1] three processes that emulate the peer telephone trying to communicate with the mobile telephone over an error-distorted channel. The base-station cluster is an example of a conceptual model of the actual base station: it merely functions as a communication partner that enables testing the communication protocol between the mobile telephone and the base station. The gray rectangles depict communication ports. Connected to these ports are channels, shown by the thicker lines.

This system-level model is closed (self-containing): both the system under development (mobile telephone) and its environment (base station and user) are specified in a single model. The model may enable measuring the average amount of information flowing from base-station to mobile telephone and back (a performance measurement), or it can be used for testing the communication protocol (validation) or even proving that the protocol is error-free (verification). Perhaps the model is part of an architectural study, enabling an early evaluation of possible system decompositions of the telephone. What information can be extracted from the model, depends mostly on how the behaviour of its processes is described at the process layer.

[1]In SHESim [16], the contents of a cluster are hidden, but the user can open a viewer to inspect the cluster's internal structure. This thesis displays the contents of clusters to reduce the figure count.

Table 1.3: Part of the process class for encryption and decryption.

```
 1  process class         EncryptionAndDecryption
 2  port interface        ec, spsrc, spsnk, ctrlr
 3  message interface     ec?packet(Packet), ec!packet(Packet), spsrc?packet(Packet),
 4                        spsnk!packet(Packet), ctrlr!packet(Packet), ctrlr?packet(Packet)
 5  instance variables    p: Packet,  TimeToDecrypt: Real
 6  initial method call   DoEncryptDecrypt()()
 7  instance methods
 8
 9  DoEncryptDecrypt()()
10    TimeToDecrypt := 2.5;         // Set constant value.
11    par
12        HandleOutgoingPackets()()
13    and
14        AwaitIncomingPackets()()
15    rap.
16
17  AwaitIncomingPackets()()
18    ec?packet(p);                 // Receive packet from error correction unit.
19    delay TimeToDecrypt;          // Model the time for decrypting the packet.
20    if p ContainsSpeech then
21        spsnk!packet(p)           // Packet contains speech.
22    else
23        ctrlr!packet(p)           // Packet contains control information.
24    fi;
25    AwaitIncomingPackets()().     // Handle next packet.
```

1.5.2 Process Layer

The process layer describes the behaviour of processes, their communication interface and their message interface. As with other object-oriented languages, process objects (or just processes) are instances of their classes. It is assumed that the reader is familiar with the concepts of object-oriented languages.

A process can be equipped with ports for communication. Connected to these ports are channels over which messages can travel. POOSL offers synchronous pair-wise message passing, based upon the indivisible handshake principle for communication from CCS [35]. Processes use messages to exchange information, such as the result of computations (discussed in the next section).

The behaviour of a process can be described with the statements shown on page 58. There are *language primitives* to define interrupts, watchdogs (**abort** S_1 **with** S_2), communication (send $p!m$ and receive $p?m$), real-time aspects (**delay** t), parallelism (**par** S_1 **and** S_2 **rap**) and more. Any combination and nesting of these statements is allowed for specifying how processes should behave. Methods allow the behaviour to be decomposed into well-ordered individual pieces of code — comparable with procedures or functions in imperative languages. This thesis introduces implementation inheritance, allowing a process to inherit methods and variables of its superclass.

As an example of how behaviour is described, we show (part of) the specification of the encryption and decryption process class in Table 1.3. After a process has been instantiated from this class, its behaviour is started by initially calling method `DoEncryptDecrypt()()`. The body of this method (lines 10–15) specifies two concurrent tasks — `HandleOutgoingPackets()()` for handling outgoing packets containing speech or control information and `AwaitIncomingPackets()()` for awaiting and handling incoming packets from the error correction process (Figure 1.2).

Table 1.4: Part of the data class `Packet`.

```
data class          Packet
extends             Object
instance variables  ContentsType: String
instance methods

SetTypeToSpeech()
  ContentsType := "Speech";
  self.                        // Return this object as the result.

SetTypeToControlInfo()
  ContentsType := "Control";
  self.                        // Return this object as the result.

ContainsSpeech()
  ContentsType == "Speech".
```

We will look at the latter method in more detail. After receiving packet `p` in line 18 (from port `ec` that is assumed to be connected to the error correction unit) and delaying to model the decryption time, the packet is either sent to the speech sink (line 21) or to the controller (line 23), depending on the type of the packet. The tail-recursive call in line 25 allows the next packet to be handled.

Processes have separate data spaces, accessible through their instance variables. At line 5, two variables are declared, `p` and `TimeToDecrypt`, so the data space of this process comprises the data objects accessible through these two variables. Processes do not share their data space with other processes, and data can only be exchanged explicitly as a parameter of a message. Line 21 gives an example of this: a copy of the object to which `p` refers is sent as a parameter along message `packet` to the speech sink (assumed to be connected via a channel to port `spsnk`).

1.5.3 Data Layer

The object-oriented data can be used for complex calculations. A data class describes what variables and methods its instances (data objects) have. Inheritance allows objects of a class to reuse and extend methods and variables of another class (called the super class). A data object can encapsulate its data —which it can access through its instance variables— and defines a clear interface of data methods through which other objects may obtain information regarding its state or modify that state. The body of a data method is a sequence of expressions, of which the last executed expression serves as the result of the entire method. Table 1.4 shows an example of the `Packet` data class, and some of its methods.

The process class discussed in Section 1.5.2 uses method `ContainsSpeech` in line 20 to determine what kind of contents object `p` of class `Packet` is carrying.

1.6 Performance Analysis

One of the reasons for building system-level models, is the ability to analyse the performance of a system early in the design trajectory. These so-called performance analyses enable designers to explore a set of possible architectures and choose one that meets

Table 1.5: Probabilistic model of the communication-packet generator.

```
 1  process class         PacketGenerator
 2  port interface        mt, ctrlr
 3  message interface     mt!packet(Packet), ctrlr?start, ctrlr!stop
 4  instance variables    DistrA, DistrB: Distribution
 5  initial method call   HandleControl()()
 6  instance methods
 7
 8  HandleControl()()
 9    DstrA := new(Bernoulli) withParameter(0.995);
10    DstrB := new(Bernoulli) withParameter(0.1);
11    abort
12        ctrlr?start;
13        SendCorrectPacket()()
14    with
15        ctrlr?stop;
16    HandleControl()().
17
18  SendCorrectPacket()()
19    mt!packet(new(Packet) SetTypeToSpeech);
20    if DistrA sample then SendCorrectPacket()() else SendFaultyPacket()() fi.
21
22  SendFaultyPacket()()
23    mt!packet(new(Packet) SetTypeToGarbled);
24    if DistrB sample then SendFaultyPacket()() else SendCorrectPacket()() fi.
```

the design requirements with respect to performance. For instance, consider a set of processors using a set of buses for communication. Depending on the (distributed) application run on the processors and the chosen architecture, there might from time to time exist serious communication bottlenecks. Analysing the performance of an abstract model of this system should reveal these problems and help designers gain a clear insight into the origin of the bottlenecks, which might ultimately lead to a modified architecture solving the problem — for instance by adding or merging buses.

To support the powerful tool of performance analysis, the language in which the abstract model is built should be given a probabilistic meaning. This thesis redefines the semantics of POOSL, providing language primitives a probabilistic meaning. We will examine the telephone call emulator process in Figure 1.2 to discuss the issues involved in parameter estimation.

The behaviour of the communication-packet generator is specified in Table 1.5. Before any packet is sent, the generator will wait for a start signal from the controller (line 12). After that, it starts producing packets, some of which are simulated to be garbled by transmission. The controller can stop the sending at any moment with a stop message (line 15). In this example, `SendCorrectPacket()()` transmits a packet containing speech (line 19), draws a sample from the Bernoulli distribution `DistrA` and decides if it should send another correct packet or a garbled one (line 20). There is a 99.5% probability that `DistrA sample` returns *true*, causing the next packet to be correct. If the coin flipping returns *false*, method `SendFaultyPacket()()` is called instead. This method has similar behaviour, but uses a different distribution to draw samples from.

To find out what the average percentage of garbled packets is, we will construct the underlying Markov chain for this model. To simplify the chain a bit, we consider executing a statement from method `SendCorrectPacket()()` as being in state A and executing a statement from method `SendFaultyPacket()()` as residing in state B. Without going into too much detail, we assert that Figure 1.6 depicts the resulting Markov chain. If the model is currently in state A, then with probability 0.995 it will stay in that state, and with probability 0.005 it will go to state B. This choice corresponds to the `if`-statement in line 20, and probability 0.995 originates from a method of the random number generator implicitly called through the expression `DistrA sample`.

The equilibrium probabilities of the resulting Markov chain can be computed —in this case analytically— revealing that the program resides in state B with probability $\frac{1}{181}$. So, on average about 0.55% of the packets is garbled.

In general, the underlying Markov chain is much larger, even so complex that it cannot be solved analytically within reasonable time. In [55] a theory for performance estimation is developed, which, based on a Markov chain generated on-the-fly from the simulated model, estimates the long-run average rewards for that chain. The same theory allows performance metrics to be specified in terms of *reward formulae*. Three kinds of rewards are defined: boolean, real and temporal. In each state, rewards can be modified and they can be used to obtain performance metrics. For example, if we define two real-valued rewards r_A, r_B and have reward $r_A = 1$ and $r_B = 0$ each time the simulator encounters state A and do the opposite ($r_A = 0$, $r_B = 1$) in state B, the ratio $\sum_{i=1}^{n} r_A(S_i) \Big/ \sum_{i=1}^{n} r_B(S_i)$, where S_i denotes the i-th encountered stochastic state, will become 180 in the long run (that is, for $n \rightarrow \infty$).

There are two ways to support this kind of performance analysis. One approach uses a special language for specifying reward formulae and telling in which states the rewards should be modified. The other technique uses POOSL to state these matters in the model itself. Such models are called *reflexive*, as they reason about themselves. Whichever method is taken, the reward theory requires each statement of the underlying language (in our case POOSL) to have a well-defined probabilistic meaning. Previously, POOSL lacked such a probabilistic semantics, but this thesis will give an initial definition. Future research is needed to ensure that this definition has the desired properties such as time-determinism, time-additivity and so on.

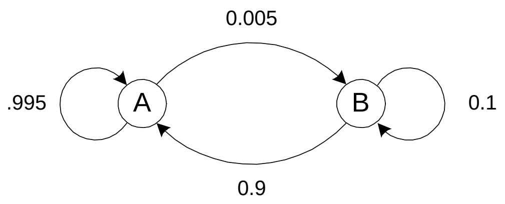


Figure 1.6: Abstracted underlying Markov chain of the packet generator.

1.7 Rotalumis

Besides the development of a new formal semantics that adds inheritance, dynamic port passing as in the π-calculus, real-time and probabilistic aspects to POOSL, an execution engine was developed: rotalumis. Important issues that should be and have been taken into account for the construction of rotalumis are:

- generic implementation concepts applicable to languages with a denotational semantics or structural operation semantics (like POOSL);
- extensibility; the engine should be extensible to serve as a verification tool, performance analysis tool and so on;
- industrial-strength; the execution engine should be able to cope with large industrial designs — this requires optimisation techniques to reduce scalability effects and increase the simulation speed.

The various concepts that have been used in the construction of rotalumis are discussed per semantical layer —in reversed order— in Chapters 3 (data), 4 (processes) and 5 (architecture), following the stages of incrementally implementing the execution engine. Each chapter first defines the formal semantics of that layer before discussing its implementation.

Before doing so, Figure 1.7 gives a global view on the components rotalumis comprises. The compiled POOSL specification on the right provides the class definitions of data objects and processes, as well as the model's architecture. During simulation this part remains unchanged and is therefore also referred to as the static part of the model. The dynamic part on the left consists of "alive" objects holding the current state of the model, which originate from the static definitions but may change while the simulation progresses. Both the dynamic and the static section have been partitioned into three pieces, corresponding to the semantical layers. On the data layer, a virtual machine (VM) evaluates expressions, assisted by a garbage collector (GC) that reclaims unreachable data objects. The process layer uses execution trees (ET)

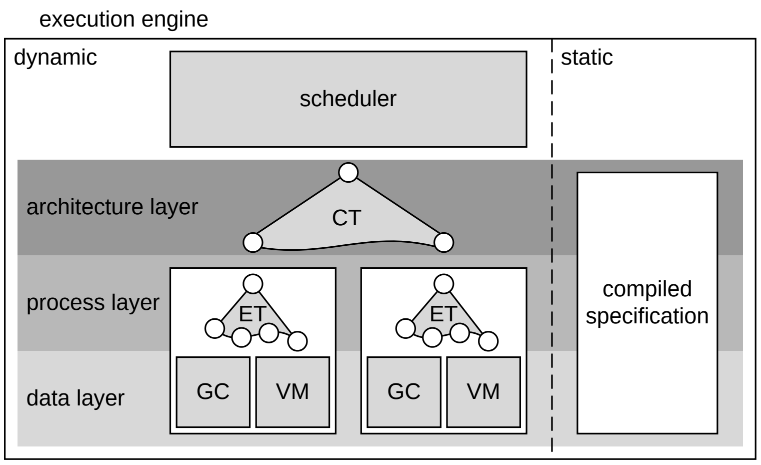


Figure 1.7: Overview of the execution engine rotalumis while running a POOSL simulation.

to represent the current state of a process' behaviour statement. Channel trees (CT) at the architecture layer combine communication requests from the execution trees before they are sent to the scheduler, along with other requests from the execution trees. The scheduler resolves nondeterminism and decides which statement is eligible for execution.

After the interlude of mathematical preliminaries, Chapters 3, 4 and 5 will explain these issues in more detail.

Chapter 2

Mathematical Preliminaries

2.1 General

2.1.1 Functions

The notation presented here is based on the representation used in [44, 38]. Let A and B be sets. A *partial function* from A to B is any correspondence f that maps every $a \in D \subseteq A$ to some unique $f(a) \in B$. For any $a \in A \setminus D$, f is undefined, which is denoted by $f(a) = \underline{\text{undef}}$. The set of all partial functions from A to B is denoted by $A \hookrightarrow B$. D is called the *domain* of f, denoted by $\mathrm{Dom}(f)$. If $\mathrm{Dom}(f) = A$, f is called a *(total) function* from A to B. We will write $A \rightarrow B$ to denote the set of all functions from A to B. Alternative notations for $f(a)$ are $f.a$ and f_a.

Let $f \in V \hookrightarrow W$, let $v, v' \in V$ and let $w \in W$. The variant notation $f\{w/v\}$ is used to build a modified or extended function based on f and is defined by:

$$f\{w/v\}(v') = \begin{cases} f(v') & \text{if } v' \neq v \text{ and } v' \in \mathrm{Dom}(f) \\ w & \text{if } v' = v \\ \underline{\text{undef}} & \text{if } v' \neq v \text{ and } v' \notin \mathrm{Dom}(f). \end{cases}$$

With the restriction operator $\restriction$, the domain of a function can be limited. Let $f \in V \hookrightarrow W$, let $v \in V$ and let $Z \subseteq V$. Then $f{\restriction}Z \in V \hookrightarrow W$ is defined by:

$$f{\restriction}Z\,(v) = \begin{cases} f(v) & \text{if } v \in \mathrm{Dom}(f) \cap Z \\ \underline{\text{undef}} & \text{otherwise.} \end{cases}$$

2.1.2 Natural numbers

We let $\mathbb{N}$ denote the set of natural numbers $\{1, 2, 3, \ldots\}$ and use $\mathbb{N}_0$ to denote $\mathbb{N} \cup \{0\}$.

2.1.3 Product of sets

The (Cartesian) product $A_1 \times A_2$ of sets A_1 and A_2 is defined as:

$$A_1 \times A_2 = \Big\{ f \in \{1,2\} \longrightarrow (A_1 \cup A_2) \;\Big|\; f(1) \in A_1 \text{ and } f(2) \in A_2 \Big\}.$$

If I is any set and A_i is a set for every $i \in I$, then

$$\prod_{i \in I} A_i = \Big\{ f \in I \longrightarrow \bigcup_{i \in I} A_i \;\Big|\; f(i) \in A_i \text{ for all } i \in I \Big\}.$$

Notice that $I \longrightarrow A = \prod_{i \in I} A$.

We will write A^n to denote the n-fold Cartesian product $\overbrace{A \times A \times \cdots \times A}^{n}$, which is defined as $\prod_{i \in \{1,\ldots,n\}} A$ or alternatively as $\{1, \ldots, n\} \longrightarrow A$. We let A^ω denote $\mathbb{N} \longrightarrow A$.

Let A be a set and let $a_1, \ldots, a_n \in A$. Further, let a be a function in A^n that maps each $i \in \{1, \ldots, n\}$ to a_i. We call a an *ordered n-tuple* and denote it by $(a_1, \ldots, a_n)$. As an alternative to the i-th argument of a we may speak of a_i as the i-th *component* of the n-tuple. Since the n-tuple $(a_1, \ldots, a_n)$ just denotes the function $a \in \{1, \ldots, n\} \longrightarrow A$, we have by definition $a(i) = a_i$.

Let A be a set and let $a_i \in A$ for all $i \in \mathbb{N}$. Furthermore, let a be a function in A^ω that maps each $i \in \mathbb{N}$ to a_i. We call a a *sequence* and denote it by $(a_1, a_2, \ldots)$.

The powerset of A is denoted by $\mathbf{2}^A$.

2.1.4 Binary relations

A binary relation on a set A is any subset $\mathrm{R} \subseteq A \times A$ and we write $a \,\mathrm{R}\, a'$ to specify that $(a, a') \in \mathrm{R}$. A binary relation R on A is:

- *reflexive* if $a \,\mathrm{R}\, a$ for all $a \in A$;
- *transitive* if $a \,\mathrm{R}\, a'$ and $a' \,\mathrm{R}\, a''$ imply $a \,\mathrm{R}\, a''$ for all $a, a', a'' \in A$;
- *anti-symmetric* if $a \,\mathrm{R}\, a'$ and $a' \,\mathrm{R}\, a$ imply $a = a'$ for all $a, a' \in A$.

A *partial order* is a reflexive, transitive and anti-symmetric relation.

2.2 Denotational Semantics

2.2.1 Chain-complete partially ordered sets and fixed-point theory

Some of the following definitions are based on the work of R.D. Tennent described in [44]. The theory presented here extends the definition of continuous functions to support chain-complete partially ordered sets of n-tuples and will be used to develop the denotational semantics of POOSL's data layer.

Definition 1 (poset) *A* partially ordered set (poset) *is a pair* $(D, \sqsubseteq)$ *where* D *is a set and* $\sqsubseteq$ *is a partial order on* D.

Definition 2 (least element) *Let $(D, \sqsubseteq)$ be a poset. An element $d \in D$ is called a* least element *of D iff $d \sqsubseteq d'$ for all $d' \in D$.*

Theorem 3 *If a poset has a least element, that element is unique.*

Proof of Theorem 3 Let $(D, \sqsubseteq)$ be a poset, and assume that D has two least elements d_1 and d_2. Since d_1 is a least element we have $d_1 \sqsubseteq d_2$. Since d_2 is a least element we also have $d_2 \sqsubseteq d_1$. But then $d_1 = d_2$ by anti-symmetry of $\sqsubseteq$. □

Such a unique element is denoted by $\perp$ (pronounced "bottom").

Definition 4 (least upper bound) *Let $(D, \sqsubseteq)$ be a poset and let Y be a subset of D. Then $d \in D$ is an* upper bound *of Y iff $d' \sqsubseteq d$ for all $d' \in Y$. An upper bound $d \in D$ of Y is a* least upper bound *of Y iff $d \sqsubseteq d'$ for all upper bounds $d' \in D$ of Y.*

Theorem 5 *Let $(D, \sqsubseteq)$ be a poset, let Y be a subset of D and let $d \in D$ be a least upper bound of Y. Then d is unique.*

Proof of Theorem 5 Assume that Y has two least upper bounds $d_1, d_2 \in D$. Since d_1 is a least upper bound we have $d_1 \sqsubseteq d_2$. But d_2 is also a least upper bound, so $d_2 \sqsubseteq d_1$. Anti-symmetry of $\sqsubseteq$ gives $d_1 = d_2$. □

If the unique least upper bound of a set Y exists, it is denoted by $\bigsqcup Y$.

Definition 6 (chain) *Let $(D, \sqsubseteq)$ be a poset. A sequence $d \in D^\omega$ is an* ascending ω-chain *(hereafter abbreviated to* chain*) in D, if for all $i, j \in \mathbb{N}$, $d_i \sqsubseteq d_j$ when $i \leq j$. If d is a chain, we write $\bigsqcup d$ or $\bigsqcup_i d_i$ to denote $\bigsqcup\{d_i \mid i \in \mathbb{N}\}$ if this least upper bound exists; likewise, if $\{d_i \mid i \in I\}$ is a set, $\bigsqcup_{i \in I} d_i$ denotes $\bigsqcup\{d_i \mid i \in I\}$.*

Using the previous definitions and theorems, we now define *chain-complete partially ordered sets*, which play an important role in the denotational semantics discussed in Section 3.2.3.

Definition 7 (ccpo) *A poset $(D, \sqsubseteq)$ is called a* chain-complete partially ordered set (ccpo) *iff each chain $d \in D^\omega$ has a least upper bound $\bigsqcup d$.*

Definition 8 *Let $d \in D^\omega$ and let $f \in D \rightharpoonup D'$. We will write $f(d)$ to denote the sequence $\big(f(d_1), f(d_2), f(d_3), \ldots\big)$.*

The following two properties are prerequisite to the functions describing the denotational semantics.

Definition 9 (monotony) *Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be poset's and let $f \in D \rightarrow D'$. Then f is* monotone *iff $d_1 \sqsubseteq d_2$ implies $f(d_1) \sqsubseteq' f(d_2)$ for all $d_1, d_2 \in D$.*

Definition 10 (continuity) *Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be ccpo's and let $f \in D \rightarrow D'$. Then f is* ω-continuous *(hereafter* continuous*) iff $f(\bigsqcup d) = \bigsqcup f(d)$ for every chain $d \in D^\omega$.*

Theorem 11 *Let $f \in D \rightarrow D'$ be a continuous function defined on ccpo's $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$. Then f is also monotone.*

Proof of Theorem 11 Let $d_1, d_2 \in D$ and let $d \in D^\omega$ be a chain of the form $(d_1, d_2, d_2, d_2, \ldots)$. Since f is continuous, $\bigsqcup f(d) = f\big(\bigsqcup d\big)$, which equals $f(d_2)$ because $\bigsqcup d = d_2$. Since $f(d_1) \sqsubseteq' \bigsqcup f(d)$ we have $f(d_1) \sqsubseteq' f(d_2)$ and hence f is monotone. $\square$

Definition 12 (fixed point) *Let $(D, \sqsubseteq)$ be a poset and let $\mathcal{F} \in D \rightarrow D$. We call $d \in D$ a* fixed point *of $\mathcal{F}$ if $\mathcal{F}(d) = d$. If $\mathcal{F}$ has a least fixed point, it is denoted by* $\mathrm{FIX}\,\mathcal{F}$.

The following key theorem shows how the least fixed point of a function can be computed.

Theorem 13 (Knaster-Tarski) *Let $(D, \sqsubseteq)$ be a ccpo with least element*[1] $\perp$ *and let $\mathcal{F} \in D \rightarrow D$ be continuous. Then $\mathrm{FIX}\,\mathcal{F} = \bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)$.*

Proof of Theorem 13 See also [43] and [38].

1. $\bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)$ is well-defined.

   $\mathcal{F}^0(\perp) = \perp$ and $\perp \sqsubseteq d$ for all $d \in D$. By induction on i and monotony of $\mathcal{F}$ we find that $\mathcal{F}^i(\perp) \sqsubseteq \mathcal{F}^i(d)$ for all $d \in D$. Hence $\mathcal{F}^i(\perp) \sqsubseteq \mathcal{F}^j(\perp)$ for $i \leq j$ and therefore $\big(\mathcal{F}^0(\perp), \mathcal{F}^1(\perp), \ldots\big)$ is a chain in D^ω. Because $(D, \sqsubseteq)$ is a ccpo, $\bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)$ exists.

2. $\bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)$ is a fixed point of $\mathcal{F}$, that is $\mathcal{F}\Big(\bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)\Big) = \bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)$.

   By continuity of $\mathcal{F}$ we get that $\mathcal{F}\Big(\bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)\Big)$ equals $\bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}\big(\mathcal{F}^i(\perp)\big)$. Since $\bigsqcup(\perp, d_1, d_2, \ldots) = \bigsqcup d$ for all chains $d \in D^\omega$, we can rewrite this to $\bigsqcup\big(\{\mathcal{F}^{i+1}(\perp) \mid i \in \mathbb{N}_0\} \cup \{\perp\}\big)$ which is $\bigsqcup_{i \in \mathbb{N}_0} \mathcal{F}^i(\perp)$.

[1] In contrast to usual ccpo definitions in literature, the existence of a least element is not automatically fulfilled by Definition 7 — hence the additional requirement.

3. $\bigsqcup_{i\in\mathbb{N}_0} \mathcal{F}^i(\bot)$ is the least fixed point of $\mathcal{F}$.

   Let d be a fixed point of $\mathcal{F}$. Since $\bot \sqsubseteq d$, monotony of $\mathcal{F}$ gives $\mathcal{F}^i(\bot) \sqsubseteq \mathcal{F}^i(d)$ for all $i \in \mathbb{N}_0$. Because d is a fixed point, we have $\mathcal{F}^i(\bot) \sqsubseteq d$ and hence d is an upper bound of the chain $\{\mathcal{F}^i(\bot) \mid i \in \mathbb{N}_0\}$. But $\bigsqcup_{i\in\mathbb{N}_0} \mathcal{F}^i(\bot)$ is the least upper bound of this chain, therefore $\bigsqcup_{i\in\mathbb{N}_0} \mathcal{F}^i(\bot) \sqsubseteq d$ and thus $\bigsqcup_{i\in\mathbb{N}_0} \mathcal{F}^i(\bot)$ is the least fixed point of $\mathcal{F}$. $\square$

This theorem is crucial in the definition of the denotational semantics in Section 3.2.3 and will be applied to the semantic function (that is defined there). Because the theorem can only be applied to continuous functions and ccpo's, the semantics will have to be built in such a way that these requirements are fulfilled. The following definitions and propositions help in the construction of the semantics and proving some of its characteristics.

Definition 14 *Let $f \in \big(D' \rightarrow D\big)^\omega$ and let $d \in D'$. We will write $f(d)$ to denote the sequence $\big(f_1(d), f_2(d), f_3(d), \ldots\big)$.*

The context should unveil whether $f(d)$ denotes the application of a sequence of functions to d (Definition 14) or the function application to the elements of a sequence d (Definition 8).

The following theorem is useful for hierarchically constructing ccpo's.

Theorem 15 *Let D' be a set and let $(D, \sqsubseteq)$ be a ccpo. For $f, g \in D' \rightarrow D$, define $f \sqsubseteq' g$ iff $f(d) \sqsubseteq g(d)$ for all $d \in D'$. $(D' \rightarrow D, \sqsubseteq')$ is a ccpo.*

Proof of Theorem 15

1. reflexiveness: because $f(d) \sqsubseteq f(d)$ for all $d \in D'$, we also have $f \sqsubseteq' f$.
2. transitivity: if $f \sqsubseteq' f'$ and $f' \sqsubseteq' f''$ then, for all $d \in D'$, we have $f(d) \sqsubseteq f'(d)$ and $f'(d) \sqsubseteq f''(d)$, which implies $f(d) \sqsubseteq f''(d)$; hence $f \sqsubseteq' f''$.
3. anti-symmetry: if $f \sqsubseteq' f'$ and $f' \sqsubseteq' f$ then $f(d) \sqsubseteq f'(d)$ and $f'(d) \sqsubseteq f(d)$ for all $d \in D'$, and therefore $f = f'$.
4. chain-completeness: Let $f \in \big(D' \rightarrow D\big)^\omega$ be a chain. Define $f_\omega(d) = \bigsqcup f(d)$ for all $d \in D'$. By definition of f, we can conclude that for all $d \in D'$, $f(d)$ is a chain in D^ω. Since $(D, \sqsubseteq)$ is a ccpo, this chain has a least upper bound $\bigsqcup f(d)$, and therefore $f_\omega(d)$ is well-defined for all $d \in D'$.

   We claim that $\bigsqcup f = f_\omega$. Since f is a chain, we have $f_1 \sqsubseteq' f_2 \sqsubseteq' f_3 \sqsubseteq' \cdots$. f_ω is an upper bound of f because $f_i \sqsubseteq' f_\omega$ for all $i \in \mathbb{N}$. To see that f_ω is the *least* upper bound of f, let f' be any upper bound of f. Then $f_i(d) \sqsubseteq f'(d)$ for all $i \in \mathbb{N}$ and any $d \in D'$. But $f_\omega(d)$ was defined as being the least upper bound of $f(d)$ for all $d \in D'$, so $f_\omega(d) \sqsubseteq f'(d)$ for all $d \in D'$ and therefore $f_\omega \sqsubseteq' f'$. $\square$

The theory is now augmented by ccpo's of n-tuples and ccpo's of sequences.

Definition 16 *Let $(D, \sqsubseteq)$ be a ccpo and let $d, d' \in D^n$ be n-tuples for $n \in \mathbb{N}$. Then $d \sqsubseteq^n d'$ iff $d_i \sqsubseteq d'_i$ for all $i \in \{1, \ldots, n\}$. Analogously for $d, d' \in D^\omega$ we define that $d \sqsubseteq^\omega d'$ iff $d_i \sqsubseteq d'_i$ for all $i \in \mathbb{N}$.*

Corollary 17 *Let $(D, \sqsubseteq)$ be a ccpo and let $n \in \mathbb{N}$. $(D^n, \sqsubseteq^n)$ and $(D^\omega, \sqsubseteq^\omega)$ are ccpo's.*

Proof of Corollary 17 Let $d, d' \in D^n$. From Theorem 15 with $D' = \{1, \ldots, n\}$, $f = d$, $g = d'$ and $\sqsubseteq' = \sqsubseteq^n$, we get that $(D^n, \sqsubseteq^n)$ is a ccpo since (Definition 16) $d \sqsubseteq^n d'$ iff $d_i \sqsubseteq d'_i$ for all $i \in \{1, \ldots, n\}$. Equivalently, but with $D' = \mathbb{N}$ and $\sqsubseteq' = \sqsubseteq^\omega$, we obtain that $(D^\omega, \sqsubseteq^\omega)$ is a ccpo since (Definition 16) $d \sqsubseteq^\omega d'$ iff $d_i \sqsubseteq d'_i$ for all $i \in \mathbb{N}$. □

We shall assume from here on that if $(D, \sqsubseteq)$ is a ccpo, it is implicitly understood that domains D^n and D^ω are equipped with relations $\sqsubseteq^n$ and $\sqsubseteq^\omega$ respectively, to form the ccpo's $(D^n, \sqsubseteq^n)$ and $(D^\omega, \sqsubseteq^\omega)$.

Double chains [44] have an interesting property that simplifies the continuity proofs in Chapter 4.

Theorem 18 (double chains) *Let $(D, \sqsubseteq)$ be a ccpo and let $d \in (D^\omega)^\omega$ be a chain of chains. $\bigsqcup_i \bigsqcup_j d_{ij}$, $\bigsqcup_j \bigsqcup_i d_{ij}$ and $\bigsqcup_k d_{kk}$ are well-defined and equivalent.*

Proof of Theorem 18 Define $d_{i\omega} = \bigsqcup_j d_{ij}$ and $d_{\omega j} = \bigsqcup_i d_{ij}$ and $d_\omega = \bigsqcup_i d_i$. Since d is a chain, we have $d_i \sqsubseteq_\omega d_{i+1}$ for all $i \in \mathbb{N}$, so $d_{i\omega} \sqsubseteq d_{(i+1)\omega}$ and therefore $\bigsqcup_i \bigsqcup_j d_{ij}$ is well-defined. Since d_ω is a chain, we have $d_{\omega j} \sqsubseteq d_{\omega(j+1)}$ for all $j \in \mathbb{N}$ and therefore $\bigsqcup_j \bigsqcup_i d_{ij}$ is well-defined. Finally, for all $k \in \mathbb{N}$, we have $d_{kk} \sqsubseteq d_{k(k+1)} \sqsubseteq d_{(k+1)(k+1)}$, so $\bigsqcup_k d_{kk}$ is also well-defined.

Because d_i is a chain, $d_{ij} \sqsubseteq d_{i\omega}$ for all $i, j \in \mathbb{N}$ and so $\bigsqcup_i d_{ij} \sqsubseteq \bigsqcup_i \bigsqcup_j d_{ij}$ for all $j \in \mathbb{N}$, and therefore $\bigsqcup_j \bigsqcup_i d_{ij} \sqsubseteq \bigsqcup_i \bigsqcup_j d_{ij}$. Since d is a chain, we have $d_i \sqsubseteq_\omega d_\omega$ for all $i \in \mathbb{N}$, so $d_{ij} \sqsubseteq d_{\omega j}$ for all $i, j \in \mathbb{N}$ and thus $\bigsqcup_j d_{ij} \sqsubseteq \bigsqcup_j \bigsqcup_i d_{ij}$ for all $i \in \mathbb{N}$, and therefore $\bigsqcup_i \bigsqcup_j d_{ij} \sqsubseteq \bigsqcup_j \bigsqcup_i d_{ij}$. By anti-symmetry of $\sqsubseteq$ we get that $\bigsqcup_i \bigsqcup_j d_{ij} = \bigsqcup_j \bigsqcup_i d_{ij}$.

For every $i \in \mathbb{N}$, we have $d_{ij} \sqsubseteq d_{kk}$ for $k = \max(i, j)$, so $\bigsqcup_j d_{ij} \sqsubseteq \bigsqcup_k d_{kk}$ for each $i \in \mathbb{N}$ and therefore $\bigsqcup_i \bigsqcup_j d_{ij} \sqsubseteq \bigsqcup_k d_{kk}$. Furthermore, for every $k \in \mathbb{N}$, we have $d_{kk} \sqsubseteq d_{k\omega} \sqsubseteq \bigsqcup_i d_{i\omega}$; so $\bigsqcup_k d_{kk} \sqsubseteq \bigsqcup_i \bigsqcup_j d_{ij}$. By anti-symmetry of $\sqsubseteq$, we conclude that $\bigsqcup_i \bigsqcup_j d_{ij} = \bigsqcup_k d_{kk}$. □

2.3 Syntax

Many functions, sets and syntactic categories in this thesis are defined by describing their elements in extended Backus-Naur form (EBNF, see Appendix B.1). An example of such an inductive definition goes as follows.

The set of boolean expressions, **B**, ranged over by b is defined by:

$$\begin{array}{rcl} b & = & \mathit{true} \\ & \mid & \mathit{false} \\ & \mid & \neg\, b \\ & \mid & b_1 \wedge b_2 . \end{array}$$

(Notice that the full stop after $b_1 \wedge b_2$ is not part of that construct but merely ends the sentence.)

Meta-variable or *typical element* b ranges over *syntactic category* **B**. In general, if φ is a meta-variable, so are its decorated versions $\varphi_1, \varphi', \ldots$, and they range over the same set. The *basis constructs true* and *false* do not depend on b, whereas *composite constructs* $\neg\, b$ and $b_1 \wedge b_2$ have constituents that are boolean expressions themselves. Example elements of **B** are: $\neg\, \mathit{true} \wedge \neg\, \mathit{false}$ and $\mathit{false} \wedge \mathit{true} \wedge \mathit{true}$. The latter string representation could have been derived in two ways, namely by $(\mathit{false} \wedge \mathit{true}) \wedge \mathit{true}$ or by $\mathit{false} \wedge (\mathit{true} \wedge \mathit{true})$. If any string can be parsed in a unique way, the corresponding syntax is called *concrete* (instead of *abstract*). Constructs can be decorated to resolve ambiguities — we will use parentheses instead. If the precedence (binding power) of constructs is specified, parentheses can be omitted; for example, if $\wedge$ binds stronger than $\neg$, then $\neg\,(\mathit{true} \wedge (\neg\, \mathit{false}))$ can simply be written as $\neg\, \mathit{true} \wedge \neg\, \mathit{false}$.

Let **C** be defined by:

$$\begin{array}{rcl} c & = & \mathit{true} \\ & \mid & \mathit{false} \\ & \mid & \neg\, c \\ & \mid & c_1 \wedge c_2 \\ & \mid & c_1 = c_2 . \end{array}$$

This definition implicitly specifies c to be a typical element of **C**. Because any construct $b \in \mathbf{B}$ can also be built with constructs from **C**, we have $\mathbf{B} \subseteq \mathbf{C}$ and, obviously, $b \in \mathbf{C}$.

2.4 Structural Operational Semantics

This thesis uses timed probabilistic labelled transition systems [42] to describe compositionally the behaviour of reactive systems modelled in POOSL.

A *timed probabilistic labelled transition system* $(\mathit{Conf}, A, \xrightarrow[A]{}, T, \xrightarrow[T^+]{})$ consists of a set of *configurations* Conf, a set of *actions* A with a set of probabilistic *action-transition relations* $\cdot \xrightarrow[A]{\cdot} \cdot \subseteq \mathit{Conf} \times A \times \mathcal{P}(\mathit{Conf})$ where $\mathcal{P}(\mathit{Conf})$ denotes the set of substochastic probability functions, and a time domain T with a set of deterministic *time-transition relations* $\cdot \xrightarrow[T^+]{\cdot} \cdot \subseteq \mathit{Conf} \times T^+ \times \mathit{Conf}$. An example of such a transition system has been depicted on the left in Figure 2.1 by a graph — nodes

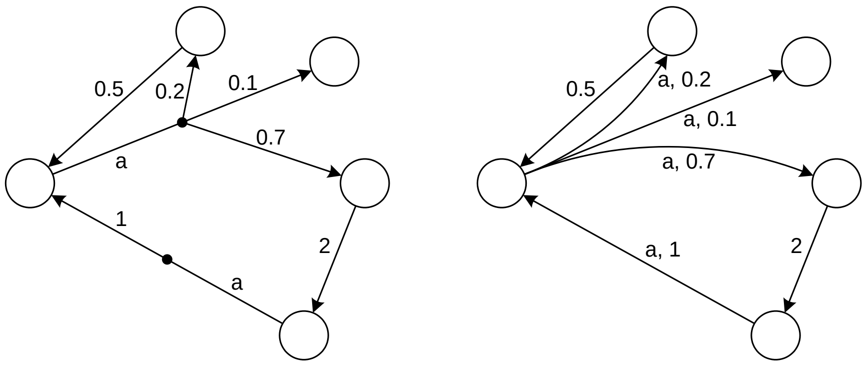


Figure 2.1: Graphic representations of a timed probabilistic labelled transition system.

represent configurations and edges symbolise transitions. Only action transitions are probabilistic; hence it is clear that 0.5 and 2 are labels of time transitions. An action transition is drawn as a line annotated by the action, followed by a fan-out of arrows that represents the corresponding probability function. This is just a shorthand notation that bundles transitions with a common action (see right graph), giving a clearer notion of nondeterminism.

A configuration $(S, I) \in \mathit{Conf}$ contains a *statement* S that is to be executed in the context of *information* I. The statement describes the (future) behaviour of the system, while the information captures the system's internal state. Assuming that $a \in A$ and $t \in T^+$, we use $(S, I) \xrightarrow[A]{a} \boldsymbol{\pi}$ to denote[2] that the system, currently in configuration (S, I), can do an action (perform a *transition* or *execution step*) and yield a distribution $\boldsymbol{\pi} \in \mathcal{P}(\mathit{Conf})$ over configurations — $\boldsymbol{\pi}.(S', I')$ gives the probability that the system will reside in configuration (S', I') after the action. Further, $(S, I) \xrightarrow{a} (S', I')$ denotes that there is a $\boldsymbol{\pi}$ such that $(S, I) \xrightarrow{a} \boldsymbol{\pi}$ with $\boldsymbol{\pi}.(S', I') > 0$. Actions are *instantaneous* (take no time). Timing aspects of a system are modelled by time transitions: $\xrightarrow{t}$ denotes the passage of t units of time. Arbitrary transitions (action or time) are denoted by $\longrightarrow$.

The sequence of configurations $(S_1, I_1), (S_2, I_2), (S_3, I_3), \ldots$ is called a *trace*, if it can be obtained by repeatedly performing an execution step: $(S_i, I_i) \longrightarrow (S_{i+1}, I_{i+1})$ for $i \geq 1$.

An important part of a transition system is its set of transition relations. The transition relations will be defined by a Plotkin-style [40] structural operational semantics in GSOS format [5], as a set of inference rules of the following form:

$$\frac{(S_1, I_1) \xrightarrow{a_1} \boldsymbol{\pi}_1,\ (S_2, I_2) \xrightarrow{a_2} \boldsymbol{\pi}_2,\ \ldots,\ (S_n, I_n) \xrightarrow{a_n} \boldsymbol{\pi}_n}{(S, I) \xrightarrow{a} \boldsymbol{\pi}}\ \textsc{name} \quad \text{if } \mathit{condition}$$

where $\boldsymbol{\pi} = f(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_n)$

[2]We also allow the notation $(S, I) \xrightarrow{a} \boldsymbol{\pi}$ (dropping A) if the kind of the transition can be determined by the element above the transition or is clear by the context.

From the *premises* $(S_1, I_1) \xrightarrow{a_1} \boldsymbol{\pi}_1$ through $(S_n, I_n) \xrightarrow{a_n} \boldsymbol{\pi}_n$, the rule NAME deduces the *conclusion* that configuration (S, I) can perform transition a and yield probability function $\boldsymbol{\pi}$ (composed of $\boldsymbol{\pi}_1$ through $\boldsymbol{\pi}_n$, denoted by function f) if the *condition* is met. A rule without premises is called an *axiom.* In a *structural* operational semantics, the rules follow the structure of statements: axioms define the behaviour of base constructs while the other inference rules describe the behaviour of composite constructs in terms of their constituents.

Concurrency POOSL models can contain several distributed concurrent processes. Each process is defined by its own labelled transition system. To combine these transition systems to a single one, *interleaving concurrency* is used. This abstraction of concurrency allows transitions of the processes to be combined in arbitrary order, allowing only pairs of synchronising statements (communication primitives) to execute at the same moment. The internal parallelism[3] of processes is described as a *pure* interleaving because concurrent activities within a process cannot synchronise with each other.

Nondeterminism Sometimes, the set of inference rules can be used to derive several transitions for a given configuration. For instance, if $(S, I) \xrightarrow{a} (S', I')$ and $(S, I) \xrightarrow{a'} (S'', I'')$, the next configuration can either be (S', I') or (S'', I''), depending on which action (a or a') is chosen. This choice is left unspecified and is made *nondeterministically.*

Time domain The time domain is chosen to be a commutative monoid $(T, +, 0)$ with the following properties [37], for all $t, t' \in T$:

- $t + t' = t \Longleftrightarrow t' = 0$;
- the relation $\leq$ defined as $t \leq t' \Longleftrightarrow \exists_{t'' \in T} t + t'' = t'$ is a total order.

It follows from these properties that:

- 0 is the least element of T;
- for any $t, t' \in T$ with $t \leq t'$, the element t'' such that $t + t'' = t'$ is unique; it is denoted by $t' - t$.

The time domain T is called *discrete* if every instant in time has a successor, so $\forall_{t \in T} \exists_{t' \in T} (t < t' \wedge \forall_{t'' \in T} t < t'' \Longrightarrow t' \leq t'')$, and is called *dense* if in between two instants there is always another moment in time ($\forall_{t,t' \in T} t < t' \Longrightarrow \exists_{t'' \in T} t < t'' < t'$). We let T^+ denote $T \setminus \{0\}$.

Examples of proper time domains are $\mathbb{N}_0$ (discrete), $\mathbb{Q}^{\geq 0}$ (dense) and $\mathbb{R}^{\geq 0}$ (dense) with their ordinary 0-elements, ordering and addition.

[3]In this thesis the term parallelism is considered synonymous with concurrency.

Chapter 3

POOSL Data Layer

3.1 Overview

This chapter presents the abstract syntax of POOSL's data layer, strongly based upon the data layer previously introduced with an operational semantics in [41][1]. Using the extended theory of ccpo's discussed in Section 2.2.1, a new formal semantics is developed here that:

- adds inheritance to the data layer of POOSL;
- gives language constructs a probabilistic meaning, which is required for performance modelling;
- is a *denotational* semantics, guiding a compositional implementation for executing the language.

The different style of semantics is further attractive for describing the syntactic constructs' probabilistic features that will be added to the POOSL language.

After Section 3.2 has specified both the syntax and semantics of the data layer of POOSL, Section 3.3 develops a compositional execution framework for the data layer, using the denotational semantics as a prescription. The resulting *virtual machine* incorporates a garbage collector to free the modeler from the tedious task of deleting objects that are no longer used.

[1] We stress that it is beyond the scope of this thesis to show equivalence between the semantics defined here and (parts of) the one defined in [41]. Such a proof might be part of future research.

3.2 Specification

3.2.1 Abstract Syntax

This section defines the abstract syntax of POOSL's data layer. For this discussion, we introduce the following syntactic categories and corresponding meta-variables:

CName	class names	C
MName	method names	m, n
IVar	instance variables	x, y
LVar	local variables	u, w, z
$Var = IVar \cup LVar$	variables	v.

Throughout this chapter, C and m are assumed to range over data class names and data method names respectively.

A POOSL model is an element of the set of system specifications containing a list, *CDList*, of class definitions from the set of class definitions, *ClassDef*, which is partially defined by:

$$\begin{array}{lcll} CD & = & \textbf{data class} & C \\ & [& \textbf{extends} & C_{super}\ ] \\ & & \textbf{instance variables} & x_1 \cdots x_n \\ & & \textbf{instance methods} & MD_1 \cdots MD_k. \end{array}$$

In Chapters 4 and 5, *ClassDef* will be augmented by process class definitions and cluster class definitions. A data-class definition specifies the name of the class, the name of its superclass, the set of variables known to each instance of the class and a set of method definitions. Only class *Object*, present in every model and from which every other class inherits[2], has no superclass.

POOSL defines five primitive data classes: *Boolean*, *Integer*, *Real*, *String* and *Nil*. Instances of these classes are called *primitive data objects*. Class *Nil* has only a single instance, the primitive object *nil*. Other data classes will be called nonprimitive; their instances are *nonprimitive data objects*.

The instance methods of a data class are elements of *MethDef*, the set of method definitions:

$$\begin{array}{lcl} MD & = & m(u_1, \ldots, u_n) \\ & & |z_1 \cdots z_m| \\ & & E \\ & | & m(u_1, \ldots, u_n) \\ & & |z_1 \cdots z_m| \\ & & \textbf{primitive}. \end{array}$$

Each data method definition starts with a name m, a list of formal parameters $u_1 \ldots u_n$ and a list of local variables $z_1 \ldots z_m$. The body of the method is an expression, or is provided by axioms in the semantics of POOSL, denoted by the keyword **primitive**.

[2]This thesis describes *implementation inheritance* [41], not inheritance for typing.

Methods of the latter type are called *primitive methods*[3] and have behaviour that cannot be captured by a POOSL data expression. Examples of such primitive methods are *deepcopy*, *shallowcopy*, *identity* (==) and *random*. The majority of the data method bodies is provided by expressions $E \in \mathit{Exp}$. The abstract syntax of *Exp* is as follows:

$$
\begin{array}{rcll}
E & = & x & \text{global variable} \\
 & | & u & \text{local variable} \\
 & | & \underline{\gamma} & \text{literal} \\
 & | & \mathbf{new}(C) & \text{object creation} \\
 & | & \mathbf{self} & \text{self} \\
 & | & \mathbf{currentTime} & \text{current model time} \\
 & | & E\ m(E_1, \ldots, E_n) & \text{dynamic method call} \\
 & | & E\ m_{C}(E_1, \ldots, E_n) & \text{static method call} \\
 & | & x := E & \text{assignment to global variable} \\
 & | & u := E & \text{assignment to local variable} \\
 & | & E_1; E_2 & \text{sequential composition} \\
 & | & \mathbf{if}\ E_c\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2\ \mathbf{fi} & \text{if} \\
 & | & \mathbf{while}\ E_c\ \mathbf{do}\ E\ \mathbf{od}. & \text{while}
\end{array}
$$

The first two expressions consist only of an *instance variable (global variable)* x or a *local variable* u. The value (or result) of such expressions is the object referred to by the respective variable. *Constant expressions* are denoted by $\underline{\gamma}$, which is the direct naming (textual representation) of the primitive object γ. The next expression is *object creation*; it results in a new instance of the nonprimitive data class C. The value of **self** is the data object that is currently executing this expression. The expression **currentTime** reflects the current model time. It can only be used in the context of a process object.

Next is the dynamic *method call* $E\ m(E_1, \ldots, E_n)$. First, E is evaluated to some data object β, expressions $E_1 \ldots E_n$ are evaluated from left to right, resulting in the actual parameters $\beta_1 \ldots \beta_n$. Then, the method definition of m is looked up in the class definition of β. The formal method parameters are initialised to $\beta_1 \ldots \beta_n$ and the body is executed, resulting in some object β_r that will also be the result of the entire expression $E\ m(E_1, \ldots, E_n)$.

A class inherits the methods of its superclass and can provide additional methods itself. Methods can also be redefined, *overriding* the method definition provided by the superclass. Method identifiers are decorated with the name of the superclass to obtain a unique method identifier. For instance, the definition of method m in superclass C is accessible through m_{C}. The *static* method call $E\ m_{C}(E_1, \ldots, E_n)$ allows explicitly calling method m defined in class C, irrespective of whether that method is overridden for the receiving object. The implementation will restrict the use of static method calls in such a way that only inherited methods can be accessed.

The following two expressions are *assignments* to instance variables and local variables respectively. The variable is set to the value of expression E, which is also the result

[3] Although primitive methods are allowed to have local variables $z_1, \ldots, z_m$, they cannot use them. The variables are only permitted to simplify semantic rules 12 and 13 in Appendix A.1.

of the assignment itself. *Sequential composition* is denoted by a semicolon; E_1 will be evaluated before E_2 is evaluated. The result of the sequential composite is the result of E_2.

Conditional execution of data expressions is possible with the `if`[4]. If condition E_c evaluates to primitive object *true*, E_1 will be evaluated. If the condition evaluates to primitive object *false* instead, E_2 is evaluated. The result of the entire `if` equals the result of the chosen branch. Execution blocks when E_c does not evaluate to a boolean value.

Finally, the `while` repeatedly evaluates body E as long as condition E_c equals *true*. The loop construct finishes when E_c equals *false*, and results in *nil*. Execution blocks when E_c does not evaluate to a boolean value.

3.2.2 Context conditions

Some additional requirements apply to the abstract syntax of the data classes of a system specification, but cannot be given in EBNF. These *context conditions* are informally described by:

1. All class names in *CDList* are different.
2. The inheritance graph of data classes is a tree of which the root is class *Object*.
3. All (inherited) instance variables of a class definition have different names.
4. All (inherited) methods of a class are discernable by their name[5].
5. All parameter and local variable names in a method definition are different.
6. Every variable used in a method body is either an (inherited) instance variable of the corresponding class, a method parameter, or a local variable of that method.
7. The class referred to by any `new` is contained in *CDList*.

3.2.3 Denotational Semantics

The technique of *denotational semantics* has been developed by C. Strachey and D. Scott. It describes the *effect* of executing a program by means of a semantic function that maps each syntactic construct to a mathematical object describing the effect of executing that construct. The semantic function is defined *compositionally*:

- there is a semantic clause for each basis construct;
- the semantic clause for a composite construct is described in terms of the semantic function applied to its immediate constituents.

As an overview, the rationale behind the denotational semantics is discussed before a detailed description of the semantics for POOSL's data layer is presented.

[4]Instead of writing "the if-expression", we use "`if`", typeset in teletype font.

[5]In the concrete syntax this is relaxed to a unique combination of name and parameter count.

3.2.3.1 Rationale Behind the Denotational Semantics

The specification language POOSL, presented with an operational semantics in [41], has been used to conduct performance studies [47, 48, 46, 45]. These explorations use a framework of POOSL data classes within the model that is being measured (reflexive performance analysis, [53]). This approach to performance analysis requires the modeler to explicitly construct objects for collecting statistical data and calculating performance metrics. The downside of this approach is its error-proneness and its cluttering the specification with information only useful during performance analyses. Also, when different metrics are pursued, the model may have to be adapted accordingly.

An approach more convenient to the modeler is to use a different language to express performance metrics in terms of rewards. The mathematical foundation for this reward theory has been developed in [55]. Now, the modeler simply states a reward formula specifying the desired performance metric and does not have to sully the system's specification.

Both approaches have one thing in common: they build a Markov chain (possibly on-the-fly during simulation) and reason about its properties. However, in general these performance results are void of meaning if the model (and hence the Markov chain) has no probabilistic semantics [54]. Earlier experiments with CCS-based languages in [52] show that introducing probabilistic information *at the process layer* by adding probabilities directly into its operational semantics is feasible but cumbersome and gives rise to artificial weighting factors. The modeler is forced to specify the probabilities for parallel composition and selection operators, even if these operators are used to model nondeterministic behaviour. An operational semantics also requires its transitions to be decorated, for correctly computing transition probabilities [54].

The natural place to extend the language with probabilistic information is at the data layer. The new semantics for POOSL's data layer is developed here and is denotational, providing a guideline for a compositional implementation. It replaces the operational semantics of the data layer described in [41]. The hallmark of denotational semantics is compositionality. Unfortunately, recursion complicates stating a compositional definition. POOSL offers two ways to describe recursion: by means of a `while` (*local* recursion), or by means of method calls (*global* recursion). We will first discuss local recursion to demonstrate why a compositional definition cannot be obtained straightforwardly.

Local recursion, described by expression $\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}$, is chosen to have the same effect as $\texttt{if}\ E_c\ \texttt{then}\ E\texttt{;while}\ E_c\ \texttt{do}\ E\ \texttt{od else nil fi}$, hence their meanings (denoted by $[\![\cdot]\!]$) should be identical. Intuitively, the meaning of `while` could have been defined by:

$$\begin{aligned}
&[\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!] \\
&\qquad = [\![\texttt{if}\ E_c\ \texttt{then}\ E\texttt{;while}\ E_c\ \texttt{do}\ E\ \texttt{od else nil fi}]\!] \\
&\qquad = \mathcal{Y}\big([\![E_c]\!], [\![E]\!], [\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]\big)
\end{aligned}$$

where $\mathcal{Y}$ is a function that gives the meaning of **while** E_c **do** E **od** in terms of the semantics of E_c, E and `while` — not **nil** because that is a constant. It is clear that this definition is not compositional because $\mathcal{Y}$ defines the meaning of the `while` in terms of `while` itself. To obtain a compositional definition, we introduce the functional[6] $\mathcal{Z}(g) = \mathcal{Y}(\llbracket E_c \rrbracket, \llbracket E \rrbracket, g)$. The semantics of **while** E_c **do** E **od** must be a fixed point (Definition 12) of $\mathcal{Z}$ and is defined as the *least* fixed point of this functional: $\llbracket$**while** E_c **do** E **od**$\rrbracket \triangleq \mathrm{FIX}\,\mathcal{Z}$. With this definition, $\llbracket$**while** E_c **do** E **od**$\rrbracket$ *is* compositional, because $\llbracket \cdot \rrbracket$ is only applied to the immediate constituents of **while** E_c **do** E **od** and not to the construct itself. Using a fixed point is a common approach taken in literature to obtain a compositional semantics for while-languages [38, 44, 22].

Global recursion (through method calls) introduces a similar compositionality problem that can also be solved with a fixed point. Problems with compositionality do not exist if method calls can statically be replaced by the bodies of the methods they invoke. However, since POOSL offers dynamic method binding, the called method is not known in advance and therefore recursion cannot be unrolled. To still obtain a compositional definition of $\llbracket \cdot \rrbracket$, an intermediate function $\llbracket \cdot \rrbracket_f$ is defined that gives the meaning of expressions, conditional upon n-tuple f containing the meaning of the method bodies. A functional $\mathcal{B}(f)$ will be defined (Definition 37) that gives the semantics of method bodies conditional upon an approximation of the meanings of all method bodies — this is comparable with functional $\mathcal{Z}$'s parameter g approximating the meaning of `while`. The least fixed point is taken to define the exact meaning of the method bodies, and is used to finally define the exact (unconditional) meaning of any expression $E \in \mathit{Exp}$ by choosing $\llbracket E \rrbracket \triangleq \llbracket E \rrbracket_{\mathrm{FIX}\,\mathcal{B}}$.

3.2.3.2 Typical Sets, Variables and Functions

The primitive data objects of classes *Boolean*, *String*, *Integer* and *Real* represent the boolean objects ($\mathbb{B}$), string objects ($\mathbb{S}$), integer objects ($\mathbb{I}$) and real objects ($\mathbb{R}$) respectively. The theories of these sets, together with their usual operators and relators, are assumed to be parameters of the semantics. We let γ range over the set of all primitive data objects, which is defined by the countable set $\mathit{PDObj} = \mathbb{B} \cup \mathbb{S} \cup \mathbb{I} \cup \mathbb{R} \cup \{\mathit{nil}\}$. The set of nonprimitive data objects $\mathit{NDObj} = \{\widehat{n} \mid n \in \mathbb{N}\}$ is ranged over by α. The capped integer serves as an identifier representing the actual nonprimitive data object. The entire collection of data objects is defined by $\mathit{DObj} = \mathit{NDObj} \cup \mathit{PDObj}$, with typical elements β.

The executing object determines the context in which an expression is evaluated. The currently executing object, typically denoted by δ, can either be a process object (proc) or any nonprimitive data object ever (indirectly) known to proc and is an element of $\Delta = \mathit{NDObj} \cup \{\mathit{proc}\}$.

$\Phi = \mathit{IVar} \hookrightarrow \mathit{DObj}$ with typical elements ϕ is the set of partial functions that map instance variables to data objects. $\Psi = \mathit{LVar} \hookrightarrow \mathit{DObj}$ is the set of partial functions that map local variables to data objects and is ranged over by ψ.

The set of *variables states* $\Sigma = \{\sigma \in \Delta \hookrightarrow \Phi \mid \mathit{Dom}(\sigma) \text{ is finite}\}$ is ranged over by σ. The domain of partial function σ consists of the executing process and the nonprimi-

[6] A *functional* is a mapping from a collection (class) of functions to a collection of functions.

tive data objects ever (indirectly) known through the process' instance variables. The data object referred to by instance variable x of δ is determined by $\sigma(\delta)(x)$. The set $\Lambda = \Delta \times \Psi$ is ranged over by λ. Each tuple $\lambda = (\delta, \psi)$ stores the executing object δ and the corresponding function ψ retaining the references of its local variables. The classes of nonprimitive data objects are stored in elements τ of $T = \mathit{NDObj} \hookrightarrow \mathit{CName}$. For assigning a unique object identifier to each nonprimitive data object, the function $\mathrm{MaxId}(\sigma)$ is defined, where:

$$\mathrm{MaxId}(\sigma) = \begin{cases} 0 & \text{if } \mathrm{Dom}(\sigma) = \{\mathit{proc}\} \\ \max\{n \mid \widehat{n} \in \mathrm{Dom}(\sigma)\} & \text{otherwise.} \end{cases}$$

The execution state of the model, restricted to the data layer, is stored in an element of $\mathit{State} = \Sigma \times \Lambda \times T$, typically denoted by s. The countable set $\mathit{State}' = \mathit{State} \times \mathit{DObj}$ has typical elements s' that describe the state and object resulting from the execution of an expression.

The collection of (inherited) instance variables of a data class C is given by $\mathcal{V}(C)$ where $\mathcal{V} \in \mathit{CName} \hookrightarrow \mathbf{2}^{\mathit{IVar}}$ is defined by:

$$\mathcal{V}(C) = \begin{cases} \{x_1, \ldots, x_n\} & \text{if } \mathit{CDList} \equiv \mathit{CD}_1 \cdots \mathit{CD}_i \cdots \mathit{CD}_n \\ & \text{and } \mathit{CD}_i \equiv \begin{array}{ll} \texttt{data class} & C \\ \texttt{instance variables} & x_1 \cdots x_n \\ \texttt{instance methods} & \mathit{MD}_1 \cdots \mathit{MD}_k \end{array} \\ \{x_1, \ldots, x_n\} & \\ \quad \cup\, \mathcal{V}(C_{super}) & \text{if } \mathit{CDList} \equiv \mathit{CD}_1 \cdots \mathit{CD}_i \cdots \mathit{CD}_n \\ & \text{and } \mathit{CD}_i \equiv \begin{array}{ll} \texttt{data class} & C \\ \texttt{extends} & C_{super} \\ \texttt{instance variables} & x_1 \cdots x_n \\ \texttt{instance methods} & \mathit{MD}_1 \cdots \mathit{MD}_k \end{array} \\ \varnothing & \text{otherwise,} \end{cases}$$

where $\equiv$ denotes syntactic identity. The definition will be completed in Section 4.2.3.

Function $\mathcal{M}_s \in \mathit{CName} \hookrightarrow \mathit{MName} \hookrightarrow (\mathit{MD} \cup \mathit{MD}^p)$ is a lookup function that returns the definition of method m if it is defined in class C and returns $\underline{\text{undef}}$ otherwise. For data classes, $\mathcal{M}_s$ is defined by:

$$\mathcal{M}_s.C.m \equiv \begin{cases} \mathit{MD}_j & \text{if } \mathit{CDList} \equiv \mathit{CD}_1 \cdots \mathit{CD}_i \cdots \mathit{CD}_n \\ & \text{and } \mathit{CD}_i \equiv \begin{array}{ll} \texttt{data class} & C \\ [\,\texttt{extends} & C_{super}\,] \\ \texttt{instance variables} & x_1 \cdots x_n \\ \texttt{instance methods} & \mathit{MD}_1 \cdots \mathit{MD}_j \cdots \mathit{MD}_k \end{array} \\ & \text{and } \mathit{MD}_j \equiv \begin{array}{l} m(u_1, \ldots, u_n) \\ |z_1 \cdots z_m| \\ E \mid \texttt{primitive} \end{array} \\ \underline{\text{undef}} & \text{otherwise.} \end{cases}$$

Table 3.1: Example of inheritance.

Class A:
```
1 data class A
2 extends Object
3 instance variables
4 instance methods
5
6 init()
7     nil.
```

Class B:
```
1 data class B
2 extends A
3 instance variables
4 instance methods
5
6 init()
7     self ^init.
```

Class C:
```
1 data class C
2 extends B
3 instance variables
4 instance methods
5
6 init()
7     self ^init.
```

For dynamic method binding, lookup function $\mathcal{M} \in \mathit{CName} \hookrightarrow \mathit{MName} \hookrightarrow (\mathit{MD} \cup \mathit{MD}^p)$ is defined. It recursively searches in superclasses for the definition of an (inherited) method m for class C. For data classes it is defined by:

$$\mathcal{M}.C.m \equiv \begin{cases} \mathcal{M}_s.C.m & \text{if } \mathcal{M}_s.C.m \neq \underline{\text{undef}} \\ \mathcal{M}.C_{super}.m & \text{if } \mathcal{M}_s.C.m = \underline{\text{undef}} \text{ and } \mathit{CDList} \equiv \mathit{CD}_1 \cdots \mathit{CD}_i \cdots \mathit{CD}_n \\ & \text{and } \mathit{CD}_i \equiv \begin{array}{ll} \texttt{data class} & C \\ \texttt{extends} & C_{super} \\ \texttt{instance variables} & x_1 \cdots x_n \\ \texttt{instance methods} & \mathit{MD}_1 \cdots \mathit{MD}_k \end{array} \\ \underline{\text{undef}} & \text{otherwise.} \end{cases}$$

The definitions of $\mathcal{M}_s$ and $\mathcal{M}$ are completed in Section 4.2.3. The static method call is introduced next to the dynamic method call to enable the "super method call" construct **self** $\hat{}m(E_1, \ldots, E_n)$ in the concrete syntax (Appendix B). Its intended meaning is to start searching for the method definition of m in the super class of the class in which the construct is placed. Notice that the class of **self** is not referred to. We will now show for the example in Table 3.1 how this super method call can be translated into a static method call of the form **self** $m_C(E_1, \ldots, E_n)$[7]. Suppose that an object of class `C` is created and then initialised: **new**(`C`) `init`. This expression will call method `init` defined in class `C`. In line 7 of this method, method `init` of class `B` is called. That method calls (line 7) method `init` of class `A`. At compile-time, the construct in line 7 of class `B` is replaced by the static method call **self** $\texttt{init}_A$ and the construct in line 7 of class `C` by **self** $\texttt{init}_B$.

Furthermore, we define function $\mathcal{I}$, which assigns unique indices to nonprimitive data methods.

Definition 19 *Let $n \in \mathbb{N}_0$ be the number of syntactically different nonprimitive data method definitions in CDList. We assume $\mathcal{I} \in \mathit{CName} \hookrightarrow \mathit{MName} \hookrightarrow \{1, \ldots, n\}$ to be any function that assigns a unique number to each different nonprimitive data method. It is such that:*

1. *$\mathcal{I}.C.m \in \{1, \ldots, n\}$ if $\mathcal{M}.C.m \equiv \begin{array}{l} m(u_1, \ldots, u_n) \\ |z_1 \cdots z_m| \\ E \end{array}$; otherwise $\mathcal{I}.C.m = \underline{\text{undef}}$*
2. *for all m, m', C and C', such that both $\mathcal{I}.C.m$ and $\mathcal{I}.C'.m'$ are in $\{1 \ldots, n\}$, we have $\mathcal{I}.C.m = \mathcal{I}.C'.m' \iff \mathcal{M}.C.m \equiv \mathcal{M}.C'.m'$.*

[7]To simplify the semantics of the static method call, the more generic form $E\ m_C(E_1, \ldots, E_n)$ is defined.

3.2.3.3 Semantics

Before the meaning of expressions can be given, a hierarchical framework of domains and ccpo's is defined. The presented framework guarantees existence of the fixed points required in establishing the semantics of POOSL's data layer. First, a set of (substochastic) probability functions is defined. These functions bind probabilities to execution states.

Definition 20 (substochastic probability function) *The set of substochastic probability functions is* $\mathcal{P} = \left\{ p \in State' \twoheadrightarrow \mathcal{R} \;\middle|\; \sum_{s' \in State'} p.s' \leq 1 \right\}$, *where* $\mathcal{R} = [0,1]$. *For* $p, q \in State' \twoheadrightarrow \mathcal{R}$, *define* $p \sqsubseteq q$ *iff* $p.s' \leq q.s'$ *for all* $s' \in State'$. *For* $p, q \in \mathcal{P}$, *define* $p \sqsubseteq_{\mathcal{P}} q$ *iff* $p.s' \leq q.s'$ *for all* $s' \in State'$.

Notice that the probabilities of all possible terminal states may not sum up to one (hence the name substochastic), for instance because of nonterminating loops. $\mathcal{P}$ is well-defined: $\sum_{s' \in State'} p.s'$ is a well-defined series because its terms $p.s'$ are non-negative, the sequence of its partial sums is bounded (by one) and $State'$ is countable. Using the fact that for such a series the summation order is irrelevant, the series converges to $\sum_{s' \in State'} p.s' = \sum_{j=1}^{\infty} p.s'_j = \lim_{n \to \infty} \sum_{j=1}^{n} p.s'_j$, where $State' = \{s'_1, s'_2, \ldots\}$ for any $s'_1, s'_2, \ldots$.

The following assertions simplify the upcoming continuity proofs and the proof that $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$ is a ccpo.

Lemma 21 *Let* $p, q \in \mathcal{P}$. *Then* $p \sqsubseteq q$ *iff* $p \sqsubseteq_{\mathcal{P}} q$.

Proof of Lemma 21 If $p \sqsubseteq q$, we have $p.s' \leq q.s'$ for all $s' \in State'$ and since $p, q \in \mathcal{P}$, we also have $p \sqsubseteq_{\mathcal{P}} q$. On the other hand, if $p \sqsubseteq_{\mathcal{P}} q$, we obtain that $p.s' \leq q.s'$ for all $s' \in State'$ and since $p, q \in \mathcal{P} \subseteq State' \twoheadrightarrow \mathcal{R}$, we also have $p \sqsubseteq q$. □

Lemma 22 $(\mathcal{R}, \leq)$ *and* $(State' \twoheadrightarrow \mathcal{R}, \sqsubseteq)$ *are ccpo's.*

Proof of Lemma 22 Let $d \in \mathcal{R}^{\omega}$ be a chain. Every non-empty, bounded subset of $\mathbb{R}$ has a least upper bound, and since $\{d_i \mid i \in \mathbb{N}\} \subseteq \mathcal{R}$ is such a non-empty, bounded subset of $\mathbb{R}$, its least upper bound $\bigsqcup_i d_i$ exists and is in $\mathcal{R}$ because $d_i \leq 1$ for all $i \in \mathbb{N}$; hence $(\mathcal{R}, \leq)$ is a ccpo. From Theorem 15 with $\mathcal{R}$, $State'$, $\leq$ and $\sqsubseteq$ substituted for D, D', $\sqsubseteq$ and $\sqsubseteq'$ respectively, we obtain that also $(State' \twoheadrightarrow \mathcal{R}, \sqsubseteq)$ is a ccpo. □

Lemma 23 *Let $d \in \mathcal{P}^\omega$ be a chain and let $V \subseteq State'$. Then* $\sum_{s' \in V} \bigsqcup d.s' = \bigsqcup \sum_{s' \in V} d.s'$.

Proof of Lemma 23 Since $\mathcal{P} \subseteq \big(State' \rightarrow \mathcal{R}\big)$ and $(State' \rightarrow \mathcal{R}, \sqsubseteq)$ is a ccpo (Lemma 22), $\bigsqcup d$ exists. Assume that $V = \{s'_1, s'_2, \ldots\}$. We can then rewrite $\sum_{s' \in V} \bigsqcup d.s'$ to $\lim_{n \to \infty} \sum_{j=1}^{n} \bigsqcup d.s'_j$. Using the definition of least upper bound, we can rewrite $\bigsqcup d.s'_j$ to $\bigsqcup_i d_i.s'_j$, which is $\lim_{i \to \infty} d_i.s'_j$ (because d is a chain), thus obtaining $\sum_{s' \in V} \bigsqcup d.s' = \lim_{n \to \infty} \sum_{j=1}^{n} \lim_{i \to \infty} d_i.s'_j = \lim_{n \to \infty} \lim_{i \to \infty} \sum_{j=1}^{n} d_i.s'_j$, because the sum is finite and the inner limit exists. Swapping the limits gives $\lim_{i \to \infty} \lim_{n \to \infty} \sum_{j=1}^{n} d_i.s'_j$ that can be rewritten to $\bigsqcup_i \sum_{s' \in V} d_i.s'$, which equals $\bigsqcup \sum_{s' \in V} d.s'$. □

Lemma 24 *Let $r, s \in \mathcal{R}^\omega$ be chains. Then* $\bigsqcup_i r_i \cdot \bigsqcup_j s_j = \bigsqcup_i \bigsqcup_j r_i \cdot s_j$.

Proof of Lemma 24 Since r and s are chains, their least upper bounds $\bigsqcup_i r_i$ and $\bigsqcup_i s_i$ exist. Rewriting the product gives $\bigsqcup_i r_i \cdot \bigsqcup_j s_j = \lim_{i \to \infty} r_i \cdot \lim_{j \to \infty} s_j = \lim_{i \to \infty} \lim_{j \to \infty} r_i \cdot s_j = \bigsqcup_i \bigsqcup_j r_i \cdot s_j$. □

We are now ready to build the semantic domain hierarchically, using the theory of chain-complete partially ordered sets discussed in Section 2.2.1, and prove some properties required for the upcoming continuity proofs.

Theorem 25 *$(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$ is a ccpo.*

Proof of Theorem 25 Let $d \in \mathcal{P}^\omega$ be a chain. According to Lemma 22, $(State' \rightarrow \mathcal{R}, \sqsubseteq)$ is a ccpo. Since $d \in \mathcal{P}^\omega \subseteq \big(State' \rightarrow \mathcal{R}\big)^\omega$, $\bigsqcup d$ exists. Using Lemma 23 and the definition of least upper bound, we deduce that $\sum_{s' \in State'} \bigsqcup d.s' = \bigsqcup \sum_{s' \in State'} d.s' = \bigsqcup_i \sum_{s' \in State'} d_i.s'$. Since $d_i \in \mathcal{P}$ for all $i \in \mathbb{N}$, we have $\sum_{s' \in State'} d_i.s' \leq 1$ and hence $\bigsqcup_i \sum_{s' \in State'} d_i.s' \leq 1$ for all $i \in \mathbb{N}$. But then $\bigsqcup d \in \mathcal{P}$, so $\big(\mathcal{P}, \sqsubseteq\big)$ is a ccpo. We conclude, using Lemma 21, that also $\big(\mathcal{P}, \sqsubseteq_{\mathcal{P}}\big)$ is a ccpo. □

Next, the semantic domain $\mathcal{S}$ is defined. A function in this domain maps a start state (begin state) to a function that defines the probability of ending up in a particular terminal state (end state).

Definition 26 (semantic domain) *Let $\mathcal{S} = State \rightarrow \mathcal{P}$. For $f, g \in \mathcal{S}$, define $f \sqsubseteq_{\mathcal{S}} g$ iff $f.s \sqsubseteq_{\mathcal{P}} g.s$ for all $s \in State$.*

Theorem 27 *$(\mathcal{S}, \sqsubseteq_{\mathcal{S}})$ and $(\mathcal{S}^n, \sqsubseteq^n_{\mathcal{S}})$ are ccpo's.*

Proof of Theorem 27 From Theorem 15 with $\mathcal{P}$, State, $\sqsubseteq_{\mathcal{P}}$ and $\sqsubseteq_{\mathcal{S}}$ substituted for D, D', $\sqsubseteq$ and $\sqsubseteq'$ we get that $(\mathcal{S}, \sqsubseteq_{\mathcal{S}})$ is a ccpo. From Corollary 17, it immediately follows that $(\mathcal{S}^n, \sqsubseteq^n_{\mathcal{S}})$ is also a ccpo. □

Definition 28 *Let $\perp_{\mathcal{R}} \in \mathcal{R}$ be such that $\perp_{\mathcal{R}} = 0$. Let $\perp_{\mathcal{P}} \in \mathcal{P}$ be such that $\perp_{\mathcal{P}}.s' = \perp_{\mathcal{R}}$ for all $s' \in \mathit{State}'$. Let $\perp_{\mathcal{S}} \in \mathcal{S}$ be such that $\perp_{\mathcal{S}}.s = \perp_{\mathcal{P}}$ for all $s \in \mathit{State}$. Finally, let $\perp^n_{\mathcal{S}} \in \mathcal{S}^n$ be such that $(\perp^n_{\mathcal{S}})_i = \perp_{\mathcal{S}}$ for all $i \in \{1, \ldots, n\}$.*

From now on it is tacitly assumed that $\perp_{\mathcal{R}}$ is defined for ccpo $(\mathcal{R}, \leq)$, $\perp_{\mathcal{P}}$ is defined for ccpo $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$, $\perp_{\mathcal{S}}$ is defined for ccpo $(\mathcal{S}, \sqsubseteq_{\mathcal{S}})$, and $\perp^n_{\mathcal{S}}$ is defined for ccpo $(\mathcal{S}^n, \sqsubseteq^n_{\mathcal{S}})$.

Theorem 29 *$\perp_{\mathcal{R}}$, $\perp_{\mathcal{P}}$, $\perp_{\mathcal{S}}$ and $\perp^n_{\mathcal{S}}$ are least elements of their respective ccpo's.*

Proof of Theorem 29 Obviously $\perp_{\mathcal{R}} = 0$ is an element of $\mathcal{R}$. For all $d \in \mathcal{R}$, $d \geq 0$ and thus $\perp_{\mathcal{R}} \leq d$ for all $d \in \mathcal{R}$. Hence $\perp_{\mathcal{R}}$ is the least element of $\mathcal{R}$.

Since $\perp_{\mathcal{P}}.s' = \perp_{\mathcal{R}} = 0$ for all $s' \in \mathit{State}'$, $\sum\limits_{s' \in \mathit{State}'} \perp_{\mathcal{P}}.s' \leq 1$ and therefore $\perp_{\mathcal{P}} \in \mathcal{P}$. For all $d \in \mathcal{P}$ and all $s' \in \mathit{State}'$, $\perp_{\mathcal{P}}.s' \leq d.s'$. Thus $\perp_{\mathcal{P}} \sqsubseteq_{\mathcal{P}} d$ for all $d \in \mathcal{P}$ and hence $\perp_{\mathcal{P}}$ is the least element of $\mathcal{P}$.

Since $\perp_{\mathcal{S}}.s = \perp_{\mathcal{P}}$ for all $s \in \mathit{State}$, $\perp_{\mathcal{S}} \in \mathcal{S}$. For all $d \in \mathcal{S}$ and all $s \in \mathit{State}$, $\perp_{\mathcal{S}}.s \sqsubseteq_{\mathcal{P}} d.s$. Thus $\perp_{\mathcal{S}} \sqsubseteq_{\mathcal{S}} d$ for all $d \in \mathcal{S}$ and hence $\perp_{\mathcal{S}}$ is the least element of $\mathcal{S}$.

Since $(\perp^n_{\mathcal{S}})_i = \perp_{\mathcal{S}}$ for all $i \in \mathbb{N}$, $\perp^n_{\mathcal{S}} \in \mathcal{S}^n$. For all $d \in \mathcal{S}^n$ and all $i \in \mathbb{N}$, $(\perp^n_{\mathcal{S}})_i \sqsubseteq_{\mathcal{S}} d_i$. Thus $\perp^n_{\mathcal{S}} \sqsubseteq^n_{\mathcal{S}} d$ for all $d \in \mathcal{S}^n$ and hence $\perp^n_{\mathcal{S}}$ is the least element of $\mathcal{S}^n$. □

The least elements stated in Definition 28 will be required later. In Section 3.2.3.1 it was discussed that because of recursive dynamic method calls, it is impossible to directly give a compositional definition of expressions, as is required by a denotational semantics. Therefore, a similar approach is taken as with the `while`. Before defining that functional (whose fixed point will define the meaning of nonprimitive method bodies), we introduce $[\![E]\!]_f$ that gives the meaning of expression E, conditional upon the approximated meaning of the nonprimitive method bodies (provided by an n-tuple $f \in \mathcal{S}^n$).

Definition 30 *Let $E \in \mathit{Exp}$, $f \in \mathcal{S}^n$ and let $[\![\cdot]\!]_{\cdot} \in \mathit{Exp} \longrightarrow \mathcal{S}^n \longrightarrow \mathcal{S}$ be defined such that the meaning of E conditional upon f, denoted by $[\![E]\!]_f \in \mathcal{S}$, is defined by semantic rules 1 through 13 in Section A.1.*

In this definition and from this point on, n (used in $\mathcal{S}^n$) is assumed to be the number of different nonprimitive data method definitions in *CDList*.

It is not immediately evident that the definition of $[\![E]\!]_f$ is sound, because of semantic rule 11. We will therefore elucidate that rule and show that it is well-defined by proving the existence of FIX $\mathcal{X}_f$ after having shown that $\mathcal{X}_f$ is continuous and using the fact that $(\mathcal{S}, \sqsubseteq_{\mathcal{S}})$ is a ccpo.

Meaning of while Since the effect of **while** E_c **do** E **od** is chosen to be the same as the effect of **if** E_c **then** E**;while** E_c **do** E **od else nil fi**, the semantics of `while` must equal:

$$\begin{aligned}
&[\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]_f.s.s' \\
&\quad = [\![\texttt{if}\ E_c\ \texttt{then}\ E\texttt{;while}\ E_c\ \texttt{do}\ E\ \texttt{od else nil fi}]\!]_f.s.s' \\
&\quad = \sum_{t'_T:\mathrm{P1}} [\![E_c]\!]_f.s.t'_T \times [\![E\texttt{;while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]_f.t_T.s' + \sum_{t'_F:\mathrm{P4}} [\![E_c]\!]_f.s.t'_F \times [\![\texttt{nil}]\!]_f.t_F.s' \\
&\quad = \sum_{t'_T:\mathrm{P1}} [\![E_c]\!]_f.s.t'_T \times \Big(\sum_{t':\mathrm{P2}} [\![E]\!]_f.t_T.t' \times [\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]_f.t.s' \Big) + \sum_{t'_F:\mathrm{P3}} [\![E_c]\!]_f.s.t'_F \\
&\quad = \sum_{t'_T:\mathrm{P1}} \sum_{t':\mathrm{P2}} [\![E_c]\!]_f.s.t'_T \times [\![E]\!]_f.t_T.t' \times [\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]_f.t.s' + \sum_{t'_F:\mathrm{P3}} [\![E_c]\!]_f.s.t'_F
\end{aligned}$$

$$\begin{aligned}
\mathrm{P1}: &\quad t'_T = (t_T, \mathit{true}); \\
\mathrm{P2}: &\quad t' = (t, \beta); \\
\mathrm{P3}: &\quad t'_F = (t_F, \mathit{false}),\ s' = (t_F, \mathit{nil}); \\
\mathrm{P4}: &\quad t'_F = (t_F, \mathit{false}).
\end{aligned}$$

The previous four lines describe the predicates P1 through P4, which are conditions that limit the set of elements used in the summations above.

Because this equation is not compositional, as is required for a denotational semantics, it cannot be used as the definition of `while`. However, we can conclude that $[\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]_f$ must be a fixed point of the following functional $\mathcal{X}_f$.

Definition 31 *Let* $E, E_c \in \mathit{Exp}$, $f \in \mathcal{S}^n$, $g \in \mathcal{S}$ *and let* $\mathcal{X}_{.} \in \mathcal{S}^n \to \mathcal{S} \to \mathcal{S}$ *be defined such that functional* $\mathcal{X}_f(g).s.s' = \sum_{t'_T:\mathrm{P1}} \sum_{t':\mathrm{P2}} [\![E_c]\!]_f.s.t'_T \times [\![E]\!]_f.t_T.t' \times g.t.s' + \sum_{t'_F:\mathrm{P3}} [\![E_c]\!]_f.s.t'_F$ *where* $\mathcal{X}_f \in \mathcal{S} \to \mathcal{S}$. *The meaning of* `while` *conditional upon* f *is defined in the usual way:* $[\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]_f = \mathrm{FIX}\,\mathcal{X}_f$.

With this definition, $[\![\cdot]\!]_.$ *is* compositional, because $[\![\cdot]\!]_.$ is only applied to the immediate constituents of `while` and not to the construct itself. To show that this definition is sound, we assert the following.

Theorem 32 $\mathcal{X}_f$ *is continuous for any* $f \in \mathcal{S}^n$.

Proof of Theorem 32 Let $f \in \mathcal{S}^n$ and let $d \in \mathcal{S}^\omega$ be a chain. By definition of $\mathcal{X}_f$, we have $\mathcal{X}_f(\bigsqcup d).s.s' = \sum_{t'_T:\mathrm{P1}} \sum_{t':\mathrm{P2}} [\![E_c]\!]_f.s.t'_T \times [\![E]\!]_f.t_T.t' \times \bigsqcup d.t.s' + \sum_{t'_F:\mathrm{P3}} [\![E_c]\!]_f.s.t'_F$. Using Lemma 24, we can rewrite this to $\sum_{t'_T:\mathrm{P1}} \sum_{t':\mathrm{P2}} \bigsqcup [\![E_c]\!]_f.s.t'_T \times [\![E]\!]_f.t_T.t' \times d.t.s' + \sum_{t'_F:\mathrm{P3}} [\![E_c]\!]_f.s.t'_F$. Applying Lemma 23 twice gives $\bigsqcup \sum_{t'_T:\mathrm{P1}} \sum_{t':\mathrm{P2}} [\![E_c]\!]_f.s.t'_T \times [\![E]\!]_f.t_T.t' \times d.t.s' + \sum_{t'F:\mathrm{P3}} [\![E_c]\!]_f.s.t'_F$, which is $\bigsqcup \mathcal{X}_f(d).s.s'$. From this we infer that $\mathcal{X}_f$ is continuous for any $f \in \mathcal{S}^n$. □

Lemma 33 FIX $\mathcal{X}_f$ *exists for any* $f \in \mathcal{S}^n$*.*

Proof of Lemma 33 Since $(\mathcal{S}, \sqsubseteq_\mathcal{S})$ is a ccpo with least element $\perp_\mathcal{S}$, and $\mathcal{X}_f$ is continuous for any $f \in \mathcal{S}^n$ (Theorem 32), we conclude that FIX $\mathcal{X}_f$ exists for any $f \in \mathcal{S}^n$ by virtue of Theorem 13. □

This shows that the meaning of `while` is well-defined by semantic rule 11; therefore, the definition of $[\![E]\!]_f$ is sound. Before presenting the semantic function that defines the meaning of expressions, a few lemmas are asserted to simplify the proof of continuity of $[\![E]\!]_. \in \mathcal{S}^n \to \mathcal{S}$.

Lemma 34 *Let* $d \in (\mathcal{S}^n)^\omega$ *be a chain, let* $k \in \mathbb{N}_0$ *and let* $j \in \mathbb{N}$*. If* $\mathcal{X}_.$ *is continuous,* $\mathcal{X}^k_{d_j}(\perp_\mathcal{S})$ *is a chain both in* k *and in* j*.*

Proof of Lemma 34 We first show that $\mathcal{X}^k_{d_j}(\perp_\mathcal{S})$ is a chain in j for any $k \in \mathbb{N}_0$. Since $\mathcal{X}_.$ is monotone, $\mathcal{X}_{d_j} \sqsubseteq_\mathcal{S} \mathcal{X}_{d_{j+1}}$ and especially $\mathcal{X}_{d_j}\big(\mathcal{X}_{d_j}(\perp_\mathcal{S})\big) \sqsubseteq_\mathcal{S} \mathcal{X}_{d_{j+1}}\big(\mathcal{X}_{d_j}(\perp_\mathcal{S})\big) \sqsubseteq_\mathcal{S} \mathcal{X}_{d_{j+1}}\big(\mathcal{X}_{d_{j+1}}(\perp_\mathcal{S})\big)$, and in general $\mathcal{X}^k_{d_j}(\perp_\mathcal{S}) \sqsubseteq_\mathcal{S} \mathcal{X}^k_{d_{j+1}}(\perp_\mathcal{S})$ for any $k \in \mathbb{N}_0$. So, $\mathcal{X}^k_{d_j}(\perp_\mathcal{S})$ is a chain in j.

To show that $\mathcal{X}^k_{d_j}(\perp_\mathcal{S})$ is also a chain in k, choose some $j \in \mathbb{N}$. Since $\perp_\mathcal{S} \sqsubseteq_\mathcal{S} \mathcal{X}_{d_j}(\perp_\mathcal{S})$ and $\mathcal{X}_f$ is monotone, also $\mathcal{X}_{d_j}(\perp_\mathcal{S}) \sqsubseteq_\mathcal{S} \mathcal{X}_{d_j}\big(\mathcal{X}_{d_j}(\perp_\mathcal{S})\big) = \mathcal{X}^2_{d_j}(\perp_\mathcal{S})$, and in general $\mathcal{X}^k_{d_j}(\perp_\mathcal{S}) \sqsubseteq_\mathcal{S} \mathcal{X}^{k+1}_{d_j}(\perp_\mathcal{S})$ for any $j \in \mathbb{N}$, so $\mathcal{X}^k_{d_j}(\perp_\mathcal{S})$ is a chain in k. □

Lemma 35 *Let* $k \in \mathbb{N}_0$ *and let* $j \in \mathbb{N}$*. If* $\mathcal{X}_.$ *is continuous,* $\mathcal{X}^k_{\bigsqcup d}(\perp_\mathcal{S}) = \bigsqcup \mathcal{X}^k_d(\perp_\mathcal{S})$*.*

Proof of Lemma 35 The proof is trivial for $k = 0$: $\mathcal{X}^0_{\bigsqcup d}(\perp_\mathcal{S}) = \perp_\mathcal{S} = \bigsqcup \mathcal{X}^0_d(\perp_\mathcal{S})$. Now assume that $\mathcal{X}^k_{\bigsqcup d}(\perp_\mathcal{S}) = \bigsqcup \mathcal{X}^k_d(\perp_\mathcal{S})$ (induction hypothesis). Unfolding $\mathcal{X}^{k+1}_{\bigsqcup d}(\perp_\mathcal{S})$ gives $\mathcal{X}_{\bigsqcup_i d_i}\Big(\mathcal{X}^k_{\bigsqcup_j d_j}(\perp_\mathcal{S})\Big)$. Using continuity of $\mathcal{X}_.$ and the induction hypothesis, we can rewrite this to $\bigsqcup_i \mathcal{X}_{d_i}\Big(\bigsqcup_j \mathcal{X}^k_{d_j}(\perp_\mathcal{S})\Big)$, which is $\bigsqcup_i \bigsqcup_j \mathcal{X}_{d_i}\big(\mathcal{X}^k_{d_j}(\perp_\mathcal{S})\big)$ (again using continuity of $\mathcal{X}_.$). By virtue of Lemma 34, we can apply Theorem 18 to rewrite this to $\bigsqcup_i \mathcal{X}_{d_i}\big(\mathcal{X}^k_{d_i}(\perp_\mathcal{S})\big)$, which is $\bigsqcup \mathcal{X}^{k+1}_d(\perp_\mathcal{S})$. □

Theorem 36 *For each $E \in Exp$, $[\![E]\!]_{.}$ is continuous.*

Proof of Theorem 36 The proof is by induction on the syntactic structure of the expressions in $E \in Exp$. First, continuity of $[\![E]\!]_{.}$ is shown for the basis constructs — primitive data objects, variables, **self** and object creation. Next, continuity is demonstrated for the composite constructs under the assumption that their immediate constituents are continuous (induction hypothesis). By induction we then obtain continuity of $[\![E]\!]_{.}$ for any expression $E \in Exp$.

Primitive data objects Let $d \in (\mathcal{S}^n)^\omega$ be a chain. For any $s \in State$ and $s' \in State'$, $[\![\underline{\gamma}]\!]_{\bigsqcup d}.s.s' = \bigsqcup [\![\underline{\gamma}]\!]_{d}.s.s'$ because the definition of $[\![\underline{\gamma}]\!]_{f}.s.s'$ is independent of f (Appendix A.1, rule 1). Furthermore, point 4 of the proof of Theorem 15 can be used to rewrite $\bigsqcup [\![\underline{\gamma}]\!]_{d}.s.s'$ to $\left(\bigsqcup [\![\underline{\gamma}]\!]_{d}\right).s.s'$. But then $[\![\underline{\gamma}]\!]_{\bigsqcup d} = \bigsqcup [\![\underline{\gamma}]\!]_{d}$ for every chain $d \in (\mathcal{S}^n)^\omega$ and therefore $[\![\underline{\gamma}]\!]_{.}$ is continuous.

Instance variables, local variables, `self`, current time and object creation. The cases $[\![x]\!]_{f}$, $[\![u]\!]_{f}$, $[\![\texttt{self}]\!]_{f}$, $[\![\texttt{currentTime}]\!]_{f}$ and $[\![\texttt{new}(C)]\!]_{f}$ are similar to $[\![\underline{\gamma}]\!]_{f}$ and their proofs are therefore omitted.

Assignment to instance variables Let $d \in (\mathcal{S}^n)^\omega$ be a chain. For any $s \in State$ and $s' \in State'$, we have $[\![x\texttt{:=}E]\!]_{\bigsqcup d}.s.s' = \sum_{t':\text{P1}} [\![E]\!]_{\bigsqcup d}.s.t'$ with proposition P1 as defined in semantic rule 7 in Appendix A.1. The induction hypothesis ensures that $[\![E]\!]_{.}$ is continuous, so $\sum_{t':\text{P1}} [\![E]\!]_{\bigsqcup d}.s.t' = \sum_{t':\text{P1}} \bigsqcup [\![E]\!]_{d}.s.t'$, which is $\bigsqcup \sum_{t':\text{P1}} [\![E]\!]_{d}.s.t'$ by Lemma 23. This equals $\bigsqcup [\![x\texttt{:=}E]\!]_{d}$ and hence $[\![x\texttt{:=}E]\!]_{.}$ is continuous.

Assignment to local variables The proof for $[\![u\texttt{:=}E]\!]_{.}$ is analogous to that of $[\![x\texttt{:=}E]\!]_{.}$ and is omitted.

Sequential composition Let $d \in (\mathcal{S}^n)^\omega$ be a chain. For any $s \in State$ and $s' \in State'$, we have $[\![E_1\texttt{;}E_2]\!]_{\bigsqcup d}.s.s' = \sum_{t':\text{P1}} [\![E_1]\!]_{\bigsqcup d}.s.t' \times [\![E_2]\!]_{\bigsqcup d}.t.s'$. By the induction hypothesis and Lemma 24 this equals $\sum_{t':\text{P1}} \bigsqcup\left([\![E_1]\!]_{d}.s.t' \times [\![E_2]\!]_{d}.t.s'\right)$, which is $\bigsqcup \sum_{t':\text{P1}} \left([\![E_1]\!]_{d}.s.t' \times [\![E_2]\!]_{d}.t.s'\right)$ by Lemma 23. But this equals $\bigsqcup [\![E_1\texttt{;}E_2]\!]_{d}.s.s'$, so $[\![E_1\texttt{;}E_2]\!]_{\bigsqcup d} = \bigsqcup [\![E_1\texttt{;}E_2]\!]_{d}$ and hence $[\![E_1\texttt{;}E_2]\!]_{.}$ is continuous.

If The proof for continuity of $[\![\texttt{if } E_c \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]_{.}$ is skipped because it is similar to the one of sequential composition.

While Let $d \in (\mathcal{S}^n)^\omega$ be a chain. We first show that $\mathcal{X}_{.}$ is continuous. By semantic rule 11, for any $g \in \mathcal{S}$, $s \in State$ and $s' \in State'$, we have $\mathcal{X}_{\bigsqcup d}(g).s.s' = \sum_{t'_T:\text{P1}} \sum_{t':\text{P2}} [\![E_c]\!]_{\bigsqcup d}.s.t'_T \times [\![E]\!]_{\bigsqcup d}.t_T.t' \times g.t.s' + \sum_{t'_F:\text{P3}} [\![E_c]\!]_{\bigsqcup d}.s.t'_F$. Since the induction hypothesis guarantees continuity of $[\![E_c]\!]_{.}$ and $[\![E]\!]_{.}$, this equals $\sum_{t'_T:\text{P1}} \sum_{t':\text{P2}} \bigsqcup [\![E_c]\!]_{d}.s.t'_T \times$

$\bigsqcup [\![E]\!]_d.t_T.t' \times g.t.s' + \sum_{t'_F:\mathrm{P3}} \bigsqcup [\![E_c]\!]_d.s.t'_F$, which is $\bigsqcup \Big(\sum_{t'_T:\mathrm{P1}} \sum_{t':\mathrm{P2}} [\![E_c]\!]_d.s.t'_T \times [\![E]\!]_d.t_T.t' \times g.t.s' + \sum_{t'_F:\mathrm{P3}} [\![E_c]\!]_d.s.t'_F \Big)$ with the help of Lemmas 23 and 24. This equals $\bigsqcup \mathcal{X}_d(g).s.s'$, so $\mathcal{X}_{\bigsqcup d} = \bigsqcup \mathcal{X}_d$ for any $d \in (\mathcal{S}^n)^\omega$ and hence $\mathcal{X}_.$ is continuous.

For $[\![$**while** E_c **do** E **od**$]\!]_.$ to be continuous, FIX $\mathcal{X}_.$ must be continuous. Notice that Theorem 33 assures the existence of FIX $\mathcal{X}_f$ for any $f \in \mathcal{S}^n$. By Theorem 13 we get FIX $\mathcal{X}_{\bigsqcup_j d_j} = \bigsqcup_{k \in \mathbb{N}_0} \mathcal{X}^k_{\bigsqcup_j d_j}(\perp_S)$ and using Lemma 35 gives $\bigsqcup_{k \in \mathbb{N}_0} \bigsqcup_j \mathcal{X}^k_{d_j}(\perp_S)$. Lemma 34 allows the use of Theorem 18 to swap the least upper bounds and rewrite this to $\bigsqcup_j \bigsqcup_{k \in \mathbb{N}_0} \mathcal{X}^k_{d_j}(\perp_S)$, which is just $\bigsqcup_j$ FIX $\mathcal{X}_{d_j}$. So FIX $\mathcal{X}_{\bigsqcup d} = \bigsqcup$ FIX $\mathcal{X}_d$ for any $d \in (\mathcal{S}^n)^\omega$ and hence FIX $\mathcal{X}_.$ is continuous.

Method call The proof for the static method call is omitted because it is essentially the same as this proof for the dynamic method call. Let $d \in (\mathcal{S}^n)^\omega$ be a chain. The proof for nonprimitive methods is given first.

Nonprimitive methods
For any $s \in \mathit{State}$ and $s' \in \mathit{State}'$, we have $[\![E\ m(E_1, \dots, E_n)]\!]_{\bigsqcup d}.s.s' = \sum_{t'_0:\mathrm{P0}} \cdots \sum_{t'_n:\mathrm{P}n} \sum_{u:\mathrm{Q}} [\![E]\!]_{\bigsqcup_j d_j}.s.t'_0 \times [\![E_1]\!]_{\bigsqcup_j d_j}.t_0.t'_1 \times \cdots \times [\![E_n]\!]_{\bigsqcup_j d_j}.t_{n-1}.t'_n \times \bigsqcup_j d_{ji}.u.w'$ (where i corresponds to method m as specified in semantic rule 12). Since we know that $[\![E]\!]_.$ and $[\![E_1]\!]_.$,…,$[\![E_n]\!]_.$ are continuous (by virtue of the induction hypothesis), this equals $\sum_{t'_0:\mathrm{P0}} \cdots \sum_{t'_n:\mathrm{P}n} \sum_{u:\mathrm{Q}} \bigsqcup_j [\![E]\!]_{d_j}.s.t'_0 \times \bigsqcup_j [\![E_1]\!]_{d_j}.t_0.t'_1 \times \cdots \times \bigsqcup_j [\![E_n]\!]_{d_j}.t_{n-1}.t'_n \times \bigsqcup_j d_{ji}.u.w'$. From Definition 16 we obtain that $d_{ji} \sqsubseteq_S d_{(j+1)i}$ for all $j \in \mathbb{N}$ and $i \in \{1, \dots, n\}$, whence d_{ji} is a chain in j. We can thus repeatedly apply Theorem 18 to join the upper bounds and use Lemma 23 to swap that upper bound with the summation. This gives $\bigsqcup_j \sum_{t'_0:\mathrm{P0}} \cdots \sum_{t'_n:\mathrm{P}n} \sum_{u:\mathrm{Q}} [\![E]\!]_{d_j}.s.t'_0 \times [\![E_1]\!]_{d_j}.t_0.t'_1 \times \cdots \times [\![E_n]\!]_{d_j}.t_{n-1}.t'_n \times d_{ji}.u.w'$, which equals $\bigsqcup [\![E\ m(E_1, \dots, E_n)]\!]_d.s.s'$. But then $[\![E\ m(E_1, \dots, E_n)]\!]_{\bigsqcup d} = \bigsqcup [\![E\ m(E_1, \dots, E_n)]\!]_d$ for all $d \in (\mathcal{S}^n)^\omega$ and hence $[\![E\ m(E_1, \dots, E_n)]\!]_.$ is continuous for nonprimitive methods.

Primitive methods
For any $s \in \mathit{State}$ and $s' \in \mathit{State}'$, we have $[\![E\ m(E_1, \dots, E_n)]\!]_{\bigsqcup d}.s.s' = \sum_{t'_0:\mathrm{P0}} \cdots \sum_{t'_n:\mathrm{P}n} \sum_{u:\mathrm{Q}} [\![E]\!]_{\bigsqcup d}.s.t'_0 \times [\![E_1]\!]_{\bigsqcup d}.t_0.t'_1 \times \cdots \times [\![E_n]\!]_{\bigsqcup d}.t_{n-1}.t'_n \times \bigsqcup \mu.u.w'$. With similar steps as in the case for nonprimitive methods this can be shown to be equal to $\bigsqcup \sum_{t'_0:\mathrm{P0}} \cdots \sum_{t'_n:\mathrm{P}n} \sum_{u:\mathrm{Q}} [\![E]\!]_d.s.t'_0 \times [\![E_1]\!]_d.t_0.t'_1 \times \cdots \times [\![E_n]\!]_d.t_{n-1}.t'_n \times \mu.u.w'$, that is $\bigsqcup [\![E\ m(E_1, \dots, E_n)]\!]_d.s.s'$. Hence $[\![E\ m(E_1, \dots, E_n)]\!]_.$ is also continuous for primitive methods. This completes the proof for the method call.

This concludes the proof that $[\![E]\!]_.$ is continuous for any expression $E \in \mathit{Exp}$. □

We now define the functional that is used to establish the (unconditional) semantics of any expression $E \in \mathit{Exp}$.

Definition 37 *Let $f \in \mathcal{S}^n$ and let $B_i = \mathcal{M}.C.m$ denote the meaning of method m in class C for $i = \mathcal{I}.C.m$. Let $\mathcal{B} \in \mathcal{S}^n \longrightarrow \mathcal{S}^n$, such that functional $\mathcal{B}(f) = \big([\![B_1]\!]_f, \ldots, [\![B_n]\!]_f\big)$.*

The meaning of an expression is then given by $[\![\,\cdot\,]\!]$, the *semantic function*.

Definition 38 (semantic function) *Let $[\![\,\cdot\,]\!] \in \mathit{Exp} \longrightarrow \mathcal{S}$ be such that $[\![E]\!] = [\![E]\!]_{\mathrm{FIX}\,\mathcal{B}}$ for any $E \in \mathit{Exp}$. $[\![E]\!]$ denotes the* meaning *of expression E.*

As with the definition of `while`, this is a sound definition only if $\mathrm{FIX}\,\mathcal{B}$ exists. Existence can be proved if $(\mathcal{S}^n, \sqsubseteq^n_{\mathcal{S}})$ is a ccpo —which holds by virtue of Theorem 27— and if $\mathcal{B}$ is continuous.

Theorem 39 *$\mathcal{B}$ is continuous.*

Proof of Theorem 39 Let $f \in (\mathcal{S}^n)^\omega$. By definition of $\mathcal{B}$ we have $\mathcal{B}(\bigsqcup f) = \big([\![B_1]\!]_{\bigsqcup f}, \ldots, [\![B_n]\!]_{\bigsqcup f}\big)$. Using continuity of $[\![E]\!]_{.}$ for any $E \in \mathit{Exp}$, this can be rewritten to $\big(\bigsqcup[\![B_1]\!]_f, \ldots, \bigsqcup[\![B_n]\!]_f\big)$, which equals $\bigsqcup\big([\![B_1]\!]_f, \ldots, [\![B_n]\!]_f\big)$, by definition of upper bound of n-tuples. But this is $\bigsqcup \mathcal{B}(f)$ and hence $\mathcal{B}$ is continuous. □

Theorem 40 $\mathrm{FIX}\,\mathcal{B}$ *exists.*

Proof of Theorem 40 Because $\mathcal{B}$ is continuous and $(\mathcal{S}^n, \sqsubseteq^n_{\mathcal{S}})$ is a ccpo with least element $\bot^n_{\mathcal{S}}$, $\mathrm{FIX}\,\mathcal{B}$ exists by virtue of Theorem 13. □

Because $\mathrm{FIX}\,\mathcal{B}$ exists, the semantic function $[\![\,\cdot\,]\!]$ is well-defined. This almost concludes the semantics of the data layer. What remains to be described is the behaviour of the primitive methods.

3.2.3.4 Primitive Methods

This section discusses the definition of $\mathcal{D} \in \mathit{CName} \hookrightarrow \mathit{MName} \hookrightarrow \mathcal{S}$, which specifies the behaviour of primitive data methods. Only the most interesting methods are discussed, others can easily be derived from the set of presented methods.

RandomNumberGenerator.Random
Let $s \in \mathit{State}$ and let $\varrho \in \mathbb{I}$ such that $\varrho > 0$. We require from *CDList* that:

$$\mathcal{M}.\mathit{RandomNumberGenerator}.\mathit{Random} \equiv \texttt{random( )}\ \ \textbf{\texttt{primitive}}.$$

Then $\mathcal{D}.\mathit{RandomNumberGenerator}.\mathit{Random} = f$, where $f \in \mathcal{S}$ is such that:

$$f.s.s' = \begin{cases} \frac{1}{\varrho} & \text{if } s' = (s, \beta) \text{ where } \beta \in \{1, \ldots, \varrho\} \\ 0 & \text{otherwise.} \end{cases}$$

The random number generator produces an integer that falls within $\{1, \varrho\}$ with equal probability. This is the source of transitions with probabilities other than one or zero. The range of random numbers (determined by ϱ) can be chosen to easily fit the implementation, as is shown in Section 3.3.4.

The following functions serve as an example for arithmetic primitive methods.

Integer.+
We require from *CDList* that:

$\mathcal{M}.\mathit{Integer}.+ \equiv$ `+`(x)
`primitive`.

Then $\mathcal{D}.\mathit{Integer}.+$ is defined by:

$$\mathcal{D}.\mathit{Integer}.+.s.s' = \begin{cases} 1 & \text{if } s = (\sigma, \lambda, \tau) \text{ such that } \lambda = (\gamma_1, \psi) \text{ with } \gamma_1 \in \mathbb{I} \text{ and } \mathrm{Dom}(\psi) = \{x\} \text{ with } \psi.x = \gamma_2 \text{ and } \gamma_2 \in \mathbb{I} \cup \mathbb{R} \text{ and } s' = (s, \gamma_3), \text{ where } \underline{\gamma_3} = \underline{\gamma_1} + \underline{\gamma_2} \\ 0 & \text{otherwise.} \end{cases}$$

Real.≤
We require from *CDList* that:

$\mathcal{M}.\mathit{Real}.\leq \equiv$ `<=`(x)
`primitive`.

Then $\mathcal{D}.\mathit{Real}.\leq$ is defined by:

$$\mathcal{D}.\mathit{Real}.\leq.s.s' = \begin{cases} 1 & \text{if } s = (\sigma, \lambda, \tau) \text{ such that } \lambda = (\gamma_1, \psi) \text{ with } \gamma_1 \in \mathbb{R} \text{ and } \mathrm{Dom}(\psi) = \{x\} \text{ with } \psi.x = \gamma_2 \text{ and } \gamma_2 \in \mathbb{I} \cup \mathbb{R} \text{ and } s' = (s, \gamma_3), \text{ where } \underline{\gamma_3} = \underline{\gamma_1} \leq \underline{\gamma_2} \\ 0 & \text{otherwise.} \end{cases}$$

The shallowcopy method copies an object, and has each instance variable refer to the object referred to by the corresponding instance variable of the original. Its definition is as follows.

Object.ShallowCopy
We require from *CDList* that:

$\mathcal{M}.\mathit{Object.ShallowCopy} \equiv$ `shallowcopy( )`
`primitive`.

Then $\mathcal{D}.\mathit{Object.ShallowCopy} = f$ where $f \in \mathcal{S}$ is defined by:

$$f.s.s' = \begin{cases} 1 & \text{if } s = (\sigma, \lambda, \tau),\ \lambda = (\delta, \psi),\ s' = (s, \delta) \text{ and } \delta \in \mathit{PDObj} \\ 1 & \text{if } s = (\sigma, \lambda, \tau),\ \lambda = (\delta, \psi),\ s' = \big((\sigma', \lambda, \tau'), \widehat{n}\big) \text{ and } \delta \in \mathit{NDObj} \text{ where } n = \mathrm{MaxId}(\sigma) + 1,\ \sigma' = \sigma\{\phi/\widehat{n}\} \text{ and } \tau' = \tau\{\tau.\delta/\widehat{n}\}, \text{ where } \mathrm{Dom}(\phi) = \mathcal{V}(\tau.\delta) \text{ and } \phi(x) = \sigma.\delta.x \text{ for each } x \in \mathrm{Dom}(\phi) \\ 0 & \text{otherwise.} \end{cases}$$

Building upon the shallowcopy is the deepcopy method that not only copies an object, but also the objects that object (indirectly) refers to. The following definitions stem from the definition of the deepcopy method in [41], where additional information and soundness proofs can be found. To calculate the set of objects that should be copied, function $\mathcal{C}_{\beta,\sigma,\tau} \in \mathbf{2}^{\mathrm{Dom}(\sigma)} \rightarrow \mathbf{2}^{\mathrm{Dom}(\sigma)}$ is introduced, which is defined for $V \subseteq \mathrm{Dom}(\sigma)$ by:

$$\mathcal{C}_{\beta,\sigma,\tau}(V) = \begin{cases} \varnothing & \text{if } \beta \in \mathit{PDObj} \\ \{\beta\} \cup V \cup V_{\mathrm{ref}} & \text{if } \beta \in \mathit{NDObj}, \\ & \text{with } V_{\mathrm{ref}} = \big\{\sigma.\beta'.x \in \mathit{NDObj} \;\big|\; \text{for all } \beta' \in V \\ & \qquad\qquad \text{and } x \in \mathrm{Dom}\big(\mathcal{V}(\tau.\beta')\big)\big\} \end{cases}$$

$\mathrm{FIX}\,\mathcal{C}_{\beta,\sigma,\tau}$ is the set containing exactly β and all of its (indirect) references. Function $\mathit{DC} \in \mathit{DObj} \times \Sigma \times T \rightarrow \mathit{DObj} \times \Sigma \times T$ isolates the objects in this set, and is defined by:

$$\mathit{DC}(\beta,\sigma,\tau) = \begin{cases} (\beta, \sigma\restriction \mathrm{FIX}\,\mathcal{C}_{\beta,\sigma,\tau}, \tau\restriction \mathrm{FIX}\,\mathcal{C}_{\beta,\sigma,\tau}) & \text{if } \beta \in \mathit{NDObj} \\ (\beta, \sigma', \tau') & \text{if } \beta \in \mathit{PDObj}, \text{ where} \\ & \mathrm{Dom}(\sigma') = \mathrm{Dom}(\tau') = \varnothing \end{cases}$$

To duplicate this set without changing the existing data structures stored in σ, the following relabelling function $\mathrm{Relabel}_{+k} \in (\mathit{DObj} \times \Sigma \times T) \rightarrow (\mathit{DObj} \times \Sigma \times T)$ is presented. It increases the object identifiers of nonprimitive data objects by k, while leaving primitive objects unchanged, and is defined by $\mathrm{Relabel}_{+k}(\beta,\sigma,\tau) = (\beta',\sigma',\tau')$, where:

1. $\mathrm{Dom}(\sigma') = \mathrm{Dom}(\tau') = \big\{\widehat{n{+}k} \;\big|\; \widehat{n} \in \mathrm{Dom}(\sigma)\big\}$
2. for all $\widehat{n} \in \mathrm{Dom}(\sigma)$ and $x \in \mathrm{Dom}(\sigma.\widehat{n})$:
$$\sigma'.\widehat{n{+}k}\,.\,x = \begin{cases} \widehat{p{+}k} & \text{if } \sigma.\widehat{n}.x = \widehat{p} \in \mathit{NDObj} \\ \gamma & \text{if } \sigma.\widehat{n}.x = \gamma \in \mathit{PDObj} \end{cases}$$
3. for all $\widehat{n} \in \mathrm{Dom}(\sigma)$, $\quad \tau'.\widehat{n{+}k} = \tau.\widehat{n}$.

The primitive deepcopy method is then defined as follows.

Object.DeepCopy
We require from *CDList* that:

$\mathcal{M}.\mathit{Object.DeepCopy} \equiv$ `deepcopy( )`
`primitive`.

Then $\mathcal{D}.\mathit{Object.DeepCopy} = f$ where $f \in \mathcal{S}$ is defined by:

$$f.s.s' = \begin{cases} 1 & \text{if } s = (\sigma,\lambda,\tau),\ \lambda = (\delta,\psi),\ s' = (s,\delta) \text{ and } \delta \in \mathit{PDObj} \\ 1 & \text{if } s = (\sigma,\lambda,\tau),\ \lambda = (\delta,\psi),\ s' = \big((\sigma \cup \sigma', \lambda, \tau \cup \tau'), \alpha\big) \text{ and} \\ & \delta \in \mathit{NDObj}, \text{ where } (\alpha,\sigma',\tau') = \mathrm{Relabel}_{+\mathrm{MaxId}(\sigma)}\big(\mathit{DC}(\delta,\sigma,\tau)\big) \\ 0 & \text{otherwise.} \end{cases}$$

3.3 Implementation

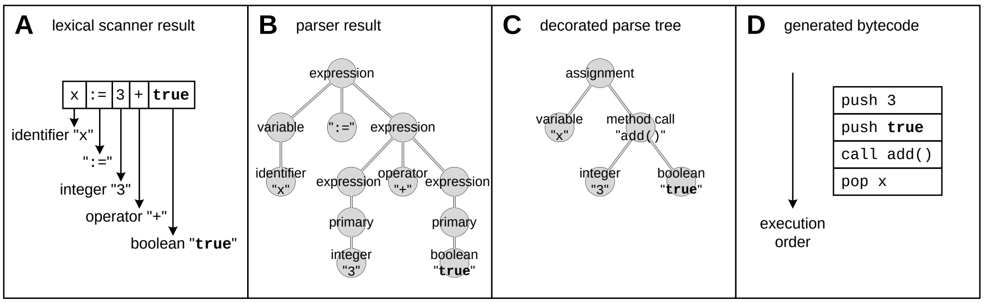
Section 3.2 formally specifies the data layer of POOSL: its abstract syntax and its meaning. Several aspects still need attention before POOSL models can actually be executed. Ambiguities in the abstract syntax, which have been discussed in Section 3.2, will have to be removed before a compiler can be constructed that parses POOSL descriptions. Then, a platform is presented for the actual execution of expressions, strictly conforming to the denotational semantics given in Section 3.2.

3.3.1 Compiler

The abstract syntax discussed in Section 3.2.1 specifies how data classes, methods and expressions are built, but it provides insufficient information for constructing unique parse trees. An extension of the abstract syntax, the *concrete syntax*, has additional information that removes the ambiguities, allowing it to be parsed by a compiler. The concrete syntax of POOSL's data layer is defined in Appendix B.

After reading a specification, the compiler's *lexical scanner* tries to recognise syntactic elements such as keywords, identifiers, variables and operators. These elements are fed into the *parser*, a finite state machine that uses the elements to construct a parse tree, based on the syntactic rules of POOSL. This initial parse tree is then *decorated*, that is, information stored in the nodes of the parse tree is rearranged, transformed and extended. The decorated parse tree finally generates compiled bytecode functioning as a description for calculating the data expressions in the original POOSL specification — this is comparable with machine code for computers.

The example in Figure 3.2 shows the subsequent steps as an expression from the human readable POOSL specification is being transformed into a machine-readable fragment of bytecode. The lexical scanner recognises the following syntactic elements in the character string `x := 3 + true` (**A**): an identifier "`x`", "`:=`", an integer "`3`", an operator "`+`" and finally a boolean "**`true`**". From these elements, the parser builds the initial parse tree shown in **B**, which is transformed during the decoration process into the tree in **C**. **D** shows the generated fragment of bytecode for computing this expression.



```
push 3
push true
call add()
pop x
```

Figure 3.2: Compilation steps for data expressions. See text for a detailed explanation.

Besides checking a specification for correct syntax, the compiler also enforces most[8] of the context conditions (stated in Section 3.2.2) to be satisfied. Notice that the example code is syntactically correct, even though it will generate a error at run time, as the addition (+) does not accept a boolean parameter.

3.3.2 Virtual Machine

The platform introduced here executes POOSL data expressions in conformance with their semantics (discussed in Section 3.2.3). The semantic rules listed in Appendix A.1 are used as a prescription for introducing the features of this so-called *virtual machine*. Figure 3.3 depicts an overview of this execution platform. The program consists of bytecode instructions assembled by the compiler for executing the data expressions found in the POOSL specification. The stack contains references to data objects that may hold intermediate results (the stack pointer SP refers to the top element). The data objects themselves are allocated on a heap, which is managed by a garbage collector. The processor is a finite state machine (FSM) that performs calculations based on the instruction present at the program counter (PC). The following paragraphs discuss in more depth the various features of this virtual machine.

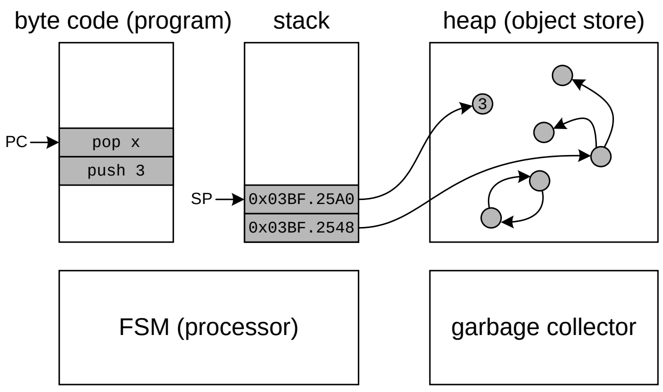


Figure 3.3: Overview of the virtual machine.

References to Objects Let $s \in State$ be the current state of an executing object, where $s = (\sigma, \lambda, \tau)$ and $\lambda = (\delta, \psi)$. The environment in which expressions are executed, called a *variables context*, can be split into two parts: global variables and local variables. The global variables are always present; they are the instance variables of the executing object, which is either a data object or a process.

Variables refer to data objects which, in turn, can also have references to (other) data objects through their instance variables. The objects (indirectly) known through the global variables of the executing object are captured by $\phi = \sigma(\delta)$. The *local* variables context is only present if the execution takes place within a data method, and consists of the parameters and local variables of that method. The objects reachable through the local variables are captured by ψ.

[8]Some of the checks can only be performed by the execution engine during run-time.

The virtual machine represents data objects by instances of class `PDO` and references are pointers to such objects. References of a variables context are stored in an array of pointers to objects of class `PDO`. The compiler maps each variable to a unique index in that array. Local variables are stored separately from global variables allowing the set of local variables to be changed efficiently upon entering and leaving the body of a data method.

Stack During the execution of an expression, temporary objects may be constructed retaining intermediate computational results. The virtual machine uses a stack to keep track of these possibly unbound objects. To this purpose, the stack holds pointers to the objects that could otherwise get lost.

The ordinary `push` and `pop` instructions allow references to be stored onto or be removed from the stack. Semantic rule 1 states that the evaluation of a constant expression leaves the current state s unchanged, and that it results in an object representing that constant. If the constant expression is part of a compound expression, its result will be used in the remaining evaluation of that compound expression. This translates to a `push` instruction that creates a primitive data object for representing the constant and leaves a reference to it on the stack.

Evaluation of an instance variable (semantic rule 2) also leaves state s unchanged, but now the result is the object that is referred to by the respective variable. The virtual machine will lookup the reference stored in the global variables context, using the index of the variable, and push the reference onto the stack. For evaluation of a *local* variable (semantic rule 3) the virtual machine uses the local variables context instead.

Assignment to variables introduces a `copy` instruction. Semantic rule 7 shows the assignment of β to global variable x, where β is the result of evaluating some expression $E \in Exp$. State s is changed such that it captures the new reference of variable x to object β: $\sigma'(\delta) = \sigma(\delta)\{\beta/x\}$. The virtual machine reflects this state change by copying the reference at the top of the stack to the array at the index of the global variable. Because the result of the assignment is β, the reference is left on top of the stack. Assignment to local variables (semantic rule 8) is handled in a similar fashion.

Data Classes Table When the compiler analyses a POOSL specification, it will create a table of the data classes it encounters. Each entry stores information such as the number of global variables of the data class, its methods and its superclass. During execution, the virtual machine's *object constructor* will lookup this information and use it to allocate sufficient memory for new objects and to initialise them properly.

The `push` instruction is used for object creation of nonprimitive data objects (semantic rule 6) in a similar way as for implementing semantic rules 1, 2 and 3: the virtual machine's object constructor creates a new instance of the specified class C and pushes a reference[9] to it onto the stack. The constructor stores references to **`nil`** at

[9]Semantic rule 6 refers to the new object by $\widehat{n}$ whereas the virtual machine can uniquely identify the constructed object by its address.

each position in the object's array of global variables, effectively setting all instance variables to *nil* as is required by semantic rule 6.

The type-mapping function τ is adapted to $\tau' = \tau\{C/\widehat{n}\}$ to store the type of $\widehat{n}$. The virtual machine distributes this type-mapping function over the created objects, by adding to each object a pointer to its class. This implementation choice will be clarified when methods are discussed.

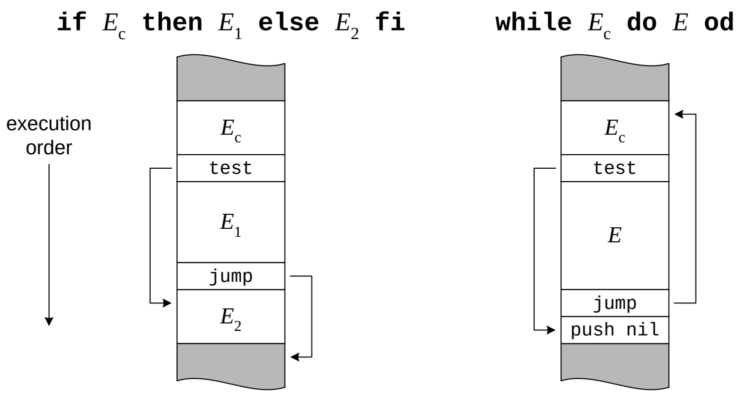

if E_c **then** E_1 **else** E_2 **fi**
while E_c **do** E **od**
execution order
E_c
test
E_1
jump
E_2
E
push nil

Figure 3.4: Code layout for `if` and `while`.

Conditional Execution The execution of `if` and `while` raises the need for *jumping*. Depending on the condition of these expressions, different execution paths have to be followed and to this end two instructions are introduced: the *unconditional* jump (`jump`) and the *conditional* jump (`test`). We will first discuss the execution of `if`. Semantic rule 10 specifies that condition E_c is to be evaluated in the current state (s) and then, depending on the outcome, E_1 or E_2 has to be evaluated in the possibly changed state t'_T or t'_F. The virtual machine will first evaluate condition E_c, leaving a reference to the result on top of the stack. The `test` instruction pops the reference off the stack and, if the referred object represents *false*, jumps to the address corresponding to the first instruction of the else-branch E_2. Otherwise, if the condition has evaluated to *true*, execution continues with the next instruction (which is the first instruction of E_1). Figure 3.4 shows the layout of the instructions for `if` and `while`. After executing E_1, an unconditional jump instruction forces the virtual machine to skip over the instructions of E_2.

Since the evaluation of **`while`** E_c **`do`** E **`od`** has the same effect as evaluating the expression
`if` E_c **`then`** E**`;`** **`while`** E_c **`do`** E **`od else nil fi`**, it is easy to see that this can be translated to the code layout in Figure 3.4, where a jump is taken only if condition E_c evaluates to *false*. The virtual machine then jumps to a `push` instruction to put the result of this `while` (which is **`nil`**, see semantic rule 11) on top of the stack. However, if E_c evaluates to *true*, the body (E) is evaluated after which an unconditional jump brings the virtual machine back to the first instruction of E_c for the next iteration.

Methods The execution of method calls is defined by semantic rules 12 (dynamic) and 13 (static). We will first discuss the dynamic method call. Its execution starts with the evaluation of E. The resulting object, β_0, is the object that will execute method m. Since in general, β_0 can not be determined beforehand (at compile time), the corresponding method is unknown and the virtual machine must employ late method binding. Before the method is called, though, the parameters to the method are evaluated from left (E_1) to right (E_n). Then, the corresponding method is searched. The semantic rule uses $\mathcal{M}.\tau_0(\beta_0).m$ to find the method definition corresponding to m, based upon the type of β_0. The virtual machine evaluates E and leaves a reference to its result on top of the stack. As was mentioned in the discussion of the data classes table, each object carries a pointer to its class, so that the virtual machine can easily determine the object's type and use the *method binding table* stored in the corresponding data class to find the method definition of m. The method binding table is simply a list that maps method names to pieces of bytecode representing their bodies. The compiler fills the table such that it implements method inheritance as prescribed by lookup function $\mathcal{M}$ (Section 3.2.3.2).

After the method is found, the virtual machine creates a new local variables context to store the method parameters and its local variables. The method parameters should be bound to the results of $E_1 \dots E_n$ and the local variables of the method should be initialised to *nil*. Instead of creating a new variables context on the heap and copying the references from the stack, the variables context is created on the stack in such a way that the references to the results of $E_1 \dots E_n$ are already stored at the correct positions (Figure 3.5). The local variables in the variables context are initialised by pushing sufficient references to **`nil`** onto the stack.

The virtual machine now changes the executing object to β_0, but stores both a reference to the currently executing object and the address of the instruction following the method call onto the stack to be able to return after the method body has ended. The

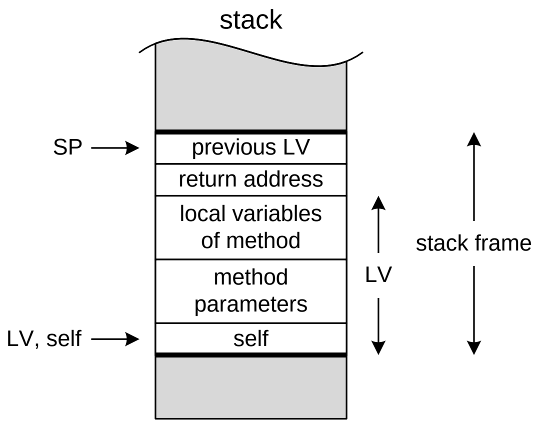


Figure 3.5: A stack frame stores the local variables (LV) of a method and other information such as the return address and the previous stack frame (coincides with the previous local variables environment).

collection of the reference to the executing object, the local variables of the method and the return information is called a *stack frame*. When the virtual machine has adapted the local variables environment and has set the instruction pointer to the first instruction of the method body, execution of the method body can commence. The virtual machine executes a `return` instruction to return to the instruction following the method call when execution of the method body has ended, thereby restoring the executing object and local variables context, while leaving the result of the method on top of the stack.

For the execution of *super method calls*, the static method call instruction is added to the instruction set of the virtual machine. It is similar to the dynamic method call, except that the address of the method body can be fixed at compile time, not requiring the table lookup at run time.

Discarding Results The expressions that were discussed so far invariantly left their result on top of the stack. Sequential composition (semantic rule 9) evaluates E_2 in the state left after executing E_1, but discards result β of E_1. The reference to the result is removed from the stack with a `pop` instruction.

Since assignment to a variable would often lead to a `copy` instruction followed by a `pop` instruction, a small optimisation is introduced by combining both instructions into a special `pop` instruction that pops the reference off the top of the stack and stores it into a variable.

3.3.3 Garbage Collection

POOSL offers no construct to deallocate data objects explicitly. Instead, the virtual machine relies on automatic storage reclamation to remove obsolete objects. Without it, the virtual machine would rapidly deplete its memory.

Garbage collection relieves the designer from the sometimes difficult task to determine the time at which objects can be deleted. Failing to remove objects introduces memory leaks, causing (slow) exhaustion of free memory. Deallocating an object too early can cause even stranger behaviour: when the free memory space it leaves behind is reused by a new object, pointers to the deallocated object give access to the wrong data. The latter kind of errors is particularly hard to debug, as their prolonged effect is usually detected long after the rash deallocation has taken place. A survey on these errors and related ones can be found in [57].

The luxury of relegating objects to oblivion and relying on garbage collection to reclaim them does not come for free. Programs with manual deallocation tend to consume less memory and run slightly faster than programs relying on automatic storage reclamation [59], be it at the increased risk of introducing errors that are difficult to track down, as was discussed above. The time spent on removing these errors hardly ever justifies the slight increase in performance.

During the development of a simulator for POOSL, several garbage collection schemes were implemented. Before discussing these schemes and weighing their pros and cons, a general overview of storage reclamation is given.

Overview The user program is called the *mutator*. It produces and modifies a collection of data objects, which can be viewed as a directed graph. Data objects are represented by nodes, and the outgoing edges correspond to their references to (other) data objects. The *root set* represents the data objects that are directly accessible through variables. Through the references stored in these variables, the mutator can access its data. A data object is *unreachable* if no path exists leading from the root set to the object. The goal of the *(garbage) collector* is to find and reclaim the objects that have become unreachable.

Reference counting Probably the most straightforward approach is to keep track of the amount of references that point to a data object [10]. When this *reference count* reaches zero, the object cannot be reached and by consequence can be safely reclaimed. Before deallocating the object, its references are removed, while properly adjusting the reference counts of the referred objects. Reference counting is without doubt the easiest form of garbage collection to implement, but it has some serious flaws: it fails to collect data trapped in cycles[10] [32] and it can impose a highly irregular computational load. The latter aspect is important in case the mutator is a real-time application. Such a program requires a garbage collector with a predictable, more uniform behaviour to know in advance it will meet its deadlines. The load of a reference counting collector can be unacceptably large when a single reference count reaching zero triggers a cascade of other objects becoming unreachable.

Such a deluge will in general not be harmful for simulation purposes, except that the user may notice an apparent disruption in the normal behaviour of the simulator. Several techniques exist to modify reference counting collectors and remove their irregular behaviour [57]. However, it is a problem that reference counting fails to collect unreachable objects trapped in cycles. We do not want to impose restrictions on the mutator (read: the POOSL model) to ensure that the data graph is acyclic, because this would limit the freedom of the modeler and require actively removing references to break down cycles. Instead, a more sophisticated collector is required.

Mark Sweep The name of this garbage collection algorithm is derived from the two phases it repeatedly passes through. During the first phase, reachable nodes in the data graph are *marked* by a *scanner*, so that during the second phase the unmarked, and therefore unreachable, objects can be *swept* (deallocated) from memory [33].

The entire process is repeated at such a rate that performance loss and memory usage are in balance. The total amount of garbage the mutator produces is independent of the behaviour of the collector. When the collector traverses the data graph frequently, a lot of computational power is wasted, since each collection cycle will, on average, reclaim only little garbage. If, on the other hand, the collection cycles are performed infrequently, a lot of memory will be retained by garbage waiting to be found. At any moment, the amount of garbage found by a Mark-Sweep collector lags behind the quantity that would have been reclaimed by a reference counting collector (in the absence of cyclic data); this translates to an increased memory usage.

[10]For example: the reference count of an object holding a reference to itself can never become less than one, and will therefore not be reclaimed.

Initially, the simulator was fitted with an *incremental* Mark-Sweep collector. Incremental means that the garbage collector does not have to finish its entire cycle before it relinquishes control to the mutator. The implementation of such a collector is more intricate than its non-incremental counterpart because the data graph can change during a single collection cycle. Usually, non-incremental garbage collectors are suitable to (and sufficient for) simulation purposes, since real-time behaviour is not an issue[11], but the prospect of being able to reuse the algorithm for other synthesis targets made the implementation of an *incremental* collector worthwhile[12].

The Mark-Sweep algorithm uses three colours to discern the following three sets of objects: *white* objects, which have not been seen by the scanner (yet); *gray* objects that still need to be scanned; and *black* objects, which have been scanned and are reachable. Initially, all objects are white and scanning commences by making the objects in the root set gray. We will conveniently speak of a *gray reference* whenever we are talking about a reference to a gray object.

The scanner operates on gray objects only. Upon inspecting an object, the scanner colours it black and marks the object's white references gray. This process is repeated until no gray objects are left; any residual white objects went undetected by the scanner and can be reclaimed. The remaining objects are reverted from black to white prior to entering the next marking phase.

During scanning, the following invariant should hold [11]: black objects have no white references. Suppose what happens if the invariant is broken: let a black object hold the only reference to a white object. Since the scanner only inspects gray objects, the white object will never be marked black and will eventually be reclaimed undeservedly. This faulty behaviour can only occur if the data graph is being changed while scanning is in progress, that is, if the mutator and the (incremental) collector run in parallel. The mutator should therefore always check the colour of the involved objects upon changing a reference and if necessary, mark the referred (white) object gray to keep the invariant. Note that marking it black is incorrect as the object itself may provide the only link to other white objects as well.

The implemented incremental Mark-Sweep collector uses an integer *pivot* to discern the four(!) sets it operates on. The fourth set (containing dead objects) springs from the lazy object *recycler* that deallocates objects gradually, even while the next marking phase may have started. To keep track of all objects (especially the unreachable ones), the objects are stored in a circular doubly-linked list: the *chain*. Each object has an integer *state* that determines its colour: if the object's state is below the pivot it is considered dead, if the state equals the pivot it is white, and gray and black objects have a state larger than the pivot. A *stack* stores the gray objects, which the scanner still has to inspect. The marking phase is finished when the stack becomes empty, and at that moment the pivot will be increased. Any white objects left are now considered dead (since their state is just below the new pivot), and black objects

[11]An exception could be in-situ simulation (prototyping) discussed in Chapter 7.

[12]Leaving out of consideration that creating an error-free implementation turned out to be a torment of Tantalus...

have instantly become white. Then, the stack is refilled with the objects in the root set and the next marking phase commences.

First Hybrid Approach As it turned out, garbage collection took considerable time if a model created a lot of data. The culprit was the huge amount of objects produced by the mutator. In order to lower this quantity, a hybrid scheme was devised. A close inspection of the data graph reveals that it contains two parts. One part (say G_1) is formed by objects that can have references to other objects, whereas the other part (say G_2) contains the objects without references. When the graph is cut in such a way that both parts become severed, edges over this cut are always directed from G_1 to G_2, since objects in G_2 have no references. Objects in G_1 can therefore never be accessed by objects in G_2 and the collector can restrict itself to inspecting objects in G_1 for determining garbage. Objects in G_2 can never be part of cycles and can hence be reclaimed by reference counting.

The hybrid collection scheme uses the Mark-Sweep algorithm on nonprimitive data objects (comprising G_1) and applies reference counting to reclaim primitive data objects (of classes *Boolean*, *Integer*, *Real*, *String* and *Nil*). For most simulations, the primitive data objects form the larger part of the data graph. The amount of objects the Mark-Sweep collector has to maintain is therefore significantly reduced, thus diminishing the performance loss induced.

Second Hybrid Approach An alternative hybrid scheme proved to be advantageous for simulation as well. Exploiting the fact that reference counting imposes a lighter computational burden than the Mark-Sweep algorithm, the latter is applied only once in a while for reclaiming the cyclic garbage that reference counting fails to collect. This scheme requires the implementation of the Mark-Sweep collector to be non-incremental: if reference counting decides that an object is garbage, it will deallocate it immediately. However, if it is currently stored in the stack of the Mark-Sweep collector, it has to be removed from this singly-linked list first. Given the average occupation of the stack, this will be quite an expensive operation. Of course, the stack could have been implemented as a doubly-linked list, making removal from the stack cheap enough to render this hybrid scheme attractive compared to the schemes discussed previously. However, the memory overhead for an object without any references will then total to $7 \times 4 = 28$ bytes on a 32-bits machine (pointer to its class, 32-bits reference count, two pointers to build the chain, two pointers to build the stack and 32 bits to hold the object's state). In actual simulations, data objects occupy a significant part of the total amount of memory required at run time, justifying optimisations that will lower their overhead.

The only reason for storing objects in the chain is to keep track of them once they have become unreachable through the root set. Since objects stored in the stack cannot get lost, there is no need to store them in the chain as well. So, when an object becomes gray it can be removed from the chain and be added to the stack, using the very same pointers that are used for storing it in the doubly-linked chain. This reduces the overhead to $5 \times 4 = 20$ bytes. At this point the implementation of the Mark-Sweep collector starts to expose a strong resemblance to another garbage collection scheme which is also known as Baker's Treadmill.

Baker's Treadmill Baker's Treadmill [1] is a large cyclic doubly-linked list containing all registered data objects, just like the chain in the Mark-Sweep implementation. The treadmill identifies four segments and uses pointers to delimit them. The segments in cyclic order contain: objects that have been processed (black objects), objects that should be scanned (gray objects), objects that have not been marked (ecru objects), and finally the objects that can be reclaimed (white).

Initially, the black segment is empty and the gray segment contains the objects in the root set. Notice that the gray segment corresponds to the stack of the Mark-Sweep implementation. The top of the stack is adjacent to the black segment, its bottom borders on the ecru segment. When the scanner inspects the object on top of the stack, it first makes the object black, simply by advancing the top pointer to the next gray object, and checks for ecru references. The corresponding objects are moved from the ecru segment to the top of the gray section for depth-first traversal or, alternatively, to the bottom for breadth-first ordering. Notice that only a single bit is needed to discern ecru references from other references. When the gray section becomes empty, the remaining ecru objects can be added to the list of dead (white) objects, simply by changing the pointer that denotes the border between the ecru segment and the dead objects. Before the next scanning phase can be repeated, the interpretation of the ecru and black objects is interchanged. This eliminates the need to reset the bit stored in the objects in the black section to make them ecru.

The simulator currently employs the hybrid incremental garbage collection scheme as it was suggested for the Mark-Sweep collector, except that it uses Baker's Treadmill to reclaim cyclic garbage.

Object Recycling After identifying which objects are considered garbage, the objects can be deallocated to reclaim the memory they occupy. Instead of deallocating the objects, the collector *recycles* them. The recycled instances are stored in a list kept by their data class, so that when a new object is requested one can be provided immediately, not making use of the expensive allocation process of the memory manager. Recycling leads to an increase in simulation speed of roughly a factor of two or three. Each class stores only a limited amount of objects, balancing between memory usage and performance. The amount of objects that is being recycled is set heuristically. If the list is full, additional (dead) objects will not be recycled, but will be deallocated to preserve memory.

3.3.4 Primitive Methods

Primitive data methods are needed to implement behaviour that cannot be described by POOSL data expressions. The body of a primitive method is implemented in a native language, for instance C++ or Smalltalk. Primitive methods can also be used to control hardware, by having them communicate with a device driver (see Chapter 7). Examples of primitive methods are `random`, `deepcopy`, `shallowcopy` and `==` (identity); see Section 3.2.3.4 for their semantics.

Random The need for a good implementation of a random number generator should not be underestimated. In general, the random number generator offered through library functions have demonstrably nonrandom characteristics [39]. The denotational semantics makes clear that probabilistic characteristics of a model depend on the distribution of numbers produced by the random number generator. This distribution is defined to be uniform and the performance analyses build upon this assumption — if the generator's implementation produces a different distribution, performance metrics will simply be incorrect.

Method `random` of class `RandomNumberGenerator` implements the behaviour defined by $\mathcal{D}.RandomNumberGenerator.Random$. The initial implementation used Lehmer's prime modulus multiplicative linear congruential generator [28] to produce a large sequence of numbers uniformly distributed from $1 \ldots 2^{31} - 1$ that will satisfy almost any statistical test of randomness [39]. Recent progress in research has made available random number generators with much longer sequences. Therefore, Lehmer's generator has been replaced by an implementation of the Mersenne Twister [31], which has a period of $2^{19937} - 1$.

The algorithm is only random in the sense that it simulates a random draw (without replacement) that is statistically indistinguishable from a sequence really drawn at random. The algorithm can be completely deterministic if the seed (first sample) is known. Traces can therefore be completely reproducible, which is ideal for debugging.

Alternatively, the set of produced random numbers can be chosen to be small, thus simplifying an exhaustive analysis of the model.

DeepCopy When a `deepcopy` method is sent to an object, the method body calls a native deepcopy function to traverse the data graph of the object and the objects (indirectly) accessible through its references for constructing a replica of that graph (as prescribed by $\mathcal{D}.Object.DeepCopy$). The algorithm is as follows, starting with N being the data object that executes the `deepcopy` method (Figure 3.6).

Let object N be the argument to the deepcopy function. The function marks the original object N, creates a copy N_{copy} and stores a reference to N_{copy} in N. The deepcopy function is called recursively to copy the references of N. The instance variables of N_{copy} are then set to point to the copied references.

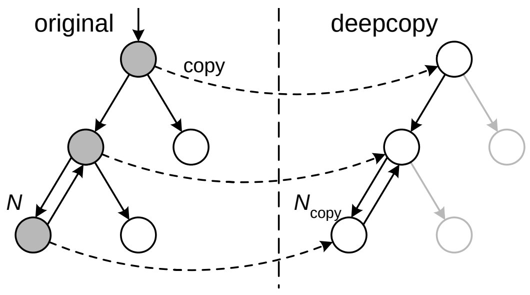


Figure 3.6: The deepcopy algorithm marks the nodes it traverses and stores references to copied objects. After copying the entire structure, these markings and references are removed, and the deepcopy (right) is returned.

The traversal is in depth-first order and each node is marked to prevent objects from being copied twice. Without giving any proof, we mention that the set of marked objects corresponds to the set of objects calculated by FIX $\mathcal{C}_{\beta,\sigma,\tau}$ (where β is the object executing the `deepcopy` method) in Section 3.2.3.4. Each object in the original data graph stores a reference to its copy to provide the same object in case the deepcopying algorithm revisits it via different paths (Figure 3.6).

After the original graph has been copied, the markings are removed, and the copy is returned to the `deepcopy` method.

ShallowCopy Like the `deepcopy` method, `shallowcopy` requires a primitive implementation. Only the object that receives the shallowcopy message is copied, and the references of the copy are set equal to the references in the original.

Identity (==) The identity method returns *true* if and only if the receiving object and the operand to this message are one and the same. Primitive objects are identical if they represent the same mathematical object, and nonprimitive objects are identical if they are stored at the same memory address.

3.4 Summary

The first part of this chapter has specified the new abstract syntax for the POOSL data layer and has redefined its formal meaning in terms of a denotational semantics. This new semantics adds probabilistic features to support performance modelling [55]. Adding probabilistic information at the data layer instead of the process layer as in [54] solves the problem identified in [52] of having to specify rather arbitrary probabilities for operators that model nondeterministic behaviour. In the denotational semantics developed here, the probabilistic behaviour springs from method *Random* of class *RandomNumberGenerator*, which returns a uniformly distributed integer value in $\{1, \ldots, \varrho\}$ — composite expressions then combine these probabilities.

Local and global recursion complicate the construction of a compositional definition of `while` and nonprimitive data methods. This problem is overcome by introducing a functional whose least fixed point serves as the compositional definition of the meaning of `while`. Because POOSL supports dynamic method calls, the behaviour of nonprimitive methods cannot be unfolded (as is often done in literature) and therefore an approach similar to the one for `while` is employed to obtain a compositional definition.

To be able to actually create executable models, a concrete syntax for POOSL is defined (Appendix B) that resolves the ambiguities present in the abstract syntax of Section 3.2.1. Using the formal semantics in Section 3.2.3 as a prescription, a platform for executing expressions is developed: the virtual machine (Section 3.3.2). The bytecode for this machine is provided by a compiler that also checks if the compiled POOSL specification (model) satisfies the context conditions stated in Section 3.2.2. POOSL offers no manual way to perform memory management, but rather relies on a garbage collector to reclaim obsolete data objects instead. The simulator employs a hybrid incremental garbage collection scheme based on reference counting complemented by Baker's Treadmill to cleanup cyclic data.

Several collection schemes have been implemented and evaluated for simulation purposes (Section 3.3.3). Reference counting cannot remove unreachable cyclic data structures, and is therefore inadequate. Mark-Sweep collection can reclaim any data structure, but induces a performance loss in simulation speed proportional to the size of the data graph it traverses and increases the memory requirements significantly. A hybrid approach identifies that primitive data cannot produce cycles, and uses reference counting to reclaim this data, while the remaining (significantly reduced) set of nonprimitive data is managed by the Mark-Sweep collector. A second hybrid approach uses reference counting on all data, but uses an incremental Baker's Treadmill to collect the garbage reference counting fails to reclaim. The simulator currently employs the latter hybrid scheme.

Chapter 4

POOSL Process Layer

4.1 Overview

After describing the abstract syntax of POOSL process statements (or just *statements*), this chapter discusses the structural operational semantics that defines the meaning of the process layer. The semantics also describes the probabilistic features of process statements, using the probabilistic information extracted from the data layer. Finally, execution of process statements by means of *execution trees* is explained.

4.2 Specification

4.2.1 Abstract Syntax

The syntactic categories used for the discussion of the syntax and semantics of the POOSL process layer have already been introduced in Section 3.2.1. Some of the definitions of that section are extended here.

The definition of *ClassDef* is augmented by process class definitions:

$$\begin{array}{llll} CD & = & \texttt{\textbf{process class}} & C(y_1,\ldots,y_r) \\ & & [\,\texttt{\textbf{extends}} & C_{super}\,] \\ & & \texttt{\textbf{port interface}} & \underline{p_1}\cdots\underline{p_g} \\ & & \texttt{\textbf{message interface}} & \overline{ms_1}\cdots\overline{ms_h} \\ & & \texttt{\textbf{instance variables}} & x_1\cdots x_n \\ & & \texttt{\textbf{initial method call}} & m_{C'}(E_1,\ldots,E_q)(\,) \\ & & \texttt{\textbf{instance methods}} & MD_1^p\cdots MD_k^p. \end{array}$$

A process class definition starts by defining the name C of the class. Some of the instance variables ($x_1\ldots x_n$ or inherited variables) serve as instantiation parameters $y_1\ldots y_r$.[1] Upon initialisation these variables are set to the results of parametric expressions provided by cluster classes; this is discussed in Chapter 5. Other variables are set to *nil*. Optionally, a process class can inherit from another process class

[1]In fact $\{y_1,\ldots,y_r\}\subseteq\mathcal{V}(C)$, where $\mathcal{V}(C)$ is the set of all (inherited) instance variables (see Section 4.2.3).

(C_{super}). Like data objects, processes have instance methods. The initial method call specifies which method is to be called first after the process has been created. Like the static data method call, $m_{C'}(E_1, \ldots, E_q)(\ )$ refers to method m, defined in class C'. In practice, the compiler restricts method calls to the set of (inherited) methods defined in the process class of the caller (see Section 4.2.3).

Each instance of the process class will be equipped with the communication ports listed in the port interface. Ports are merely denoted by (non-empty) strings: p is a typical element of the set of ports $\mathit{Ports} = \mathbb{S} \setminus \{\texttt{""}\}$. The message interface defines the *signatures* of the messages that travel across the ports. Variable ms is a typical element of *MsgSignatures* that is defined by:

$$\begin{array}{rcl} ms & = & \underline{p}\,\texttt{!}\,m(\underline{n}) \\ & | & \underline{p}\texttt{?}m(\underline{n}). \end{array}$$

Each signature defines the name p of the port that accepts sending or receiving messages called m with n parameters. The architecture layer can impose additional restrictions on communication before processes are interconnected by channels — these details will be discussed in Chapter 5.

The instance methods of a process class, *process methods*, are elements of $\mathit{MethDef}^p$ (the annotation p discerns $\mathit{MethDef}^p$ from *MethDef* defined in Section 3.2.1):

$$\begin{array}{rcl} MD^p & = & m(u_1, \ldots, u_n)(w_1, \ldots, w_k) \\ & & |z_1 \cdots z_m| \\ & & S^b. \end{array}$$

Each method definition starts with the name of the method (m) followed by input parameters $u_1 \ldots u_n$, output parameters $w_1 \ldots w_k$ and local variables $z_1 \ldots z_m$. The method body S^b is an element of Stat^b, the set of basic statements:

$$\begin{array}{rcll} S^b & = & E & \text{expression} \\ & | & \texttt{skip} & \text{skip} \\ & | & \texttt{delay}\ E & \text{delay} \\ & | & E_p\,\texttt{!}\,m(E_1, \ldots, E_n)\{E\} & \text{message send} \\ & | & E_p\texttt{?}m(v_1, \ldots, v_n|E_{rc})\{E\} & \text{message receive} \\ & | & m_C(E_1, \ldots, E_n)(v_1, \ldots, v_k) & \text{method call} \\ & | & [E]S^b & \text{guarded command} \\ & | & \texttt{if}\ E\ \texttt{then}\ S_1^b\ \texttt{else}\ S_2^b\ \texttt{fi} & \text{if} \\ & | & \texttt{while}\ E\ \texttt{do}\ S^b\ \texttt{od} & \text{while} \\ & | & S_1^b\texttt{;}S_2^b & \text{sequential composition} \\ & | & \texttt{par}\ S_1^b\ \texttt{and}\ S_2^b\ \texttt{rap} & \text{parallel composition} \\ & | & \texttt{sel}\ S_1^b\ \texttt{or}\ S_2^b\ \texttt{les} & \text{select} \\ & | & \texttt{abort}\ S_1^b\ \texttt{with}\ S_2^b & \text{abort} \\ & | & \texttt{interrupt}\ S_1^b\ \texttt{with}\ S_2^b & \text{interrupt.} \end{array}$$

Most expressions $E \in \mathit{Exp}$ are also valid statements[2]. The `skip` models an execution step that does not change the variables state of the model; it is syntactic sugar for

[2]The context conditions in Section 4.2.2 elaborate on the forbidden constructs.

`nil` and is for example a convenient placeholder for behaviour that will be filled out later when refining the model.

Passage of time is modelled by a `delay`. The amount of time to pass is given by the accompanying expression, which should evaluate to a *Real*[3] or an *Integer*.

Next, two statements for interprocess communication are defined. Both constructs specify the communication port the message should be sent to or received from. Only if the connected channel has a matching (complementary) message, communication can take place. This *rendez-vous* message exchange is based on the *synchronous pair-wise message passing* mechanism of CCS [34, 35].

The *message send* statement $E_p\,!\,m(E_1, \ldots, E_n)\{E\}$ models a message m being sent to the port whose name corresponds to the string resulting from evaluating E_p. When a rendez-vous takes place, the parameters $E_1 \ldots E_n$ are evaluated from left to right and are deepcopied[4] before their results are bound to the parameters $v_1 \ldots v_n$ of the complementary *message receive* statement $E'_p?m(v_1, \ldots, v_n|E_{rc})\{E\}$. After binding the data, the *reception condition* E_{rc} of the receive statement is evaluated and only if the result is *true*, the communication can actually take place. If the reception condition evaluates to *false*, communication cannot take place, and the model's state is reverted to the state prior to the rendez-vous. Communication statements can optionally be equipped with an expression (E) that is evaluated immediately after communication has taken place.

The following statement is the (process) *method call* with input expressions $E_1 \ldots E_n$ and output parameters $v_1 \ldots v_k$. When the corresponding (process) method m is called in class C, the input expressions are evaluated from left to right and the results are bound to the method's input parameters. After the method's body has been executed, the output parameters are bound to variables $v_1 \ldots v_k$ as specified in the method call. Process methods without output parameters allow for tail-recursion; this facilitates the modelling of repetitive behaviour.

The guarded command $[E]S^b$ limits the execution of S^b. Only if condition E evaluates to *true* is S^b allowed to execute, for any other value the guarded command blocks.

Conditional execution of statements is offered by the `if`. If condition E evaluates to *true*, S_2^b is discarded, leaving S_1^b for execution. If E evaluates to *false* instead, S_1^b is discarded, leaving S_2^b for execution.

The `while` repeatedly executes statement S as long as condition E evaluates to *true*.

Sequential composition ($S_1^b; S_2^b$) specifies that the execution of S_1^b must be finished before S_2^b is being executed. Parallel composition, on the other hand, allows for any interleaving of the execution steps of *concurrent activities* S_1^b and S_2^b. The `select` allows two alternative statements to compete for execution. Only the first one that actually starts executing may proceed, the other statement is discarded.

[3]We use "a C" as a shorthand for "an instance of class C".

[4]Processes are not allowed to share data objects and therefore the result is deepcopied. The deepcopies are used by the receiver and are unknown to the transmitter.

An executing statement can be terminated prematurely if it is enclosed by an `abort`. When S_2^b performs an execution step, S_1^b is aborted. A similar construct is the `interrupt`. However, when S_2^b performs a step, S_1^b is merely suspended, and continues where it left off after S_2^b has terminated. S_1^b can be interrupted by S_2^b repeatedly.

Several semantic rules require additional constructs for representing intermediate execution states. For instance, a method call executing the body of the corresponding method must be remembered, so that the output parameters can be bound upon termination of the body. To this end, an extended set of statements *Stat* is defined, which contains constructs similar to the basic constructs of *Stat*^{b} and a few more:

$$
\begin{array}{rcl}
S & = & E \\
 & | & \texttt{\textbf{skip}} \\
 & | & \texttt{\textbf{delay}}\ E \\
 & | & E_p\,!\,m(E_1,\ldots,E_n)\{E\} \\
 & | & E_p?m(v_1,\ldots,v_n|E_{rc})\{E\} \\
 & | & m_C(E_1,\ldots,E_n)(v_1,\ldots,v_k) \\
 & | & [E]S \\
 & | & \texttt{\textbf{if}}\ E\ \texttt{\textbf{then}}\ S_1\ \texttt{\textbf{else}}\ S_2\ \texttt{\textbf{fi}} \\
 & | & \texttt{\textbf{while}}\ E\ \texttt{\textbf{do}}\ S\ \texttt{\textbf{od}} \\
 & | & S_1\,\texttt{;}\,S_2 \\
 & | & \texttt{\textbf{par}}\ S_1\ \texttt{\textbf{and}}\ S_2\ \texttt{\textbf{rap}} \\
 & | & \texttt{\textbf{sel}}\ S_1\ \texttt{\textbf{or}}\ S_2\ \texttt{\textbf{les}} \\
 & | & \texttt{\textbf{abort}}\ S_1\ \texttt{\textbf{with}}\ S_2 \\
 & | & \texttt{\textbf{interrupt}}\ S_1\ \texttt{\textbf{with}}\ S_2 \\
 & | & \texttt{\textbf{delay}}\ \widetilde{t} \\
 & | & \underline{\widetilde{p}}\,!\,m(E_1,\ldots,E_n)\{E\} \\
 & | & \underline{\widetilde{p}}?m(v_1,\ldots,v_n|E_{rc})\{E\} \\
 & | & m_C(\,)(w_1,\ldots,w_k)[S]^{\psi} \\
 & | & [S]^{\psi} \\
 & | & \texttt{\textbf{interrupt}}\ S_1\ \texttt{\textbf{with}}\ S_2, S_3 \\
 & | & S_1\ \texttt{\textbf{interrupted by}}\ S_2, S_3 \\
 & | & \surd .
\end{array}
$$

A `delay` that expires in t units of time is denoted by **`delay`** $\widetilde{t}$. The tilde distinguishes this statement from **`delay`** t, which denotes a `delay` whose expression (the variable t) has not yet been evaluated.

The statement $\underline{\widetilde{p}}\,!\,m(E_1,\ldots,E_n)\{E\}$ denotes a send statement whose port identifier has evaluated to p and is kept fixed at that value (suggested by the tilde). Likewise, $\underline{\widetilde{p}}?m(v_1,\ldots,v_n|E_{rc})\{E\}$ denotes a receive statement that has been fixed and is ready to communicate.

A method call, currently executing the remainder S of the corresponding method body in local variables context ψ, is denoted by $m_C(\,)(w_1,\ldots,w_k)[S]^{\psi}$. It stores the parameters $w_1 \ldots w_k$ for binding when the method terminates. For an executing method call without return parameters, only the body and the local variables context it executes in are preserved, yielding $[S]^{\psi}$.

For restoring a finished `interrupt` to its original state, **`interrupt`** S_1 **`with`** S_2, S_3 is introduced. S_1 is the statement that can still be interrupted by S_2, where S_2 originally emanated from S_3. An executing `interrupt` (with suspended S_1) is denoted by S_1 **`interrupted by`** S_2, S_3. When the currently executing interrupt S_2 has ended, it will be restored to S_3.

Finally, $\surd$ (pronounced *tick*) denotes the terminated statement.

4.2.2 Context Conditions

Besides the conditions stated in Section 3.2.2, additional context conditions must be fulfilled by a system specification:

1. The inheritance graph of process classes is free of cycles.
2. All (inherited) instance variables and instantiation parameters of a process class definition have different names.
3. All (inherited) methods of a process class are discernable by their name[5].
4. All parameter and local variable names in a process method definition are different.
5. Every variable used in a process method body is either an inherited instance variable of the corresponding class, or, a method parameter or a local variable of that method.
6. The use of **`self`** within a process method is forbidden.
7. The actual parameters to an initial method call do not contain any local variables nor do they use **`self`**.
8. Every method call invokes an existing method definition.
9. Parameters of message-send statements and expressions of `guards` are side-effect free and deterministic.
10. All (inherited) variables and ports in a process class have different names.

4.2.3 Structural Operational Semantics

G.D. Plotkin introduced the technique of *structural operational semantics* in [40]. Section 2.4 has provided preliminary knowledge that will now be tailored to POOSL.

The meaning of statements is specified by the timed probabilistic labelled transition system $(Conf, Act, \xrightarrow[Act]{}, Time, \xrightarrow[Time^+]{})$. The countable set of configurations $(Conf)$ is ranged over by c. Configurations have the form[6] $\left([S]_C^{\psi}, \sigma, \tau\right)$. The first component, $[S]_C^{\psi}$, represents the statement $S \in Stat$ that is to be executed in the context of class C and local variables context ψ. The global variables context is stored in σ, while type information can be found in τ.

[5]In the concrete syntax this is relaxed to a unique combination of name and parameter count.

[6]Actually, the set of configurations is larger, and is defined completely in Section 5.2.3.

Act is the set of *actions*, ranged over by a. Three different kinds of actions can be identified:

- the *internal action* τ (also called *silent action*), denoting an internal computation that is unobservable for the system's environment;
- *communication actions* (or *synchronisation actions*) of the following forms: the (message) *send action* $p\,!\,m[data]$, denoting the system's willingness to send a message m with data to port p; and the (message) *receive action* $p?m[data]$, denoting that the system is susceptible to receiving a message m with data from port p;
- the *fix action* f, denoting an internal action that does not affect its context[7].

The subtle difference between the internal action τ and fix action f will be explained later. The set of communication actions will be called $\mathcal{L}$. When the exact signature of a communication action is not important, it is denoted by ℓ. The complement function $\overline{\cdot} \in Act \rightarrow Act$ is defined in such a way that $\overline{p\,!\,m[data]} = p?m[data]$ and such that it does not affect the internal action and fix action ($\overline{\tau} = \tau$ and $\overline{f} = f$). The complement function is extended to the whole of $\mathcal{L}$, so that $\overline{\overline{\ell}} = \ell$. The time domain *Time*, ranged over by t, should satisfy the definition in Section 2.4 and can either be discrete or dense. Each process is equipped with a special instance variable called *currentTime* that equals the model time.

To describe the probabilistic effect of executing a statement, a set of substochastic probability functions is defined (comparable with Definition 20 for expressions). These functions bind probabilities to configurations.

Definition 41 (substochastic probability function)
Let $\mathcal{P}(Conf) = \left\{ \boldsymbol{\pi} \in Conf \rightarrow \mathcal{R} \;\middle|\; \sum_{c \in Conf} \boldsymbol{\pi}.c \leq 1 \right\}$.

The sum of probabilities from a particular configuration to any other configuration may be less than one because of nonterminating loops. For all $\boldsymbol{\pi} \in \mathcal{P}(Conf)$, $\sum_{c \in Conf} \boldsymbol{\pi}.c$ is a well-defined series because its terms $\boldsymbol{\pi}.c$ are non-negative, the sequence of its partial sums is bounded (by one) and *Conf* is countable.

The semantics uses two labelled transition relations to describe the effect of statements: one describes *action steps*, the other describes the passage of time (*delay steps*).

Definition 42 (action transitions)
We let the action transitions $\cdot \xrightarrow[Act]{\cdot} \cdot \subseteq Conf \times Act \times \mathcal{P}(Conf)$ *be defined by the semantic rules 14 through 37 in Appendix A.2.*

Definition 43 (time transitions)
We let the time transitions $\cdot \xrightarrow[Time^+]{\cdot} \cdot \subseteq Conf \times Time^+ \times Conf$ *be defined by the semantic rules 38 through 52 in Appendix A.2.*

[7] By context, we mean the behavioural description of the executing process, not its variables state. This is explained in full detail in Section 4.2.3.1.

Table 4.1: The termination function $\surd \in \mathit{Stat} \rightarrow \mathit{Stat}$.

S	$\surd(S)$	
E	E	
$\texttt{skip}$	$\texttt{skip}$	
$\texttt{delay}\ E$	$\texttt{delay}\ E$	
$E_p!m(E_1,\ldots,E_n)\{E\}$	$E_p!m(E_1,\ldots,E_n)\{E\}$	
$E_p?m(v_1,\ldots,v_n \mid E_{rc})\{E\}$	$E_p?m(v_1,\ldots,v_n \mid E_{rc})\{E\}$	
$m_C(E_1,\ldots,E_n)(v_1,\ldots,v_k)$	$m_C(E_1,\ldots,E_n)(v_1,\ldots,v_k)$	
$[E]S$	$\surd$	if $\surd(S) = \surd$
	$[E]\surd(S)$	otherwise
$\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}$	$\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}$	
$\texttt{while}\ E\ \texttt{do}\ S\ \texttt{od}$	$\texttt{while}\ E\ \texttt{do}\ S\ \texttt{od}$	
$S_1;S_2$	S_2	if $\surd(S_1) = \surd$
	$\surd(S_1);S_2$	otherwise
$\texttt{par}\ S_1\ \texttt{and}\ S_2\ \texttt{rap}$	$\surd(S_2)$	if $\surd(S_1) = \surd$
	$\surd(S_1)$	if $\surd(S_2) = \surd$
	$\texttt{par}\ \surd(S_1)\ \texttt{and}\ \surd(S_2)\ \texttt{rap}$	otherwise
$\texttt{sel}\ S_1\ \texttt{or}\ S_2\ \texttt{les}$	$\surd$	if $\surd(S_1) = \surd$ or $\surd(S_2) = \surd$
	$\texttt{sel}\ \surd(S_1)\ \texttt{or}\ \surd(S_2)\ \texttt{les}$	otherwise
$\texttt{abort}\ S_1\ \texttt{with}\ S_2$	$\surd$	if $\surd(S_1) = \surd$ or $\surd(S_2) = \surd$
	$\texttt{abort}\ \surd(S_1)\ \texttt{with}\ \surd(S_2)$	otherwise
$\texttt{interrupt}\ S_1\ \texttt{with}\ S_2$	$\texttt{interrupt}\ S_1\ \texttt{with}\ S_2$	
$\texttt{interrupt}\ S_1\ \texttt{with}\ S_2, S_3$	$\surd$	if $\surd(S_1) = \surd$
	$\texttt{interrupt}\ \surd(S_1)\ \texttt{with}\ S_3, S_3$	if $\surd(S_1) \neq \surd$ and $\surd(S_2) = \surd$
	$\texttt{interrupt}\ \surd(S_1)\ \texttt{with}\ \surd(S_2), S_3$	otherwise
$S_1\ \texttt{interrupted by}\ S_2, S_3$	$\texttt{interrupt}\ S_1\ \texttt{with}\ S_3, S_3$	if $\surd(S_2) = \surd$
	$S_1\ \texttt{interrupted by}\ \surd(S_2), S_3$	otherwise
$m_C(\,)(w_1,\ldots,w_k)[S]^{\psi}$	$m_C(\,)(w_1,\ldots,w_k)[\surd(S)]^{\psi}$	
$[S]^{\psi}$	$\surd$	if $\surd(S) = \surd$
	$[\surd(S)]^{\psi}$	otherwise
$\underline{\widetilde{p}}!m(E_1,\ldots,E_n)\{E\}$	$\underline{\widetilde{p}}!m(E_1,\ldots,E_n)\{E\}$	
$\underline{\widetilde{p}}?m(v_1,\ldots,v_n \mid E_{rc})\{E\}$	$\underline{\widetilde{p}}?m(v_1,\ldots,v_n \mid E_{rc})\{E\}$	
$\texttt{delay}\ \widetilde{t}$	$\surd$	if $t = 0$
	$\texttt{delay}\ \widetilde{t}$	otherwise
$\surd$	$\surd$	

These transition relations are defined by the inference rules 14 through 52 listed in Appendix A.2. To simplify these semantic rules, a termination function $\surd \in \mathit{Stat} \rightarrow \mathit{Stat}$ is defined by Table 4.1 in accordance with [18]. Without such a function, some of the rules would have to discriminate between nonterminated statements and terminated statements. To give an example: without a termination function the semantic rule for sequential composition would look like:

Sequential composition

$$\frac{\left([S_1]^{\psi}_{C}, \sigma, \tau\right) \xrightarrow{a} \boldsymbol{\pi}'}{\left([S_1\,\mathbf{;}\,S_2]^{\psi}_{C}, \sigma, \tau\right) \xrightarrow{a} \boldsymbol{\pi}} \text{ SEQ}$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\left([S_1']^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{for } c = \left([S_1'\,\mathbf{;}\,S_2]^{\psi'}_{C}, \sigma', \tau'\right) \text{ if } S_1' \neq \surd \\ \boldsymbol{\pi}'.\left([\surd]^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{for } c = \left([S_2]^{\psi'}_{C}, \sigma', \tau'\right) \\ 0 \qquad \text{otherwise.} \end{cases}$$

Thus, when the first statement terminates, the sequential composition should reduce to S_2. With the termination function, there is no separate rule for the second case ($S_1' = \surd$), because $\surd(\surd\,\mathbf{;}\,S_2) = S_2$. The semantic rule now reduces to:

Sequential composition

$$\frac{\left([S_1]^{\psi}_{C}, \sigma, \tau\right) \xrightarrow{a} \boldsymbol{\pi}'}{\left([S_1\,\mathbf{;}\,S_2]^{\psi}_{C}, \sigma, \tau\right) \xrightarrow{a} \boldsymbol{\pi}} \text{ SEQ}$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\left([S_1']^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{for } c = \left([\surd(S_1'\,\mathbf{;}\,S_2)]^{\psi'}_{C}, \sigma', \tau'\right) \\ 0 \qquad \text{otherwise.} \end{cases}$$

Function $\mathcal{V}$ was introduced in Section 3.2.3.2 to compute the collection of (inherited) instance variables of a data class C. Its definition is extended here to cope with process classes as well:

$$\mathcal{V}(C) = \begin{cases} \{x_1, \ldots, x_n\} & \text{if } \mathit{CDList} \equiv \mathit{CD}_1 \cdots \mathit{CD}_i \cdots \mathit{CD}_n \text{ where} \\ & \mathit{CD}_i \equiv \begin{array}{ll} \texttt{process class} & C(y_1, \ldots, y_r) \\ \texttt{port interface} & \underline{p_1 \cdots p_g} \\ \texttt{message interface} & \overline{ms_1 \cdots ms_h} \\ \texttt{instance variables} & x_1 \cdots x_n \\ \texttt{initial method call} & m_{C'}(E_1, \ldots, E_q)(\,) \\ \texttt{instance methods} & \mathit{MD}^p_1 \cdots \mathit{MD}^p_k \end{array} \\ \{x_1, \ldots, x_n\} & \text{if } \mathit{CDList} \equiv \mathit{CD}_1 \cdots \mathit{CD}_i \cdots \mathit{CD}_n \text{ where} \\ \quad \cup\, \mathcal{V}(C_{super}) & \mathit{CD}_i \equiv \begin{array}{ll} \texttt{process class} & C(y_1, \ldots, y_r) \\ \texttt{extends} & C_{super} \\ \texttt{port interface} & \underline{p_1 \cdots p_g} \\ \texttt{message interface} & \overline{ms_1 \cdots ms_h} \\ \texttt{instance variables} & x_1 \cdots x_n \\ \texttt{initial method call} & m_{C'}(E_1, \ldots, E_q)(\,) \\ \texttt{instance methods} & \mathit{MD}^p_1 \cdots \mathit{MD}^p_k \end{array} \\ \varnothing & \text{otherwise.} \end{cases}$$

Function $\mathcal{M}_s$ for looking up method definitions (see Section 3.2.3.2) is completed by extending it for process methods:

$$
\mathcal{M}_s.C.m \equiv \begin{cases}
MD^p_j & \begin{array}[t]{l}
\text{if } \mathit{CDList} \equiv CD_1 \cdots CD_i \cdots CD_n \text{ and} \\
CD_i \equiv \begin{array}[t]{ll}
\;\;\texttt{\textbf{process class}} & C(y_1, \ldots, y_r) \\
[\,\texttt{\textbf{extends}} & C_{super}\,] \\
\;\;\texttt{\textbf{port interface}} & \underline{p_1} \cdots \underline{p_g} \\
\;\;\texttt{\textbf{message interface}} & \overline{ms_1} \cdots \overline{ms_h} \\
\;\;\texttt{\textbf{instance variables}} & x_1 \cdots x_n \\
\;\;\texttt{\textbf{initial method call}} & m_{C'}(E_1, \ldots, E_q)(\,) \\
\;\;\texttt{\textbf{instance methods}} & MD^p_1 \cdots MD^p_j \cdots MD^p_k
\end{array} \\
\text{and } MD^p_j \equiv \begin{array}[t]{l}
m(u_1, \ldots, u_n)(w_1, \ldots, w_k) \\
|z_1 \cdots z_m| \\
S^b
\end{array}
\end{array} \\
\underline{\text{undef}} & \text{otherwise.}
\end{cases}
$$

Although process method calls are static and $\mathcal{M}_s$ provides sufficient knowledge to give their semantics, we will also extend recursive lookup function $\mathcal{M}$ for process methods:

$$
\mathcal{M}.C.m \equiv \begin{cases}
\mathcal{M}_s.C.m & \text{if } \mathcal{M}_s.C.m \neq \underline{\text{undef}} \\
\mathcal{M}.C_{super}.m & \begin{array}[t]{l}
\text{if } \mathcal{M}_s.C.m = \underline{\text{undef}} \text{ and} \\
\mathit{CDList} \equiv CD_1 \cdots CD_i \cdots CD_n \text{ and} \\
CD_i \equiv \begin{array}[t]{ll}
\texttt{\textbf{process class}} & C(y_1, \ldots, y_r) \\
\texttt{\textbf{extends}} & C_{super} \\
\texttt{\textbf{port interface}} & \underline{p_1} \cdots \underline{p_g} \\
\texttt{\textbf{message interface}} & \overline{ms_1} \cdots \overline{ms_h} \\
\texttt{\textbf{instance variables}} & x_1 \cdots x_n \\
\texttt{\textbf{initial method call}} & m_{C'}(E_1, \ldots, E_q)(\,) \\
\texttt{\textbf{instance methods}} & MD^p_1 \cdots MD^p_k
\end{array}
\end{array} \\
\underline{\text{undef}} & \text{otherwise.}
\end{cases}
$$

We define $\upharpoonleft\!\mathcal{M}.C.m = \mathcal{M}.C_{super}.m$ if C_{super} is the superclass of C, and $\upharpoonleft\!\mathcal{M}.C.m = \underline{\text{undef}}$ otherwise. Function $\upharpoonleft\!\mathcal{M}$ is like $\mathcal{M}$, except that the recursive search starts at the superclass of the caller. It has been introduced to impose restrictions on the use of method calls offered by implementations of POOSL.

To support inheritance, the static method call was introduced. The semantics allows a process or data class to call any of its inherited methods by explicitly stating the name of the class and method; in practice, this is too liberal[8]. Implementations are restricted to offer process method calls in the form of $m(E_1, \ldots, E_n)(v_1, \ldots, v_k)$ and $\hat{\ }m(E_1, \ldots, E_n)(v_1, \ldots, v_k)$. In the context of class C, these calls must use definitions $\mathcal{M}.C.m$ and $\upharpoonleft\!\mathcal{M}.C.m$ respectively. The compiler can replace the calls by equivalent static method calls because at that time the inheritance tree is known. For data method calls, $E\ m(E_1, \ldots, E_n)$ and `self` $\hat{\ }m(E_1, \ldots, E_n)$ in the concrete syntax represent the dynamic method call and static method call respectively, and the corresponding definitions in the context of class C are given by $\mathcal{M}.C.m$ and $\upharpoonleft\!\mathcal{M}.C.m$.

[8]During the development of a class that inherits from some superclass C_{super}, the programmer is not allowed to rely on the implementation of C_{super}, only on the interface formed by the methods that C_{super} offers.

4.2.3.1 Discussion

This section discusses some of the more interesting rules in Appendix A.2 of POOSL's process layer.

Expressions Axiom 14 (EXP) states the effect of evaluating *statement* E in the current context formed by σ, ψ and τ. The rule stores the probabilistic effect of evaluating *expression* E in probability function $\boldsymbol{\pi}$. From the given start configuration $\big([E]^{\psi}_{C}, \sigma, \tau\big)$, start state s is computed in which expression E is to be evaluated. This yields terminal state s' with probability $[\![E]\!].s.s'$. The terminal state is defined to correspond to terminal configuration $\big([\surd]^{\psi'}_{C}, \sigma', \tau'\big)$. Other configurations cannot be reached; for these configurations the probability equals zero.

Method call The behaviour of the method call is defined by rules 29 (MC$_1$), 30 (MC$_2$), 31 (MC$_3$), 32 (MC$_4$), 41 (MC$_5$) and 42 (MC$_6$). Rule MC$_1$ defines the first phase of executing a method call. A new local variables environment ψ' is created in which the local variables $z_1 \dots z_m$ and $w_1 \dots w_k$ are set to *nil*. The actual parameters $E_1 \dots E_n$ are evaluated from left to right and the results are bound to the formal parameters $u_1 \dots u_n$. Method body S runs in the new local variables context ψ', denoted by $[S]^{\psi'}$. The end configuration for methods with output parameters $\big([\, m_{C'}(\,)(w'_1, \dots, w'_k)\, [S]^{\psi'}]^{\psi_n}_{C}, \sigma_n, \tau_n\big)$ remembers the variables $w'_1 \dots w'_k$ in which the results of the method must be stored after the body terminates. If the method has no return parameters, the method call can be dropped, yielding the terminal configuration $\big([\, [S]^{\psi'}]^{\psi_n}_{C}, \sigma_n, \tau_n\big)$.

The second phase of executing the method call is executing the body. Rules MC$_5$ and MC$_6$ allow the body to perform time transitions in the context of an executing method call (with or without output parameters), whereas rules MC$_2$ and MC$_3$ allow the body to perform actions. To support tail recursion, rule MC$_3$ uses function $\mathbb{V} \in \mathit{Stat} \longrightarrow \mathit{Stat}$ for cleaning up redundant local variables contexts, which is defined by:

$$\mathbb{V}(S) = \begin{cases} [S']^{\psi'} & \text{if } S = [\, [S']^{\psi'}]^{\psi} \\ S & \text{otherwise.} \end{cases}$$

The third and final phase is described by rule MC$_4$. When the body of the called method has terminated in local variables context ψ'', the formal output parameters $w_1 \dots w_k$ are bound to the actual output parameters $w'_1 \dots w'_k$ (either global variables or local variables of context ψ in which the method call itself runs). The yielded end configuration cleans up the terminated body and the method call.

Rule MC$_1$ defines the expansion of the method call to the corresponding method body as a fix action. These transitions are invisible to the statement context as long as they do not yield $\surd$. If the transition were a silent action, this would not necessarily be the case. This subtle yet important difference, is clarified by examining the `select`.

Select The `select` is defined by three rules: 21 (SEL_1), 22 (SEL_2) and 45 (SEL_3). Rules SEL_1 and SEL_2 are one another's counterparts because the `select` is symmetrical. Rule SEL_1 discards the second branch (S_2) when the first branch performs an action, and continues to behave as the remainder of the first branch, S_1'. The rule discriminates between the fix action and other actions; the following example motivates this distinction.

Consider the statement **sel** $m(\,)(\,)$ **or** S **les**. During the first phase of execution, the method call uses a fix action to expand. If SEL_1 would not discern fix actions from others, the `select` would immediately choose for the first branch, discarding the second. The behaviour is clearly different if the method call is replaced by the corresponding method body *syntactically.* Then, the `select` founds its choice on the first action performed by the method's body.

We would like to see the method call as an abstraction of the corresponding method body. To support this view, the fix action has been introduced. This allows the `select` to treat fix actions differently from other actions. For fix actions, rule SEL_1 yields a terminal configuration that leaves the second branch intact. In the example above, the method call expands but the `select` leaves both alternatives open — just as it would when the method call were replaced syntactically.

Fix action The fix action has been introduced to solve the following problems emerging from adding time and data to the language. [17] already signified the problem that evaluating a `delay`'s expression in the context of a `select` unwantedly leads to discarding other branches. In Table 4.2.**A**, evaluation of `z+2` (line 2) leads to discarding the branch in line 4. A similar problem occurs if the input parameter of method `n` is bound (line 4) — this will lead to discarding line 2. The source of the problem is that the expression evaluation is performed with an internal action (τ) that triggers the `select`. Notice that the input parameter `z` must be bound, when the `delay` in method `n` depending on `z` via `x` is evaluated. Because the `select` cannot discern the internal action originating from the `delay`'s expression from other internal actions, it is unable to yield a different terminal state. Both problems can be solved with a different kind of internal action: the fix action.

The fix action is an internal action like τ, except that it is "invisible" to its statement context. More precise: compound statements do not react to changes due to fix actions as long as the changed constituent does not tick. So, in the example of Table 4.2.**A**, the `delay` in line 2 can evaluate and fix the expression `z+2` and the `select` will keep both branches. The same applies to binding input and output parameters of method calls.

Originally, method output parameters were bound at the same moment the method body terminated. Because the new semantics introduces parallelism within processes, binding at termination is no longer always valid. In combination with time transitions, which are not interleaved like actions but performed simultaneously instead, multiple method calls can terminate at exactly the same moment. This can cause problems as shown in Table 4.2.**B**. Consider method `m` where output parameter `z` is set to input parameter `y`, and after a delay that value is bound to `x`. When concurrently executing

Table 4.2: Problems solvable with fix actions.

Problem **A**:

```
sel
    delay z+2; p?m
or
    n(z)(v); p?n
les
```

```
n(x)(y)
    delay x
```

Problem **B**:

```
par
    m(2)(x)
and
    m(3)(x)
rap
```

```
m(y)(z)
    z := y;
    delay 1
```

two method calls to `m`, as is shown in the left column of problem **B**, both methods first execute (interleaved) the assignment to their local variable `z`. Then, both methods delay and have to bind different values (2 and 3) to `x` simultaneously. What value should `x` get? To prevent that a variable is assigned several different values at the same instant, a separate fix action is introduced for binding the output parameters. After having performed the `delay` simultaneously, both methods sequentially bind their output parameter to `x` (in nondeterministic order).

Five inference rules can generate a fix action, namely rules 29 (MC$_1$), 32 (MC$_4$), 33 (COMM$_1$), 34 (COMM$_2$) and 37 (DELAY$_1$). Rules COMM$_1$ and COMM$_2$ describe the evaluation and fixation of port expression E_p along which communication will take place. The result is denoted by $\widetilde{p}$ to avoid confusion with the port identifier $\underline{p}$ that has not been evaluated. Similarly, rule DELAY$_1$ defines the evaluation and fixation of E, the time to delay. Again, the result $\widetilde{t}$ is denoted slightly different to distinguish it from the unevaluated literal value t.

Several rules for compound statements discern between the fix action and the silent action τ: 17 (GRD$_1$), 21 (SEL$_1$), 22 (SEL$_2$), 24 (ABORT$_2$) and 27 (INTR$_3$). The reader can easily verify that all these rules leave the context of a constituent performing a fix transition intact as long as that constituent does not tick. For instance, rule INTR$_3$ allows interrupt S_2 to do a fix action without interrupting S_1.

Communication The semantics presented in this dissertation introduces the concept of dynamic port passing in POOSL. Previously, the communication port a message was sent to or received from, was static and determined at compile-time. Now, the port is determined by an expression that is evaluated at run-time. This so-called port expression should yield a string naming a port defined in the port interface of the executing process — see rules 33 (COMM$_1$) and 34 (COMM$_2$).

The late-binding of ports enables dynamic port passing as in the π-calculus [36]: a process can receive the name of the port via which it can communicate with a previously unknown process. This can be convenient for modelling relay stations that have to pass on messages to recipients initially only known by the message's sender.

Semantic rules 35 (COMM$_3$) and 36 (COMM$_4$) take care of producing communication actions. In COMM$_3$ the message parameters $E_1 \ldots E_n$ are evaluated and deepcopied before they are sent as $[data]$ along the message if the port is defined in the port interface of the corresponding process. The copying is required to keep the data spaces of the sender and receiver separated. Before injecting the data into the receivers data space (rule COMM$_4$) the data objects are relabelled to prevent that existing data objects are discarded.

The actual communication takes place via rule 57 (PAR$_6$), discussed in Section 5.2.3.

Time Timing properties of a model can be specified with the `delay`. Its behaviour is defined by rules 37 (DELAY$_1$) and 52 (DELAY$_2$). After evaluating and fixing the time to delay (DELAY$_1$), the `delay` can let an arbitrary amount of time up to t go by (DELAY$_2$). This is called *time additivity* or *time continuity* [17, 37]. In DELAY$_2$ the global variables context is updated to reflect the increased model time: $\sigma\upharpoonleft_t$ denotes $\sigma.proc\{(\sigma.proc.currentTime + t)/currentTime\}$; it increments $currentTime$ by the amount of t, while leaving other variables unchanged.

The semantics of POOSL should exhibit *wait-timing*: processes are allowed to wait for communication. To this end synchronisation primitives allow unbound idling (arbitrary waiting). To ensure *maximal progress*, time is not allowed to progress if actions can be performed. This *action urgency* is to be guaranteed[9] by semantic rule 60 (PAR$_7$) for parallel composition of *behaviour specifications* (see Section 5.2.1). This rule uses the notion of urgent actions. A behaviour specification $BSpec$ is *urgent*, written as $Urgent(BSpec)$, if configuration $BSpec$ can perform an internal action (denoted by $BSpec \xrightarrow{\tau,f}$). A behaviour specification $BSpec$ is urgent within time t, denoted by $Urgent(BSpec, t)$, if there exists a $t' \leq t$ and configuration $BSpec'$ such that $BSpec \xrightarrow{t'} BSpec'$ and $BSpec'$ is urgent. This urgency predicate is comparable to the ones in [18, 20, 58].

As was mentioned, rule PAR$_7$ uses the urgency predicate as a condition to prevent time to progress beyond an instant in time where $BSpec_1$ and/or $BSpec_2$ can perform an action. Note that when $BSpec_1$ and $BSpec_2$ synchronise, rule 57 (PAR$_6$) produces a silent action, making $BSpec_1 \parallel BSpec_2$ urgent. Without the urgency predicate, synchronisation could be postponed indefinitely because of the unbound idling property of the communication primitives.

Temporal deadlock is chosen not to exist in POOSL. To prevent temporal deadlock, statements such as $\surd$ and a blocking `guard` must allow time to progress indefinitely. To this end rules 38 (TICK) and 46 (GRD$_2$) allow unbound idling.

The semantics is expected to be *time-deterministic*: when a configuration c performs a delay t, the resulting behaviour is completely determined by c and t. More precisely: for all c, c', c'' and t, it follows that if $c \xrightarrow{t} c'$ and $c \xrightarrow{t} c''$, then $c' = c''$. Together with maximal progress, time-determinism gives rise to a two-phase execution model. During the first phase actions are executed until none are left, followed by a second phase in which all processes synchronously advance time until one or more processes become urgent. The phases are then repeated.

[9]Rule PAR$_7$ uses a negative condition. In general negative premises could endanger the existence of a unique solution of the transition relation [19]. The existence of a transition system is shown in [51, 50] for a case similar to ours, but we can also show it by a stratification that separates action transition rules from time transition rules. Time transition rules refer negatively to only action transitions, while action transitions do not refer to time transitions. It is therefore impossible to infer a transition whose very existence could prevent it from occurring.

Guard Rules 17 ($\textsc{grd}_1$), 46 ($\textsc{grd}_2$) and 47 ($\textsc{grd}_3$) define the behaviour of the `guard`. Rule $\textsc{grd}_1$ states that the guarded statement S can only perform an action if the guarding expression E evaluates to *true*. The conditions $[\![E]\!].s.(s, \mathit{true}) = 1$ and $\mathit{Timeless}(E, s)$ ensure that E is not probabilistic (deterministically yields *true*) and is independent of *currentTime*. To determine this, the function $\mathit{Timeless} \in \mathit{Exp} \times \mathit{State} \twoheadrightarrow \{\mathit{true}, \mathit{false}\}$ is defined:

$$\mathit{Timeless}(E, (\sigma, \lambda, \tau)) = \begin{cases} \mathit{true} & \text{if } [\![E]\!].s_1.s_1' = [\![E]\!].s_2.s_2' \text{ for all } t \in \mathit{Time}, \\ & \text{where } s_1 = (\sigma, \lambda, \tau),\ s_1' = \big((\sigma', \lambda', \tau'), \beta\big), \\ & s_2 = (\sigma.\mathit{proc}\{t/\mathit{currentTime}\}, \lambda, \tau) \text{ and} \\ & s_2' = \big((\sigma'.\mathit{proc}\{t/\mathit{currentTime}\}, \lambda', \tau'), \beta\big) \\ \mathit{false} & \text{otherwise.} \end{cases}$$

This premise[10] is required for making $\textsc{grd}_1$ implementable and preventing temporal deadlock. For instance, when is S allowed to run in **`[currentTime>3]`**S if the time domain is dense? If $\mathit{currentTime} = 3$, S is not allowed to run, but *any* moment beyond that is too late, hence action urgency will be violated. To avoid the problem altogether, the guarding expression is not allowed to depend on *currentTime*.

4.3 Implementation

The processes in a specification will be mimicked by process objects in the simulator. The implementation of process objects and the communication channels between them is part of the architecture layer and will be discussed in the next chapter. Here, we concentrate on how to execute the statements defined in Section 20, which are used for describing the behaviour of processes.

The collection of process statements is a rich set that contains primitives for parallelism, interrupts, communication, selection, delaying and more. To obtain an executable model, these primitives have to be mapped to the constructs of a target language such as Java, Pascal, C, C++ or Smalltalk. This mapping is nontrivial because the target constructs are different and less powerful — in the sense that the behaviour given by a single POOSL primitive can mostly not be captured by a single construct in the target language, but only by a set of constructs. An additional problem is that there may be a restriction on the composition of certain constructs, whereas POOSL allows each primitive to be combined with any other.

To support a smooth mapping of POOSL statements to different target languages, *execution trees* have been developed. The initial idea for these data structures was conceived by M. Geilen and implemented in SHESim [16]. The further development of the execution trees will now be presented. A brief description can also be found in [7] and earlier results for CCS have been published in [6].

[10]The engine restricts itself to expressions that do not (indirectly) refer to *currentTime* at runtime.

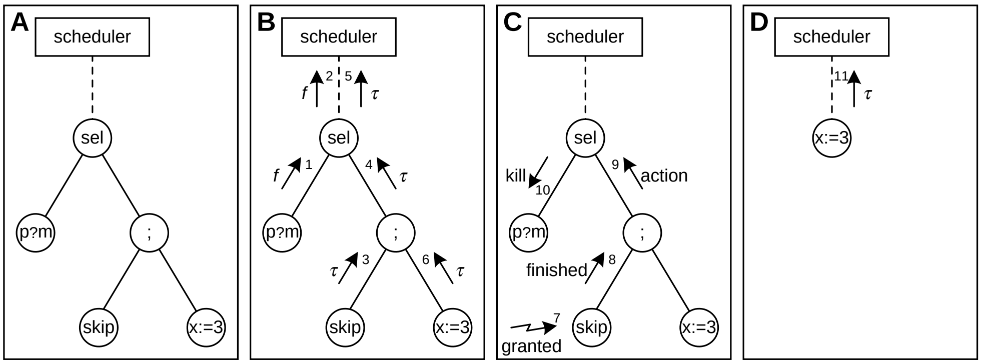


Figure 4.3: Generic execution tree for **sel** p?m **or** skip ; x := 3 **les**. See the text for a detailed description.

4.3.1 Execution Trees (Concept)

[18] introduces execution trees as an implementation method to represent (part of) the state of a process and calculate its next transitions. Actions of the underlying labelled transition system are decorated to form uniquely identifiable *action requests*. Execution trees follow the syntactic structure of a statement and compute the action requests for the next transitions, given their current shape. From these action requests, one is granted, after which the tree adapts itself to reflect the next state. [18] finally shows the correctness of this execution method —with respect to the semantics— by means of bisimulation. We will concentrate on deriving an efficient implementation of execution trees for POOSL, but first the concept of (generic) execution trees is explained, guided by Figure 4.3. It is the starting point for a more detailed discussion in the next section of the actually implemented (optimised) execution trees.

In Figure 4.3.**A**, the initial execution tree is displayed, representing the statement **sel** p?m **or** skip ; x := 3 **les**. Each part of this statement is reflected by a node whose behaviour obeys the inference rules for that part. Nodes of composite statements have ordered directed edges pointing to their children, nodes that represent the constituents of the construct.

After the construction of the execution tree, each leaf node generates a request for performing an action. The actions are propagated towards the root of the tree by means of messages (depicted by numbered arrows). Figure 4.3.**B** shows the message-receive node sending its parent a fix request (1) for performing a fix action to evaluate port expression p, conform semantic rule 34 (COMM$_2$). Rule 21 (SEL$_1$) states that a `select` can perform an action transition if its first branch can. The select node reflects this behaviour by forwarding the fix request (2); since the node is the root of the tree, the request ends up at the scheduler. Similarly, the skip node sends a silent request (3) to its parent (rule 14, EXP) which is forwarded (4) by the sequential-composition node (rule 20, SEQ$_1$) and also (5) by the select node (rule 22, SEL$_2$). The silent request (6) of expression x := 3 is *not* forwarded by the sequential-composition node because there is no semantic rule that allows its second branch to execute before the first one has finished.

From the received requests, a scheduler picks a request and grants it, allowing the model to perform the corresponding transition. When a request is granted, the execution tree adapts itself to reflect the next configuration. Figure 4.3.**C** shows the skip node receiving a granted message (7). After doing its computation —which happens to be nothing in case of `skip`— the skip node signals its parent that it has finished (8). The sequential-composition node replaces itself by its second child (`x := 3`) to yield the terminal statement in rule 20 (SEQ_1), and informs its parent that it has performed an action (9). The select node then kills (10) its first child and replaces itself by its second child, `x := 3` (!), to yield the terminal statement in rule 22 (SEL_2).

The yielded statement, represented by the execution tree in Figure 4.3.**D**, computes the next transitions by reissuing its requests (11). So, execution trees alternately reside in the following phases:

- computing requests to represent the transitions leaving the present configuration;
- adapting the tree to reflect the yielded configuration.

The next section shows how an incremental approach can yield an efficient implementation by avoiding the recalculation of requests after each execution step. After the second step, the execution engine is in a stable state — it can therefore be used as a slave component in a larger simulation framework.

4.3.2 Execution Trees (Implementation)

This section presents the implementation of execution trees in rotalumis, suitable for handling large industrial models. This implementation arose from earlier realisations after careful assessment of their pros and cons. Relevant aspects to the implemented execution trees are motivated by properties of the semantics of POOSL. The section concludes with a survey of the implementation choices and discusses what properties a language should possess to enable these choices.

4.3.2.1 Initialisation

At the start of a simulation, each process constructs its initial execution tree, consisting of a single node: the root node. The root node implements semantic rule 53 (PROC). After being constructed, the root node issues a silent request at the scheduler to evaluate and bind the instantiation parameters of the process, set the other instance variables to *nil*, and create a method-call node to represent the initial method call. During execution, the root node remains to function as a representative of the execution tree.

4.3.2.2 Separating Construction from Activation

The example in Figure 4.3.**B** shows that the sequential-composition node initially blocks requests from its second child because semantic rule 20 (SEQ_1) states that the first branch must be finished before the second branch can execute. This gives rise to separating a node's construction from its activation. Nodes are activated by sending them a *startup* message[11]. Most of the nodes for composite constructs do not react

[11]In this context, *message* should be understood as a function call or method call.

to the startup message, yet simply forward it to their children. The three exceptions are the sequential-composition node (that only sends a startup message to the child eligible for execution) and the nodes for `while` and `if`. The latter two first issue a request for evaluating their condition before they construct and activate their child, a subtree representing the body or branch.

When leaf nodes receive a startup message, they issue a request for performing a transition. The method-call node, for instance, issues a fix request to evaluate and bind its input parameters and construct its body, conform semantic rule 29 ($\textsc{mc}_1$). After the body has been built, it either becomes the child of the method-call node or replaces that node if output parameters are absent (as is the case with tail-recursive calls).

4.3.2.3 Conditional Granting

Inspection of the rules for statements of POOSL reveals that most of them simply infer the same transition as stated in their premises; only the `guard`, `interrupt` and sequential composition may block transitions. The sequential-composition node can be excluded from this list, because only the branch eligible for execution is activated; this guarantees that the node will not receive any requests it will have to block. We now examine when the other two constructs (`guard`, `interrupt`) block their children's requests.

An interrupt node represents the behaviour defined by semantic rules 25 ($\textsc{intr}_1$), 26 ($\textsc{intr}_2$), 27 ($\textsc{intr}_3$) and 28 ($\textsc{intr}_4$). The node can be in two states: normal operation and interrupted. During normal operation, requests from the first (interruptible) branch can be executed freely, but when the node is in the interrupted state, requests of the first branch are suspended (blocked) until the second branch, the interrupt, terminates.

The guarded statement $[E]S$ allows S to execute if and only if E evaluates to *true* (17, $\textsc{grd}_1$). This means that the requests of the subtree representing S may only be issued if E yields *true*, in the current data context. After each execution step, this data context may have changed, influencing the outcome of E and therefore the blocking of requests of S. However, it is computationally too expensive to compute E after each transition, and so is checking whether a transition actually has influenced the outcome of E.

To avoid the overhead of recalculating requests after each step, interrupt nodes and guard nodes forward all of their children's requests, regardless of whether these requests should be blocked or not. Because now every node forwards requests from its children until they end up at the scheduler, leaf nodes can just as well send their requests to the scheduler directly, without the intervention of other nodes in the execution tree. In order to still respect the semantics of the `guard` and `interrupt`, *conditional granting* is introduced.

With conditional granting, the scheduler grants requests like before, but now the granted request must check if it is *executable*. A request is executable if the corresponding node that issued it neither has guards that evaluate to *false*, nor is interrupted. If the request is executable, the requested transition is performed; otherwise the request is blocked and the scheduler will have to grant another request.

Because any leaf node issues at most one request at a time, the node itself can function as its request. So, in the implementation, issuing a request simply becomes handing over the memory address of the requesting leaf node to the scheduler. The address automatically enables the scheduler to inform the requestor of its granted request. This implementation choice prevents the need to construct additional objects that would function as requests, and means a significant increase in performance with respect to execution speed.

4.3.2.4 Adaptation

A node that has performed an action notifies the rest of the tree by sending an *action message* to its parent. The message carries several parameters that tell the receiving parent what kind of transition was performed —an action transition or a time transition— and which child sent it. The message also indicates whether the child finished (has become $\surd$, the terminated statement). This information is used to determine the residual behaviour of a construct. The following paragraphs show why this information is needed.

Discerning terminated behaviour from nonterminated behaviour The termination function, defined by Table 4.1, inventories the statements whose residual behaviour depends on a constituent being terminated or not. The termination function has an implementation that is distributed over the different execution tree nodes: each node decides (based on its transition rules) what to do when one of its children terminates. As an example, we will examine how an interrupt node implements the behaviour defined by the termination function and semantic rule 28 (INTR$_4$) in Appendix A.2 — notice that this rule implies the first statement (S_1) to be interrupted. When the second child (interrupt) of an interrupt node finishes, it is replaced by a new subtree representing the interrupt, and the first child is resumed. If the second child performed an action without terminating, the first child would not have been resumed.

Discerning action requests from time requests The semantic rules for `interrupt` clearly show that a node must also know the nature of the transition that has just been performed. Semantic rule 27 (INTR$_3$) (for action transitions) and rule 49 (INTR$_5$) (for time transitions) define different residual behaviour for an `interrupt` when the interrupting branch performs a transition while the interruptible branch is not suspended. If a nonterminating time transition has occurred, the interrupt node can keep the interruptible branch running, whereas a nonterminating action transition would suspend that branch.

No distinction between different kinds of actions A closer inspection on semantic rule 27 (INTR$_3$) reveals that the fix request is handled differently from communication requests or the silent requests. However, in the implementation, nodes do *not* have to distinguish between different kinds of actions as long as the node that caused the action does not terminate. We can show this by examining the semantic rules for the statements in *Stat* involved with action transitions, namely rules 14–37. Rules 17 (GRD$_1$), 21 (SEL$_1$), 22 (SEL$_2$), 24 (ABORT$_2$) and 27 (INTR$_3$) draw a distinction between between fix actions and other actions, but it is also easy to see that for fix actions not yielding tick, the only part changed in the resulting configuration of these rules is the statement already changed in the premise. For instance, a fix transition for rule 27 (INTR$_3$) only changes S_2 into S'_2, and leaves the rest of **interrupt** S_1 **with** S_2, S_3 unchanged if $S_2 \neq \surd$. So, in the implementation there is no need for nonterminating nodes to inform their parent or any other node in the tree of fix actions, because they do not react to that message. Furthermore, none of the rules 14–37 discern between internal actions and communication actions. Since these are the only actions communicated between nodes, the implementation does not need to know their type either. Only if a node ceases to exist, the type of action can be important.

Terminating nodes The *kill message* is introduced because some inference rules (21 (SEL$_1$), 22 (SEL$_2$), 23 (ABORT$_1$), 24 (ABORT$_2$) and 26 (INTR$_2$)) discard statements. Semantic rule 22 (SEL$_2$), for instance, can transit from **sel** S_1 **or** S_2 **les** to S'_1, if S_1 transits to S'_1 because of a (non-fix) action. This means that the subtree representing S_2 is no longer needed. To free the resources used by that subtree, a kill message is sent to its root. Composite nodes receiving this message, forward it to their children, and commit suicide afterwards. Leaf nodes first retract their requests (if they had issued any) before they cease to exist.

4.3.2.5 Incorporating Data

Each process object has global variables (instance variables) and this set of references is fixed. The set of local variables on the other hand is dynamic: each called method can introduce new variables. These variables are accessible to constructs of the method's body only, and after the method returns its result(s), the local variables are no longer needed. This observation has led to the implementation depicted in Figure 4.4 for incorporating local data in execution trees.

A local variables environment is a data entity that not only stores the local variables themselves (the references to local data objects) but also keeps a reference count of the execution tree nodes using this environment. When the reference count reaches zero,

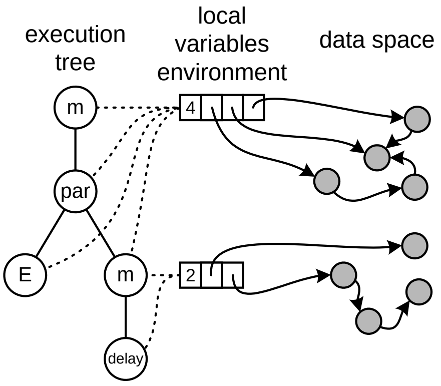


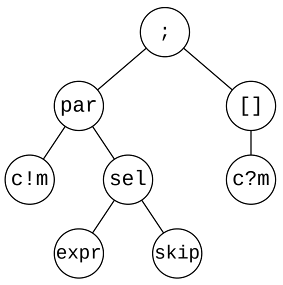
Figure 4.4: Execution tree nodes can access data through local variables.

the local variables are no longer needed and the environment can destroy itself. The garbage collector (Section 3.3.3) takes care of reclaiming unreachable data objects.

The method-call node is given the responsibility of creating the local variables environment and loading a subtree that represents the corresponding method body. First, the method-call node issues a request to fix the method parameters. The resulting data objects are stored in a new local variables environment that is handed down to the constructor of the subtree. New nodes receive a reference to this local variables environment when they are created.

4.3.2.6 Tree Constructor

For the construction of execution trees, the POOSL compiler converts the statements found in process methods into sequences of instructions: bytecode. The bytecode is a static part of the execution engine (Figure 1.7). For each statement a different instruction exists, but all instructions have two things in common: an identifier to determine what kind of statement it represents and the number of children of the statement. The children immediately follow this instruction, but are not listed contiguously. The tree is stored in a pre-ordered fashion, and the list of children can therefore be interleaved by the children's children (Figure 4.5). The pre-order traversal simplifies the finite-state machine that builds trees. An extra instruction marks the end of the list of children of composite nodes.



```
par
    c!m
and
    sel
        x := 3
    or
        skip
    les
rap;
[x == 2] c?m
```

;	par	c!m	sel	expr	skip	↑	↑	[]	c?m	↑	↑

Figure 4.5: The execution tree (left) corresponds to the process statements on the right. Below is the marshalled tree as it is stored in the compiled specification (the arrows mark the end of a list of children).

4.3.2.7 Survey

Execution trees form a means to implement the behaviour of statements on the basis of the corresponding semantic rules. This section restates the requirements for the implementation choices as they were discussed for POOSL in the previous sections. If a different language fulfills these prerequisites, the same techniques can be applied to execution trees for that language as well.

After the construction of a subtree, its activation can be postponed for as long as the requests of that tree are all blocked. Actually, there is no need to build the tree before it can issue requests — this reduces the memory requirements during simulation.

Some execution tree nodes may have to block requests on the basis of certain conditions. By postponing these checks until granting a possibly inexecutable request, requests can be sent to the scheduler directly. This avoids the overhead of other execution tree nodes along the path handling the request as it is forwarded towards the root before it ultimately reaches the scheduler. Notice that this is a tradeoff because of the performance penalty coming from the additional requests being issued.

The action message, used for informing the execution tree that a transition has been performed, is sent from the leaf node that caused the transition to the nodes along the path towards the root. The message can carry additional information, that tells what kind of transition occurred, if a statement terminates and so on. The need for such information depends on the specific semantics of the language being implemented. Next to the action message, the tree can also be transformed by sending a kill message to subtrees that must be discarded because the statements they represent no longer exist in the yielded configuration.

A local variables environment can be referred to by execution tree nodes representing statements in the scope of these variables. By keeping track of the number of referees, it is possible to clean up the environment when it is no longer used. If the language offers parallelism within a single process, a multiple of active local variable environments can coexist, rendering the usual approach of storing local variables on a stack impossible.

4.3.3 Scheduler

Execution trees and channel trees (Section 5.3.1) issue their requests at the scheduler. Requests are stored in two lists — one just for action requests, the other for time requests. This separation is merely for implementation convenience. Given the issued requests, the scheduler determines which transition will be taken next. The set of transitions thus encountered is called a (simulation) trace. The following sections discuss the details of granting requests (the scheduling policy).

4.3.3.1 Action requests

The scheduler must obey several rules when it chooses a request; for instance, the semantics prescribes action urgency, so action requests have priority over time requests.

In rotalumis, the action request list is implemented as a dynamic linear array. Such a list can hold a varying number of requests and enables cheap insertion and removal of requests.

If several action requests are available, the scheduler has to pick one to resolve the nondeterminism. The next section discusses how requests are chosen. The selected request is granted by sending a granted message to the execution tree node that issued the request. Remember that an execution tree leaf node acts as its own request to prevent the need for constructing a new object for representing the request. The leaf node notifies the scheduler if the corresponding transition was performed. If the request was inexecutable, the scheduler will have to mark it, and choose another one until either a transition step has been made or no executable action requests are left.

4.3.3.2 Time requests

When none of the action requests can be executed, the scheduler tries to advance the model time. From the executable time requests, the scheduler chooses the ones with the smallest value. The model time is advanced by this amount to make maximal progress, the executable time requests are decreased accordingly and the expired requests are granted. To respect action urgency (rule 60, PAR$_7$), the scheduler returns to granting action requests, which may have become executable now (but were not before).

Time is handled differently from actions. Semantic rule 52 (DELAY$_2$) shows the special nature of time: a delay statement willing to delay for some amount of time, is also willing to let only part of that time go by and wait for the remainder of the period afterwards. This means that a delay node representing **delay** t would have to issue a possibly infinite set of time requests to represent its willingness to wait for an arbitrary amount of time up to t. Since this approach is infeasible, the following technique is used instead (see [18] for a similar approach).

A delay node, representing **delay** E, first issues a fix request for evaluating E, which yields the time to delay, say t, as stated by semantic rule 37 (DELAY$_1$). Then, the delay node issues a time request of value t. When time advances by $t' < t$, the value of the time request is simply decreased by t', allowing it to wait for only $t - t'$ from now on. Section 6.4 elaborates on efficiently computing the minimal time request and reducing the overhead of adjusting the time requests. When a time request becomes zero, it is said to expire and the scheduler will send a granted message to inform the corresponding delay node, which terminates (as specified by the termination function in Table 4.1).

4.3.3.3 Resolving nondeterminism

The transition system of a POOSL model contains nondeterminism. For performance analysis, we need an entirely probabilistic system, so the nondeterminism should be resolved (Figure 4.6). An easy way to let the scheduler efficiently resolve the nondeterminism is by using a uniform distribution. We are aware that the result of performance analyses depends on the transition selection algorithm, but resolving nondeterminism differently is subject of future research.

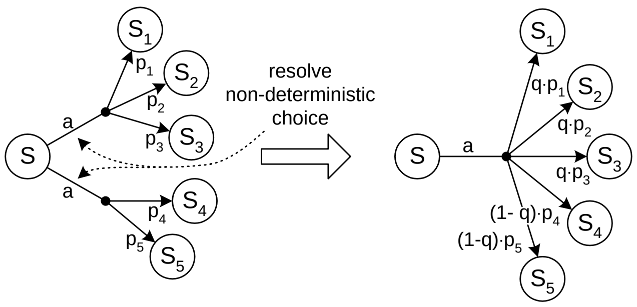


Figure 4.6: Resolving nondeterminism.

From the user's point of view, it is attractive that the scheduler chooses each alternative transition with equal probability. For instance, when a `select` is executed with identically starting branches, none of the branches is expected to have priority over the other. It is easy to choose requests according to a uniform distribution if the list of requests is a linear array: a random number generator produces the index of the request that the scheduler grants.

4.4 Summary

This chapter has discussed the process layer of POOSL, adding parallelism within processes. Communication primitives can now be equipped with an optional expression (*immediate data*) that is evaluated immediately after the communication has taken place.

The semantics of [41] is expanded with the real-time concepts discussed in [17] and given probabilistic features. Proving equivalence between the semantics described in this thesis and (parts of) the former definitions, as well as proving desired properties, such as time-additivity, are part of future research. Furthermore, both dynamic port passing as in the π-calculus and implementation inheritance are added to the language.

The added concepts of time and data cause problems discussed in Section 4.2.3.1. The introduction of a different kind of internal action, called *fix action*, solves these problems.

For implementing the behaviour specifications, the generic execution trees of [18] have been explained and adapted to efficiently execute statements (Section 4.3.2). The language properties enabling these optimisations have been summarised in Section 4.3.2.7, along with some of the characteristics of the implemented execution trees.

Section 4.3.3 has discussed a scheduling policy that resolves nondeterminism while respecting the action urgency and maximal progress as stated by the semantics.

Chapter 5

POOSL Architecture Layer

5.1 Overview

This chapter presents the architecture layer of POOSL. The architecture of a system is a hierarchical topology of processes and channels defined by *behaviour specifications*. After introducing the constructs for building such specifications, the structural operational semantics of the previous chapter is extended to provide the meaning of constructs of the architecture layer. Then, a generic approach for implementing behaviour specifications is presented, based on their semantic rules.

5.2 Specification

5.2.1 Abstract Syntax

The set of all POOSL system specifications *SSpecifications*, ranged over by *SSpec*, is defined by:

$$\begin{array}{lll} \mathit{SSpec} = & \textbf{system specification} & \\ & \textbf{behaviour specification} & \mathit{BSpec}^b \\ & \mathit{CDList}, & \end{array}$$

where $\mathit{CDList} = \mathit{CD}_1 \cdots \mathit{CD}_n$ is a list of class definitions; its existence is implicitly assumed in the semantic rules. The system specification contains the *top-level* behaviour specification BSpec^b that describes the architecture of the system in terms of a topology of processes and channels. This topology can be defined hierarchically by means of *clusters*, for which *ClassDef* is finally extended to include cluster class definitions:

$$\begin{array}{lll} \mathit{CD} = & \textbf{cluster class} & C(y_1, \ldots, y_r) \\ & \textbf{port interface} & \underline{p_1 \cdots p_g} \\ & \textbf{message interface} & \overline{ms_1 \cdots ms_h} \\ & \textbf{behaviour specification} & \mathit{BSpec}^b. \end{array}$$

A cluster class has instantiation parameters $y_1 \ldots y_r$ and a port and message interface, just as process class definitions. Like system specifications, cluster classes describe their constituents with BSpec^b, which is a typical element of $\mathit{BSpecifications}^b$, the basic set of *behaviour specifications*:

$$\begin{array}{lcll} \mathit{BSpec}^b & = & C(PE_1,\ldots,PE_r) & \text{process/cluster instantiation} \\ & \mid & \mathit{BSpec}^b_1 \parallel \mathit{BSpec}^b_2 & \text{parallel composition} \\ & \mid & \mathit{BSpec}^b \setminus L & \text{hiding} \\ & \mid & \mathit{BSpec}^b\,[f] & \text{relabelling.} \end{array}$$

A behaviour specification can either be a single process or cluster, parameterised with expressions $PE_1 \ldots PE_r$, or it can be composed of several instances running concurrently (specified by the parallel composition operator $\parallel$).

The port interface of a constituent can be modified in two ways. Ports can be hidden ($\mathit{BSpec}^b \setminus L$) by listing their names in the hiding set L. The name of a port can be changed by means of relabelling ($\mathit{BSpec}^b\,[f]$). We introduce function $f \in \mathit{Act} \rightarrowtail \mathit{Act}$ to relabel a set of ports and write $\underline{p'_1}/\underline{p_1} \ldots \underline{p'_n}/\underline{p_n}$ to define f such that $f(p_i) = p'_i$ for $i \in \{1,\ldots,n\}$ and $f(p) = p$ otherwise. Relabelling function f is extended by:

$$\begin{array}{lcl} f(\underline{p}!m(\underline{n})) & = & \underline{f(p)}!m(\underline{n}) \\ f(\underline{p}?m(\underline{n})) & = & \underline{f(p)}?m(\underline{n}) \\ f(\underline{p}!m[\mathit{data}]) & = & \underline{f(p)}!m[\mathit{data}] \\ f(\underline{p}?m[\mathit{data}]) & = & \underline{f(p)}?m[\mathit{data}] \\ f(\tau) & = & \tau \\ f(f) & = & f. \end{array}$$

Equally named ports within a single cluster are interconnected by an imaginary channel with the same name. To demonstrate this, Figure 5.1 gives a graphical representation of the following cluster definition:

$$\begin{array}{lcll} CD & = & \textbf{cluster class} & E \\ & & \textbf{port interface} & \underline{q}\ \underline{s} \\ & & \textbf{message interface} & \underline{q}!\,\mathit{out}(2)\ \underline{s}?\mathit{ack}(0) \\ & & \textbf{behaviour specification} & ((A \parallel B)[\underline{s}/\underline{r}] \parallel (C \parallel D)) \setminus \{\underline{p}\} \end{array}$$

assuming that class A has ports p, q and r, class B has ports p and q, class C has ports p and s, and finally, class D has port p.

To describe a system specification during execution, the following set of extended behaviour specifications *BSpecifications* is defined:

$$\begin{array}{lcl} \mathit{BSpec} & = & C(PE_1,\ldots,PE_r) \\ & \mid & \mathit{BSpec}_1 \parallel \mathit{BSpec}_2 \\ & \mid & \mathit{BSpec} \setminus L \\ & \mid & \mathit{BSpec}\,[f] \\ & \mid & \big([S]^{\psi}_{C},\sigma,\tau\big). \end{array}$$

BSpecifications encompasses $\mathit{BSpecifications}^b$ and adds $\big([S]^{\psi}_{C},\sigma,\tau\big)$ to denote that statement S is being executed in a process of class C in the local variables context ψ and global variables context σ (type information is stored in τ).

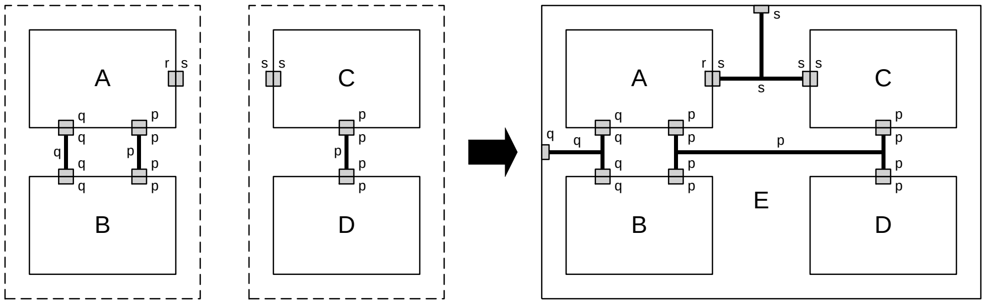


Figure 5.1: The construction of the behaviour of E is shown in two stages. On the left, respectively $(A \parallel B)[s/r]$ and $C \parallel D$ are shown. In the final stage (shown on the right) these components are combined to form cluster E.

From the discussion of $C(PE_1, \ldots, PE_r)$, we know that processes and clusters can be parameterised. However, only a restricted set of expressions is allowed. This set of parametric expressions $PExp$ is defined by:

$$\begin{array}{rcl} PE & = & y \\ & | & \underline{\gamma} \\ & | & \textbf{new}(C) \\ & | & PE\ m(PE_1, \ldots, PE_n) \\ & | & PE_1; PE_2 \\ & | & \textbf{if}\ PE_c\ \textbf{then}\ PE_1\ \textbf{else}\ PE_2\ \textbf{fi} \\ & | & \textbf{while}\ PE_c\ \textbf{do}\ PE\ \textbf{od}. \end{array}$$

Parametric expressions are evaluated in the context of clusters or in the context of the top-level specification. In neither environment instance variables are available, hence assignments to and the use of local variables has been prohibited. Because **self** and $E\ m_C(E_1, \ldots, E_n)$ are only allowed in the context of data objects, these constructs have been omitted as well. Only the use of instantiation parameters y of the current context is allowed.

5.2.2 Context Conditions

Several functions are defined to facilitate the stating of the context conditions for the architecture layer. Function *Port* extracts the port identifier from a message signature and is defined by:

$$\begin{array}{lcl} Port(\underline{p}!m(\underline{n})) & = & p \\ Port(\underline{p}?m(\underline{n})) & = & p \\ Port(\underline{p}!m[data]) & = & p \\ Port(\underline{p}?m[data]) & = & p \\ Port(\tau) & = & \underline{\text{undef}} \\ Port(f) & = & \underline{\text{undef}}. \end{array}$$

Function *MSS* computes the *message signature sort* of various constructs:

$$
MSS(C)=\begin{cases}
\{ms_1,\ldots,ms_h\} & \text{if } CDList \equiv CD_1\cdots CD_i\cdots CD_n \text{ and} \\
 & CD_i \equiv \begin{array}{ll}\textbf{process class} & C(y_1,\ldots,y_r)\\ \textbf{port interface} & \underline{p_1}\cdots\underline{p_g}\\ \textbf{message interface} & \underline{ms_1\cdots ms_h}\\ \textbf{instance variables} & x_1\cdots x_n\\ \textbf{initial method call} & m_{C'}(E_1,\ldots,E_q)(\,)\\ \textbf{instance methods} & MD^p_1\cdots MD^p_k\end{array} \\
MSS(C_{super})\cup \{ms_1,\ldots,ms_h\} & \text{if } CDList \equiv CD_1\cdots CD_i\cdots CD_n \text{ and} \\
 & CD_i \equiv \begin{array}{ll}\textbf{process class} & C(y_1,\ldots,y_r)\\ \textbf{extends} & C_{super}\\ \textbf{port interface} & \underline{p_1}\cdots\underline{p_g}\\ \textbf{message interface} & \underline{ms_1\cdots ms_h}\\ \textbf{instance variables} & x_1\cdots x_n\\ \textbf{initial method call} & m_{C'}(E_1,\ldots,E_q)(\,)\\ \textbf{instance methods} & MD^p_1\cdots MD^p_k\end{array} \\
\{ms_1,\ldots,ms_h\} & \text{if } CDList \equiv CD_1\cdots CD_i\cdots CD_n \text{ and} \\
 & CD_i \equiv \begin{array}{ll}\textbf{cluster class} & C(y_1,\ldots,y_r)\\ \textbf{port interface} & \underline{p_1}\cdots\underline{p_g}\\ \textbf{message interface} & \underline{ms_1\cdots ms_h}\\ \textbf{behaviour specification} & BSpec^b\end{array} \\
\varnothing & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{lcl}
MSS(C(PE_1,\ldots,PE_r)) & = & MSS(C)\\
MSS(BSpec^b_1 \parallel BSpec^b_2) & = & MSS(BSpec^b_1)\cup MSS(BSpec^b_2)\\
MSS(BSpec^b \setminus L) & = & \{ms \in MSS(BSpec^b) \mid Port(ms)\notin L\}\\
MSS(BSpec^b[f]) & = & \{f(ms)\mid ms\in MSS(BSpec^b)\}.
\end{array}
$$

Likewise, function *PS* computes the *port sort* of cluster classes, process classes and behaviour specifications:

$$
\begin{array}{lcl}
PS(C) & = & \{Port(ms)\mid ms\in MSS(C)\}\\
PS(BSpec^b) & = & \{Port(ms)\mid ms\in MSS(BSpec^b)\}.
\end{array}
$$

The following context conditions should be met by a system specification, next to the conditions stated in Sections 3.2.2 and 4.2.2:

1. Each class used in the behaviour specification of a cluster class is defined.
2. Any variable used in the behaviour specification of a cluster must be an instantiation parameter of that cluster.
3. The port interface and message interface of cluster classes should be consistent with their behaviour specifications. Let cluster class C be defined by:

   $$\begin{array}{ll}\textbf{cluster class} & C(y_1,\ldots,y_r)\\ \textbf{port interface} & \underline{p_1}\cdots\underline{p_g}\\ \textbf{message interface} & \underline{ms_1\cdots ms_h}\\ \textbf{behaviour specification} & BSpec^b.\end{array}$$

   The port interface of C is consistent with $BSpec^b$ iff $PS(BSpec^b)\subseteq\{p_1,\ldots,p_g\}$. Similarly, the message interface is consistent iff $MSS(BSpec^b)\subseteq\{ms_1,\ldots,ms_h\}$.

4. The port interface and message interface of a process class must agree. The port interface $p_1 \dots p_g$ of process class C is consistent with C's message interface iff $PS(C) \subseteq \{p_1, \dots, p_g\}$.
5. The port sort of the top-level specification is empty.
6. Behaviour specifications of cluster classes may not be defined (indirectly) recursively.

5.2.3 Structural Operational Semantics

Although architecture and process classes are described in separate conceptual layers, their meaning is provided by a single semantics, the structural operational semantics of Section 4.2.3, which will now be completed. The added semantic rules are listed in Appendix A.3.

The countable set of configurations $\mathit{Conf} = \mathit{BSpecifications}$ is ranged over by c. Notice that configurations of the form $\big([S]^{\psi}_{C}, \sigma, \tau\big)$ have already been encountered in Chapter 4. The initial configuration on page 81, BSpec^b, is the top-level behaviour specification that recursively defines the architecture of the model.

The definition of the model time update function $\upharpoonleft_t$ is modified to include the following behaviour specifications:

$$\begin{array}{lcl}
\big(\mathit{BSpec}_1 \parallel \mathit{BSpec}_2\big)\upharpoonleft_t & = & \mathit{BSpec}_1\upharpoonleft_t \parallel \mathit{BSpec}_2\upharpoonleft_t\\
\big(\mathit{BSpec} \setminus L\big)\upharpoonleft_t & = & \mathit{BSpec}\upharpoonleft_t \setminus L\\
\big(\mathit{BSpec}\,[f]\big)\upharpoonleft_t & = & \mathit{BSpec}\upharpoonleft_t\,[f]\\
\big([S]^{\psi}_{C}, \sigma, \tau\big)\upharpoonleft_t & = & \big([S]^{\psi}_{C}, \sigma\upharpoonleft_t, \tau\big).
\end{array}$$

Cluster initialisation When a cluster is initialised, it is replaced by its behaviour specification (rule 54, CLUS). Any occurrence of C's instantiation parameters $y_1 \dots y_r$ is syntactically replaced by expressions $PE_1 \dots PE_r$ respectively. To this end the set of syntactic substitution functions $\mathit{SyntSubst} = \mathit{Var} \longrightarrow \mathit{PExp}$ with typical elements ς is introduced. We write $PE_1/y_1 \dots PE_r/y_r$ to define ς such that $\varsigma(y_i) = PE_i$ for $i \in \{1, \dots, r\}$ and $\varsigma(y) = y$ otherwise. $\mathit{BSpec}[\![\varsigma]\!]$ denotes the application of syntactic substitution function ς to the parametric expressions in BSpec and is defined inductively by:

$$\begin{array}{ll}
C(PE_1, \dots, PE_r)[\![\varsigma]\!] & \equiv\ C(PE_1[\![\varsigma]\!], \dots, PE_r[\![\varsigma]\!])\\
(\mathit{BSpec}_1 \parallel \mathit{BSpec}_2)[\![\varsigma]\!] & \equiv\ \mathit{BSpec}_1[\![\varsigma]\!] \parallel \mathit{BSpec}_2[\![\varsigma]\!]\\
(\mathit{BSpec} \setminus L)[\![\varsigma]\!] & \equiv\ \mathit{BSpec}[\![\varsigma]\!] \setminus L\\
(\mathit{BSpec}\,[f])[\![\varsigma]\!] & \equiv\ \mathit{BSpec}[\![\varsigma]\!]\,[f]\\
y[\![\varsigma]\!] & \equiv\ \varsigma(y)\\
\underline{\gamma}[\![\varsigma]\!] & \equiv\ \underline{\gamma}\\
\mathbf{new}(C)[\![\varsigma]\!] & \equiv\ \mathbf{new}(C)\\
(PE\ m(PE_1, \dots, PE_n))[\![\varsigma]\!] & \equiv\ PE[\![\varsigma]\!]\ m(PE_1[\![\varsigma]\!], \dots, PE_n[\![\varsigma]\!])\\
(PE_1; PE_2)[\![\varsigma]\!] & \equiv\ PE_1[\![\varsigma]\!]; PE_2[\![\varsigma]\!]\\
\mathbf{if}\ PE_c\ \mathbf{then}\ PE_1\ \mathbf{else}\ PE_2\ \mathbf{fi}[\![\varsigma]\!] & \equiv\ \mathbf{if}\ PE_c[\![\varsigma]\!]\ \mathbf{then}\ PE_1[\![\varsigma]\!]\ \mathbf{else}\ PE_2[\![\varsigma]\!]\ \mathbf{fi}\\
\mathbf{while}\ PE_c\ \mathbf{do}\ PE\ \mathbf{od}[\![\varsigma]\!] & \equiv\ \mathbf{while}\ PE_c[\![\varsigma]\!]\ \mathbf{do}\ PE[\![\varsigma]\!]\ \mathbf{od}.
\end{array}$$

Hiding ports Rule 58 (HIDE$_1$) excludes communication actions from *BSpec* that are sent to a port listed in the hiding set L. If *BSpec* can perform an internal action (τ, f), so can $BSpec \setminus L$; rule 61 (HIDE$_2$) specifies the same for time transitions.

Relabelling ports Rule 59 (RELAB$_1$) alters the port description in communication actions as specified by relabelling function f. Internal actions and time transitions (rule 62, RELAB$_2$) remain unchanged.

Parallel composition The semantic rules 55 (PAR$_4$), 56 (PAR$_5$) for the parallel composition operator $\|$ implement interleaving concurrency to combine the transition systems of two behaviour specifications. Rule 57 (PAR$_6$) allows the two components to synchronise. The composite can perform a silent action τ if $BSpec_1$ can perform a communication action ℓ while $BSpec_2$ can perform the complementary communication action $\overline{\ell}$. The probability of this action is the product of the probabilities of the corresponding actions ℓ and $\overline{\ell}$. Finally, inference rule 60 (PAR$_7$) describes how two parallel components synchronously allow time to progress; PAR$_7$ has already been discussed in Section 4.2.3.1.

5.3 Implementation

5.3.1 Channels

5.3.1.1 Generic Approach

Section 5.2.1 explained that the model's topology of processes and channels is defined by behaviour specifications, using three different kinds of constructs. Two of them, hiding and relabelling, modify the ports of a component. The third, parallel composition, allows components to synchronise if their communication actions match. Conceptually, this can be interpreted as if parallel composition creates channels between equally named ports.

A *channel tree* implements the behaviour of these constructs. It relabels, hides and combines communication requests from processes. The channel tree looks for matching communication requests, and issues an internal action request at the scheduler for each match it can find.

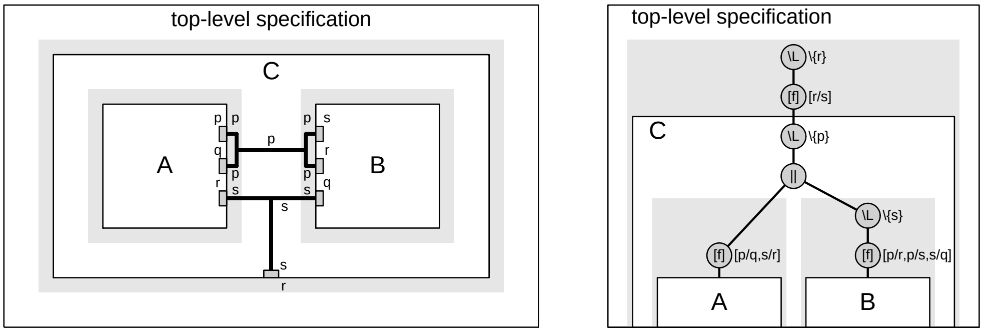


Figure 5.2: Example of a topology (left) and a hierarchical view (right) showing the channel tree. The light gray entities represent adapted instances.

Figure 5.2 depicts the architecture and the corresponding channel tree for the following behaviour specification (inessential details have been omitted):

`system specification`	
`behaviour specification`	$C[\underline{r}/\underline{s}] \setminus \{\underline{r}\}$
`cluster class`	C
`port interface`	$\underline{s}$
`behaviour specification`	$\big(A[\underline{p}/\underline{q}, \underline{s}/\underline{r}] \parallel B[\underline{p}/\underline{r}, \underline{p}/\underline{s}, \underline{s}/\underline{q}] \setminus \{\underline{s}\}\big) \setminus \{\underline{p}\}$
`process class`	A
`port interface`	$\underline{p}\ \underline{q}\ \underline{r}$
`process class`	B
`port interface`	$\underline{q}\ \underline{r}\ \underline{s}.$

Message-send nodes and message-receive nodes of the execution tree of a particular process transmit their communication requests to the channel tree's leaf node connected to that process. From there on, the requests are forwarded towards the root as follows.

Hiding A hiding node satisfies the behaviour of semantic rule 58 (HIDE$_1$) by forwarding only those communication requests to its parent that are not listed in the hiding set. Notice that HIDE$_1$ never rules out internal actions τ and f.

Relabelling A relabelling node changes the port identifier in the message signature of communication requests in accordance with the relabelling function it represents; this is conform semantic rule 59, RELAB$_1$. Again, requests for internal actions are forwarded unchanged.

Parallel composition The parallel composition node not only forwards the requests of its children to its parent conform rules 55 (PAR$_4$) and 56 (PAR$_5$), but also issues a silent action request (τ) for every pair of matching communication requests it can find (57, PAR$_6$). Like the relabelling and hiding nodes, the parallel composition node does not affect requests for internal actions (τ,f), but simply forwards them.

Notice that the implementation discussed in this section always yields precisely one channel tree; context condition 5 in Section 5.2.2 implies that the root of the channel tree will never have to forward communication requests.

None of the channel tree nodes block or react to requests for internal actions. These requests can therefore be sent to the scheduler directly, without intervention of the channel tree. The same applies to time requests, because semantic rules 61 (HIDE$_2$), 62 (RELAB$_2$) and 60 (PAR$_7$) infer the same time transitions for the composite behaviour specification as their constituents. The latter rule allows time to progress only up to the moment at which actions transitions can be performed again. Section 4.3.3.2 explained that the scheduler already takes care of this behaviour by collecting delay requests from every process in the model and granting only the smallest time request if none of the action requests is currently executable. From this we can conclude that only communication requests have to be sent to the channel tree.

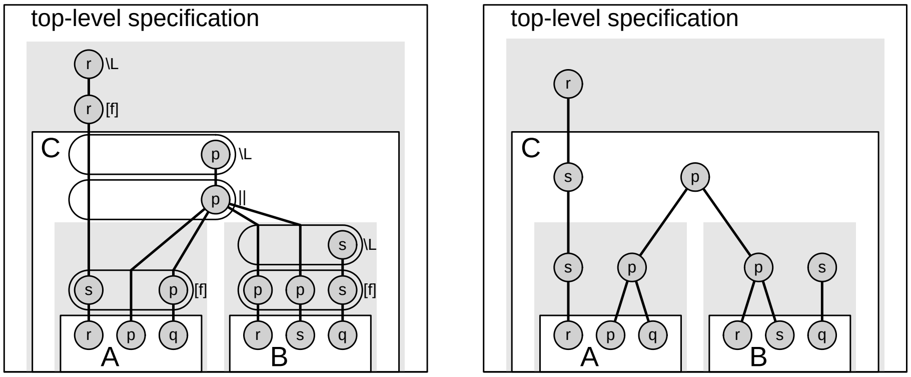


Figure 5.3: Decomposition of the channel tree on a per-channel basis (left), and the actually implemented channel trees (right).

5.3.1.2 Channel Decomposition

The previous section has explained a generic approach to implement the hiding, relabelling and parallel composition operators of behaviour specifications. However, the initial[1] implementation of channels uses a different kind of node —one that combines the behaviour of these three operators— to reduce the height of the channel tree and consequently the processing time per communication request. The new kind of node will simply be called *channel-tree node*. Section 6.3 shows a more efficient implementation for calculating silent action requests for synchronisation.

Figure 5.3 decomposes the relabelling, hiding and parallel composition on a per-channel basis (left). Each node is decorated with the name of the channel it represents. Each channel is mapped onto a separate channel tree. The implemented channel trees are depicted on the right. The behaviour of a channel-tree node is as follows: if and only if the node has a parent, it forwards the communication requests received from its children.

The structure of the channel tree is such that the relabelling and hiding rules for the channel are satisfied. Besides forwarding requests, the channel-tree node also computes matches (like the parallel composition node). Notice that the port identifier of the requests' message signatures automatically match, due to the construction of the tree. An additional check is required however, to ensure that components cannot synchronise with themselves.

The extra node s in cluster C is an implementation artefact introduced to allow clusters and processes being treated similarly. The node temporarily functions as a port (like node p in C) for clusters on the next (higher) hierarchical level during the recursive construction of the model's architecture.

[1]Several optimisations can still be carried out; these are discussed in Section 6.3.

5.3.2 Clusters and Processes

For an attractive graphical representation of behaviour specifications, the implementation of the architecture layer restricts behaviour specifications of the system specification and cluster classes to the following form:

$$BSpec^g \quad = \quad BSpec^l \ \backslash L \ [f]$$

where $[f]$ denotes an optional[2] global relabelling, $\backslash L$ denotes an optional global hiding, and $BSpec^l$ denotes a parallel composition of adapted classes:

$$\begin{array}{rcl} BSpec^l & = & AC \\ & | & BSpec^l_1 \parallel BSpec^l_2. \end{array}$$

An *adapted class* is a class whose ports are adapted by an optional local hiding and local relabelling:

$$AC \quad = \quad C(PE_1, \ldots, PE_r) \ \backslash L \ [f].$$

The restricted set of behaviour specifications is attractive because it allows joining the conceptual boundaries introduced by relabelling and hiding ports with process and cluster boundaries, yielding system specifications that are visually less cluttered. The construction of a model's architecture proceeds as follows[3].

An *adapted instance* represents a behaviour specification of the form $AC = C(PE_1, \ldots, PE_r) \ \backslash L \ [f]$ and contains not only an instance of class C, but also channel-tree nodes to perform the local hiding and relabelling. A *process object* is an object in the implementation, that represents an instance of a process class. A process object owns *port nodes* that represent the ports listed in the port interface of its class. *Cluster objects* are objects in the implementation that represent instances of cluster classes. A cluster object contains a set of adapted instances to represent the adapted classes in the behaviour specification $BSpec^l \ \backslash L \ [f]$ of its class. Furthermore, it contains channel-tree nodes to represent the globally relabelled ports — given by $PS(BSpec^l \, [f])$. These channel-tree nodes also function as the parallel composition of the adapted instances, as explained in the previous section.

The architecture of a model is built recursively, starting at the lowest hierarchical level with the process objects. The ports of these process objects can be adapted by channel-tree nodes. Then, cluster objects are constructed with channel-tree nodes to compose adapted instances in parallel. The ports of cluster objects can also be adapted, just like the ports of process objects, yielding an adapted instance that can be a constituent in a higher hierarchical layer. The construction ends at the top-level specification.

[2]Optional, because f can be chosen to be the identity function. Likewise, the hiding is optional since it is allowed to choose $L = \varnothing$.

[3]For the unrestricted set of behaviour specifications a similar construction can be derived.

5.4 Summary

This chapter first introduced the syntax of constructs in the architecture layer for constructing a specifications' topology of processes and channels. Then, the meaning of these constructs (behaviour specifications) has been added to the structural operational semantics introduced in the previous chapter.

A constructive approach to implementing the behaviour specifications has been discussed. The implementation constructs a process object for each process found recursively in the top-level specification. Channel trees, which interconnect process objects, compute and issue silent action requests to represent synchronisation (rendez-vous) possibilities between processes. Only communication requests are sent to the channel tree, other requests go straight to the scheduler.

Chapter 6

Execution Engine Optimisations

6.1 Introduction

POOSL offers designers a wide variety of language primitives to describe for instance parallelism, communication, guarded commands and selection. Combinations of these primitives allow for even more intricate and varied behaviour. Several combinations may offer different solutions to the same design problem, and it is this flexibility that gives designers the freedom to choose a subset of primitives appropriate for solving their modelling problem. Initially, solutions with concise descriptions may be preferred. These compact specifications enable clear and efficient communication to other team members. During the design process other solutions may become more interesting, for instance because of their speedier execution. To help designers objectively consider alternative solutions that use different primitives, each primitive should —ideally— offer approximately the same performance, regardless of the model's size. If the execution speed of certain primitives differs greatly from others, designers might become prejudiced and use only the subset of faster primitives; hence, a strong imbalance in the execution speed of primitives reduces design freedom.

One of the key issues to contemplate during the implementation of POOSL's language primitives for the simulator, is the size of models. Industrial models tend to be large. Without an efficient implementation, the algorithms behind certain primitives will force the execution of large models to a grinding halt. If the modeler cannot devise an alternative solution that does not use those primitives, lack of simulation speed will render the simulator impracticable. This situation, where the modeler is wasting valuable time trying to cope with the simulator, should be avoided because it distracts the modeler's attention from the real problem: designing the model. Instead, the simulator should use algorithms that minimise the overhead of language primitives even when the size of the model is scaled up. This section discusses the major performance problems, which were encountered during several industrial case studies with earlier versions of the simulator, and presents algorithms that deal with these *scalability* issues.

The performance of the algorithms is shown by stress tests whose results have been collected in Appendix C. After having presented an algorithm, its performance is discussed. An algorithm will be called *efficient* when it is realising the minimum complexity expected to be possible while satisfying the constraints set by the semantics of POOSL. For instance, the execution of a `select` will be linearly dependent on the amount of alternatives it offers, because each of its branches has to construct and register its requests for the next transition. Figure 6.1 shows that the increase in execution time is indeed directly proportional to the number of branches (with equal tasks). Since the actual algorithm has the expected (linear) order of complexity, it will be called efficient. We mention the little twist in the curve when the branch count is low. To explain this and other recurring specifics in the measurements, the following section will explain on what machine the stress tests have been run.

Although we are mostly interested in the complexity of algorithms in rotalumis, we should not forget to keep an eye on its absolute speed. An algorithm with constant complexity scales splendidly, but if its run time is measured in centuries, it clearly looses its practical value. The case studies that have been performed in conjunction with IBM and Alcatel have confirmed that rotalumis is capable of executing large industrial models. At the end of this chapter, in Section 6.7, the scalability is tested for an actual industrial model.

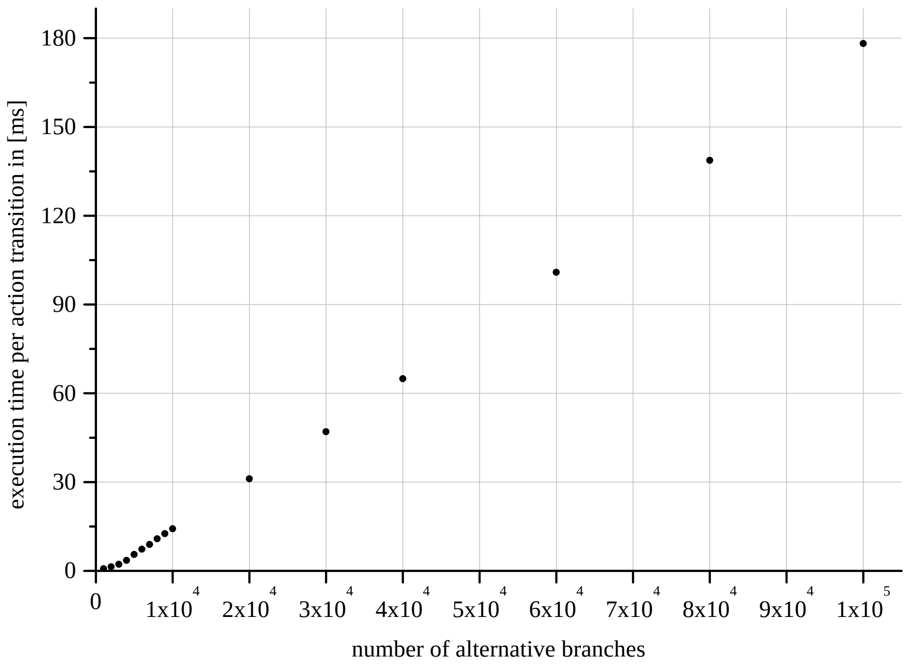


Figure 6.1: Choosing a branch from a `select`.

In guiding the design of the simulator we can ask ourselves the question: where does the simulator spend its time? A close inspection of the scheduler provides an answer to this question. The simulator is said to run at full speed when the scheduler can always immediately pick out an executable request from its request lists. However, as some requests may block, the scheduler might have to select several requests before it provokes a transition. Blocking can be caused by guards, interrupts, delays and (conditional) communication statements. The scheduler marks requests to be able to find the requests it has not tested yet. After executing a request, the scheduler removes the marks and continues with the next transition. The overhead of testing and (un)marking requests can slow the simulator down significantly and is related to the proportion between blocking requests and executable requests. When executable requests are outnumbered by blocking requests, the scheduler probably[1] encounters several blocking requests before it can actually perform a step.

The scheduling overhead is reduced by separating the tested blocking requests from the other requests until a transition is provoked. This prevents the scheduler from testing them twice and simplifies finding the untested requests. After performing a step, the tested requests are moved back in constant time. An optimisation that can significantly speed up simulation is reducing the amount of blocking statements.

6.2 Measuring the Performance of Rotalumis

The performance of rotalumis has been measured with stress tests that explore the capabilities of the execution engine. The tests are carefully designed POOSL models that show the scalability of a particular type of statement. For instance, there is a test to determine the time required for performing a computational task (evaluating an expression) in the presence of other tasks. The measurement (Appendix C, Figure C.20) shows that the execution time per task is nearly independent of the number of tasks that is running in parallel. The graph exhibits two puzzling features that require an explanation: the faster execution time when there are but a few tasks running concurrently, and the minor performance degradation when more tasks are being run. To find the cause of these effects, we will examine the platform on which the tests have been executed.

The execution platform for the stress tests is a machine running the Windows 98 operating system on a Pentium III, 600 MHz microprocessor with 256kB L2-cache, 133MHz data bus and 512MB SD-RAM (100MHz) of memory. The memory available to rotalumis is approximately 400MB, and none of the tests induce swapping of virtual memory pages. All[2] tests have been performed with a single executable whose maximum stack size has been increased to 100MB to allow for the extreme set of models to run. The initial size of the request queue of the scheduler has been set to ten million requests. This eliminates dynamically resizing the queue at run-time, which would distort the time measurements of the stress tests.

[1]The scheduler picks potential requests at random. This allows the simulator to cover the entire transition space in the long run.

[2]The only exception is the test demonstrating the effect of (lacking) tree height reduction (Appendix C, Figure C.18).

The models incorporate a data class `Clock` with primitive methods for measuring the time accurately. The methods implicitly use the `rdtsc` instruction to read the 64-bit clock cycle counter of the microprocessor. Since the operating system that rotalumis is run on is multitasking, the timing results will be slightly off, but we will ignore this effect and try to minimise it by not running other applications during stress testing.

The following three tests written in C++ show scalability effects imposed by this hardware/software configuration, that will bias the actual stress tests. Figure C.24 shows the time required for creating execution tree nodes (a node occupies 56 bytes) with the `new` operator of C++. As the number of nodes increases, the cost for allocating nodes can become significantly higher (for $5 \cdot 10^6$ nodes the costs are a factor of three higher than can be expected from extrapolation). This effect causes measurements of algorithms with theoretically linear complexity to display a slightly ascending curvature. Figures C.26 and C.27 give an indication of the memory access time when randomly modifying parts of an array of a certain size. Randomly accessing memory reflects the type of memory access caused by the (simulated) nondeterminism of the scheduler in the execution engine. For array sizes below 256kB, caching improves memory access significantly, whereas arrays with sizes above 64MB require more time to access. The final major effect that can be retrieved in some stress tests is caused by the `delete` operator of C++. The highly irregular graph shown in Figure C.25 reveals that sometimes destroying *more* execution tree nodes can actually significantly *reduce* the total time destruction takes.

These scalability aspects, related to dynamically creating or destroying nodes and accessing them in random order, will play a smaller or larger role in the actual stress tests. It is assumed that the reader recognises this biasing in the measurements. For example, one of the stress tests measures the execution speed of an `abort` killing N concurrent tasks. The time (see Figure C.21) required for constructing the statement —merely a tree consisting of $N+1$ nodes— is proportional to the time for constructing the same amount of nodes with a simple C++ program. Likewise, the destruction of the N concurrent tasks (Figure C.22) exhibits a highly irregular execution time due to the `delete` operator of C++.

Before discussing the algorithms for enhancing execution speed and their measured behaviour, a model is presented to elucidate the need for such optimisations.

6.3 Communication

Interprocess communication can only occur if a message-send statement of one process *matches* a message-receive statement of another process so as to form a rendez-vous. Two message statements match iff:

- they are complementary: one transmits and the other receives;
- they have identical names;
- they have equal parameter counts;
- the corresponding channel allows communication between the respective processes.

The last condition is fulfilled only when:

- a channel between the two processes exists;
- the hiding and relabelling rules specified for that channel permit communication;
- the semantics for clustering is respected.

If two message statements match, communication can occur if both statements are executable and the optional reception condition evaluates to *true*. Efficiently computing matches is the key to reducing communication overhead, and the following paragraphs show why and how the matching algorithm has evolved since the simulator was brought into action in numerous case studies.

Initial Approach The initial machinery behind communication primitives used *channel trees* to represent channels that interconnect processes. The example in Figure 6.2 shows how such a data structure (middle) represents a channel of a model (left). Each node represents a part of the channel at a certain hierarchical level: in Figure 6.2 node `c` represents the channel at the hierarchical level of cluster `C`, whereas node `d` represents the channel at the highest hierarchical level of the specification. Processes `P` and `Q` are connected to channel `c` that is linked to port `e` of process `R` via channel `d`. We will assume that process `P` currently has two message-send statements `a!m` (whose requests will be represented by `!m`$_1$ and `!m`$_2$), while process `Q` has two message-receive statements `b?m` (whose requests will be represented by `?m`$_1$ and `?m`$_2$). The following matches can then be made: `!m`$_1$ with `?m`$_1$, `!m`$_1$ with `?m`$_2$, `!m`$_2$ with `?m`$_1$ and `!m`$_2$ with `?m`$_2$; computing these matches is exactly what channel-tree node `c` does. For each match, node `c` issues a *silent request* that becomes registered at the scheduler, as depicted in Figure 6.2 on the right. When the scheduler grants such a silent request, the request will notify both the message-send and message-receive statement it represents.

After calculating all matches, channel-tree node `c` will forward the message requests it manages (that is the requests originating from port `a` and `b`) to its parent, node `d`, which will try to match them with requests it got from port `e`. So, each non-leaf channel-tree node stores references to the message requests it gets from its children and creates silent requests for the matches it finds.

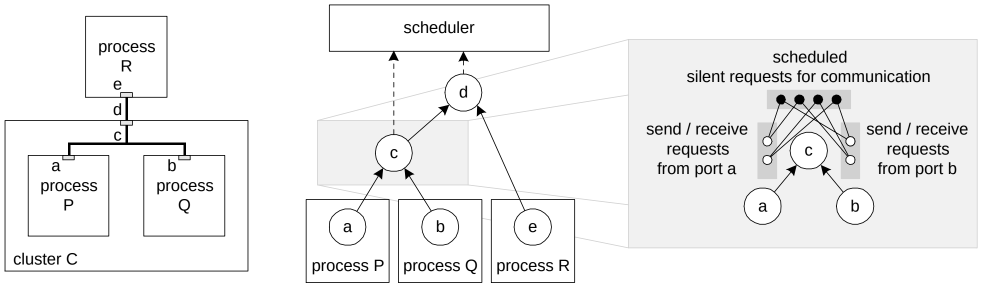


Figure 6.2: Processes `P`, `Q` and `R` in the model on the left are connected through ports `a`, `b` and `e` to the channel formed by parts `c` and `d`. In the simulator (middle image) the channel is represented by a channel tree. The right image shows that channel-tree node `c` computes silent requests from send/receive requests forwarded by port nodes `a` and `b`.

From a computational point of view, the above implementation of communication primitives can be expensive. First of all because each message request that ceases to exist must notify all the objects referring to it (such as channel-tree nodes and silent requests); and second —more importantly— because of the worst-case overhead of creating $n \times m$ silent requests for n equal message-send statements matching m equal message-receive statements. In that case $n + m - 1$ silent requests are created and destroyed for at most one successful rendez-vous.

Preventing Explicit Construction of Requests for Matches The overhead introduced by explicitly creating silent requests for each possible match can be very high. Obviously, there must be searched for better approaches. The algorithm has therefore been adapted so that matches are no longer computed beforehand, but only when a message-send request is about to be executed — now without explicit construction of silent requests for matches. To allow this approach to work, message-send nodes of an execution tree issue a request at the scheduler. Since a message-send node knows to which port its message should be sent, it can ask the corresponding channel tree to gather the matching message-receive requests. The message-send node then randomly picks one of the matching requests and tries to communicate. If communication is unsuccessful because the receiver's reception condition evaluates to *false*, other requests are tried either until communication does take place or until all requests have been tested.

Like the previous algorithm, each channel-tree node stores references to the currently active message requests. However, only message-receive requests are sent to the channel tree; message-send requests are registered at the scheduler. For gathering matches, the message-send node asks the corresponding port to generate a list of matching message-receive requests. The port will redirect this request to its parent (a channel-tree node), which will add all requests matching at that hierarchical level. The parent's parent will then add matches at the next hierarchical level, and so on, until all matching requests have been collected.

Although this algorithm does not generate explicit silent requests, it still has to (repeatedly) find all matching receivers. The time spent looking for matches depends on both the hierarchical depth of the model (determining the height of the channel tree) and the number of message-receive requests active at that moment. Because the channels are static, the following optimisation is possible.

Removing Hierarchy The fact that communication between two ports is allowed or not, is determined by the topology of channels and since this topology is static, this fact will not change during a simulation. The channel tree can therefore be replaced by a single node (representing the entire channel) that stores a square *connection matrix*. The matrix holds *true* at position (x, y) if the channel between ports x and y allows communication and holds *false* otherwise.

The channel uses a single list to store all current message-receive requests. The memory and computational overhead of storing references to a request at each hierarchical level of the channel tree is thus reduced to storing only a single reference per request.

Table 6.3: Example to demonstrate the need for message representatives.

Process P:

```
  1   par
  2       p!m
  3   and
  4       p!m
  ⋮    ⋮
2n+1  rap
```

Process Q:

```
1  while true do
2      p?m
3  od;
```

Also the search for matching requests no longer depends on the hierarchical depth of a channel: only a single list is to be searched and a simple lookup determines if the channel allows communication.

Two problems still remain; they are dealt with by the next (final) matching algorithm.

Message Representatives The algorithm that prevents explicit construction of requests for matches still has two deficiencies: it is asymmetrical and it does not use all static information that is already available at compile-time and which could be used for reducing the work at run-time. The algorithm's asymmetry expresses itself clearly when many transmitters try to communicate with but a few receivers.

Consider n concurrent transmitters trying to communicate with n receivers sequentially (Table 6.3). Because the transmitters of process P are added to the scheduler's request list, n message-send requests occupy this list, next to an action request to unroll the while-loop of process Q once. To make progress, the scheduler must find that action request, because until that moment, no matching receiver is available to render the message-send requests executable. After that, the scheduler picks the next request and will find a transmitter that can communicate. Immediately after that, $n-1$ blocking requests are left in the list, next to an action request to unroll the while-loop once more.

If the message-send statements (lines 2,4,. . .) were to be replaced by message-receive statements and, conversely, the message-receive statement in line 2 by a message-send statement, the scheduler's queue would initially contain only a single action request to unroll the while-loop once. After the scheduler has executed that request, process Q issues a single message-send request. The scheduler has but one possible next step, which is executable. Then, another action request is scheduled for the next iteration.

In the original example (Table 6.3), the scheduler had to find the action request buried within $n, n-1, n-2, \ldots$ (blocking) message-send requests to perform a communication, slowing it down. In the second case, after swapping the message-send statements and the message-receive statement, the request list contained only a single executable request at each step — hence the scheduler would immediately find the request to make progress.

This asymmetrical behaviour of communication primitives is unacceptable — it could force modelers to use a message-send statement where they would normally have used a message-receive statement, only to prevent a severe slow down in execution speed.

A more efficient approach is highly desirable and also feasible: schedule only a single *message representative* when matches have been found.

Figure 6.4 shows the data structure that each channel employs to render communication efficient. For each different combination of message name and parameter count, the channel has a message representative accompanied by two circular doubly-linked lists. These lists store the requests of corresponding message nodes — one storing only message-send requests, the other storing the message-receive requests. The message representative issues a request at the scheduler iff both its lists contain a request. The crux of this symmetrical(!) message-matching algorithm is that the scheduler's request list contains only a single request (the message representative's) when matches exist, independent of the number of message requests.

Inevitable Run-Time Checks When a message representative issues a request at the scheduler, it cannot guarantee that communication will occur. Two final checks can still thwart successful synchronisation: testing if a pair of executable message requests exists, and testing the optional reception condition. These checks must be computed at run-time since they depend on the corresponding processes' dynamic data that may change while the model is executing.

To reduce the overhead of these run-time checks, a message-send request is paired with any of the available receivers before proceeding with the next message-send request. This allows for single evaluation of the message parameters. Only when one of the receivers permits synchronisation will the message's data be deepcopied to the receiver. Then —at long last— has the communication been settled.

Measured Cost of Communication Figure C.1 in Appendix C shows that the time required for communication over a separate channel between a pair of a transmitting process and a receiving process is independent of other concurrently running processes. Figure C.2 demonstrates the constant cost of communicating through hierarchical levels. One of the communicating partners is a process wrapped into clusters

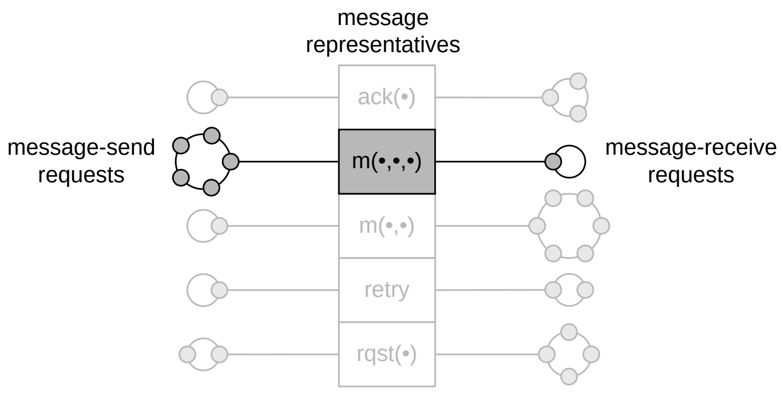


Figure 6.4: A channel node uses a list of message representatives to represent equally typed messages. The message representative issues a request iff communication might be possible.

(introducing the hierarchical levels), the other is a process at the top level. Communication cost is also independent of the hierarchical depth, as is shown by Figure C.3. In this test, both the transmitting and receiving process are wrapped into clusters.

The communication algorithm is symmetrical: synchronising a single transmitter with one out of N receivers takes the same amount of time as synchronising a single receiver with one out of N transmitters (Figures C.4 and C.5). The first test constructs a single transmitting process, connected via a single channel to N receiving processes, whereas the second test has N transmitters and one receiver connected to a single channel. The measurements show that the cost is independent of channel connectivity for (nonblocking) communication. The measurements in Figures C.6 and C.7 show the execution time of synchronising with one out of N concurrent communication partners within a single process. The reader is reminded that the increase in execution time and noise are caused by slower memory access for large amounts of memory and object destruction respectively.

A deviation on the last two tests is by using conditional receive statements. Only one of the N communication pairs matches, the other communication statements block. The communication cost (Figures C.8 and C.9) is independent of the number of blocking partners. This constant execution time is the result of using message representatives — otherwise, the time per communication would have been linearly dependent on the amount of blocking partners.

When no run-time checks are required, the overhead of communication is (almost) independent of the amount of message statements in a POOSL specification; almost, because the number of different combinations of message name and parameter counts determines the amount of message representatives that can become active at a single moment. It is safe to say that this matching algorithm scales very well for unconditional messages.

6.4 Delays

The semantics of POOSL dictates action urgency. The scheduler must therefore perform action transitions before it may schedule a delay transition. When none of the action requests is executable, the scheduler lets the model time proceed to the next moment at which action transitions may have become executable. To determine the next point in time, the scheduler calculates the minimal delay from its list of delay requests, increases time by this minimum and grants the expired delays.

Initial Approach The initial algorithm that determines the next moment in model time, used a list to store delay requests. When a delay statement issues a delay request, the scheduler evaluates the expression of the delay statement to determine the time period after which the statement will expire, and stores this initial duration in the request. If the model time progresses when a delay statement is blocked, the moment of expiry of the corresponding delay request is prolonged. When the scheduler is about to perform a time transition, it determines the minimum of the durations that are stored in the delay request list. The durations are then lowered by this minimum and delay requests that expire because their remaining duration becomes zero are moved to a separate list for convenient retrieval. Finally, the scheduler adjusts model time, grants the expired delay requests and proceeds with scheduling action transitions.

The scheduling overhead is proportional to the amount of delay requests in the list. In the worst case, each delay request determines a unique point in time, introducing adjustments to the delay requests that do not expire (Figure 6.5).

Special Case: Invariable Delays Often, delay statements are neither guarded, nor interruptible. For such statements, the duration of the corresponding delay request is *invariable* from the moment it is issued, and therefore its expiry time can be precalculated. By sorting the expiry times of these invariable delay requests, the scheduler can quickly find the next instant in time and make a delay transition.

The scheduler uses a *binary heap* to store invariable delay requests. Elements in a heap are stored in such a way that the smallest or largest value is accessible in $O(1)$ (constant time) at the root of the heap. Adding or removing elements can be done in $O(\log n)$ where n is the size of the heap. Since requests are usually added and removed only once in their lifetime, the performance of this algorithm is mainly determined by the fast retrieval of the minimum.

Because absolute expiry times are stored, and not relative durations, the delay requests require no adjustments as was the case in the initial algorithm. This algorithm performs significantly better, because the invariable delay requests do not need adjusting, something that caused a large overhead in the initial approach. *Variable* delay requests originate from delay statements that are guarded or interruptible; they are still handled as in the initial approach. Luckily, in many models these statements are simply absent.

Measured Cost of Delays The actual performance of the discussed algorithm for computing the next time transition for guarded delays is shown in Figure C.10. The cost per time transition is, as is expected, linearly dependent on the amount of concurrent guarded delays. The use of *true* guards will prevent the execution engine from

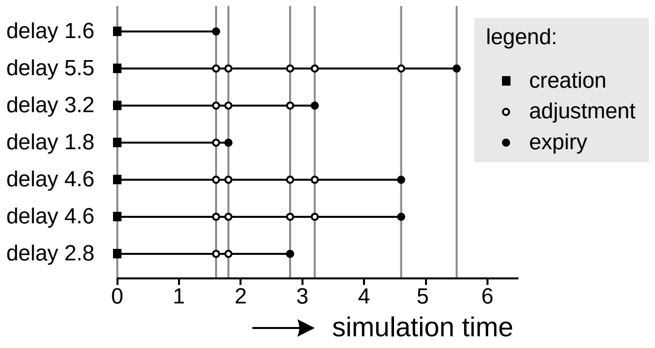


Figure 6.5: Creation, adjustments and expiry of concurrent delays.

using the binary heap. Figure C.11 shows the time required for computing the next time transition if the model contains only invariable delays. The cost is constant (taking into account the increased memory access time, as discussed in Section 6.2). Using the binary heap for storing expiry times of invariable delays dramatically improves the execution speed over normal handling of delays.

6.5 Guards

Guarded statements can only execute if their guards evaluate to *true*. The implemented approach assumes that since the last transition, reevaluation of a guard is necessary (the scheduler only evaluates a guard if it encounters one). This is costly, but the alternative of keeping track of possible changes in the outcome of a guarding expression is generally even more expensive.

A guarding expression usually depends on data that changes during simulation, so it is often impossible to reduce the overhead of guards by trying to calculate if an execution step has influenced the outcome of a guard. To see this, consider the simple case of `[x m()]` S, where `x` is a data object, and m is a method. POOSL offers dynamic method binding and is an untyped language, so it is only at run-time that the corresponding method body is known. The outcome of the guarding expression `x m()` can only be determined if it is known of which class `x` is an instance, such that the correct body of method `m` can be found. The method may depend on other data objects as well (via the instance variables of `x`), so to determine whether the outcome of method `m` has changed, the execution engine should also keep track of the changes in these objects. This bookkeeping may all be without purpose, because `x` could be reassigned to a different kind of object before the scheduler encounters the guarded statement.

In general the objects determining the outcome of a guarding expression cannot be computed efficiently, and it is therefore infeasible to detect when a guard has changed and the scheduler cannot use the previous outcome of a guarding expression. Although the real problem cannot be solved, we can alleviate it a bit.

To determine whether the request of a statement is guarded, the scheduler traverses the corresponding execution tree from the leaf node that issued the request towards the root, looking for guards. Because most statements are not guarded, the scheduler can save a lot of time if this information is available in the request itself. When a request is unguarded, the scheduler can immediately grant it without traversing the tree. In the unfortunate case that the request is guarded, the scheduler proceeds as normal.

Another optimisation seems to be storing the result of an evaluated guard, and use that result as long as it is not invalidated. Because it is impossible to determine which changing data object can affect the result, the scheduler can only assume the result to be valid until the next transition is taken. Because processes do not share any data, only transitions of the corresponding process might affect the guard. The overhead of storing and invalidating results does not seem to be worthwhile in general.

Blocking guards greatly influence the performance of the simulator. When the amount of blocking requests is large compared to executable requests, the scheduler probably encounters many guards and must evaluate their guarding expressions before it actually finds an executable request to perform a transition. The higher the percentage of blocking requests is, the more probable the scheduler is to encounter them and spend time evaluating their expressions.

Measured Cost of Guards The measured cost of guards as a function of their count confirms our suspicions. Figure C.12 shows the time required to find a single executable request hidden among N blocking guards. The speed degradation is linear in the ratio of blocking guards to executable requests. A second test with guards measures the cost guards impose when they are nested. The test repeatedly executes a statement of the form `[true]...[true] skip`. The graph in Figure C.14 shows that the cost linearly depends on the depth of nesting. The cost is primarily caused by having to evaluate the guarding expressions for determining whether the guarded command is executable. Another part of the cost is caused by the increased depth of the execution tree, which is the topic of the following section.

6.6 Execution Tree Height Reduction

When a process performs a transition, its execution tree must adapt itself to reflect the process' next state. The overhead of changing the execution tree depends on its height, because the executed request that caused the transition sends a message along the path from the request to the root. Each node along this path will then decide, based on its semantic rules, how it must react. The shorter the path, the less nodes that process the requests' message and the lower the overhead for performing a transition is.

There is a second reason for keeping the path from request to root short: guards. When a guarded request is granted, the request must evaluate its guards to determine if it can execute. Along the path towards the root, starting at its parent, the request searches for its guards. The overhead of searching for guards is lower when less nodes have to be visited. This can be achieved by reducing the height of the execution tree.

Table 6.6 shows an example of a method that dynamically creates N concurrent activities S. The parameter of the initial method call specifies the number of concurrent activities that `m()()` should start. In the method, the parameter is interpreted as an identifier of the activity S in line 5. In parallel, the method tail-recursively creates the remaining activities. The recursion ends when `ID` becomes one.

Table 6.6: Dynamically creating concurrent activities.

```
1  m(ID: Integer)()
2  par
3      if ID > 1 then m(ID - 1)() fi
4  and
5      S
6  rap.
```

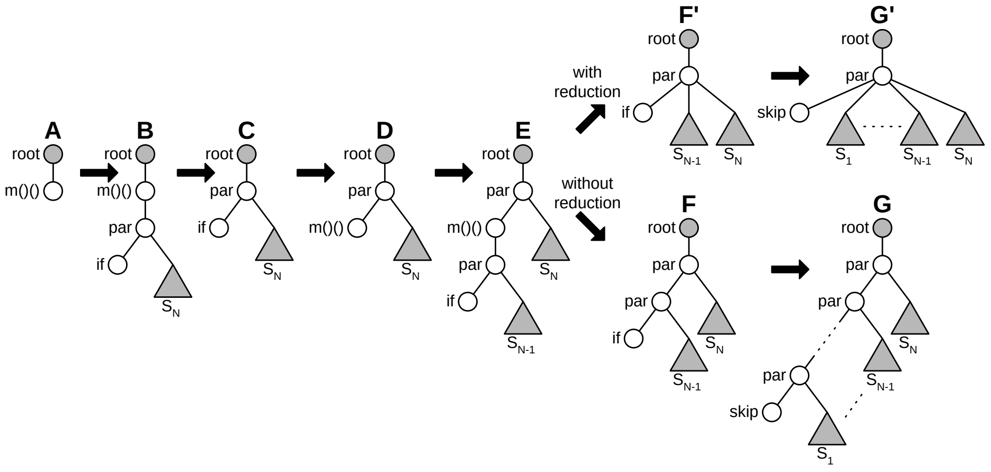


Figure 6.7: Subsequent trees during dynamic creation of concurrent activities.

Figure 6.7 represents a possible trace of intermediate trees while `m()()` unfolds. Tree **A** shows the initial method call (with parameter N). The method-call node creates its body (**B**) and has its parent —the root— replace the method-call node by its body (**C**). The if-node representing the `if` in line 3 replaces itself by a new method-call node (**D**) that creates its body in **E**. Then, the method-call node has its parent (a parallel-composition node) replace the method-call node by its body (**F**). Recursion ends in **G** where the if-node has just replaced itself by `skip` (representing its else-branch). The tree is unbalanced, and the average path length from the root to a subtree S_i is $\frac{N+1}{2}$ nodes.

For compound statements such as parallel composition, sequential composition and selection, the tree can be reduced (no proof is provided, this is future work). When the method call node in **E** is replaced by its body, the tree can thus be reduced to **F'**. When recursion ends in **G'** the average path length from the root to a subtree S_i has been reduced to only one node.

Measured Cost of Execution Tree Height Figure C.17 shows the time required for running a task at a certain depth of the execution tree. When the depth increases, the execution time scales proportionally. Nonterminating statements that run after their fix requests have been granted, however, have a run-time that is independent of the depth at which they are positioned in the tree (Figure C.15). The reason for this constant complexity is that fix requests are not communicated to other parts of the execution tree, because fix transitions do not modify the statement context.

The graph in Figure C.16 shows that returning from a method (with return parameters!) exhibits a complexity linear to the depth the method call node is positioned at. This may become a problem, as was experienced during a stress test: if a statement nested in N method calls terminates, it takes $O(N^2)$ to destroy the tree, because each terminating method call must inform its parent that it has terminated. The parent (also a method call) will issue a request at the scheduler for binding its output parameters, and will inform its predecessors that its child has terminated. A possible optimisation would seem to allow the parent to immediately bind its output parameters without requesting permission to the scheduler. In that case, the entire tree could be terminated in one pass, yielding a total complexity of only $O(N)$ (that is, a constant cost per terminating method). Such an optimisation should be backed by a proof showing that the resulting behaviour is bisimilar to the original behaviour. In this case, it is easy to produce a counterexample.

The effect of tree reduction on execution trees with dynamically created concurrent activities is shown in Figures C.18 and C.19. The former graph shows the time required for executing a task (a `while` loop) at varying depth in the execution tree. For trees with a modest depth (less than hundred nodes), the execution speed is predominantly determined by other overhead, but for a depth of over a thousand nodes, the execution time depends linearly on the depth of the tree. With the tree reduction proposed in the previous section, the execution speed becomes independent of the depth of the tree (Figure C.19).

6.7 Industrial Case Study: Internet Packet Switch

The previous sections have discussed several optimisations separately, but real-life models will usually contain a motley collection of language primitives. To demonstrate the combined effect of the accelerations, this section presents the execution speed of an industrial model of a packet switch as a function of its scale parameter N, for various loads. Because the model itself is confidential, a greatly abstracted model is presented for theoretical analysis of its complexity (and validating the measured execution time).

6.7.1 Abstract Model of the Packet Switch

Figure 6.8 shows the conceptual model of a packet switch that routes IP traffic over the backbone of the Internet. This model has been derived from a model used for performance analyses of a large industrial case study that is fully described in [47, 48]. Nonconfidential excerpts can be found in [46, 45]. The size of the switch can be changed by varying parameter N, which determines the number of input and output ports of the switch. Parameter M determines the number of independent sources offering traffic on a single input of the switch.

The conceptual model simulates the environment that produces IP traffic (the sources) and consumes it (the sinks). For each of the N inputs, M independent sources produce a *bursty* stream of IP packets of varying length. Each source uses a two-state Markov mechanism to model transferral of files, alternated by bursts of small packets. Incoming traffic goes to input adapters where it is buffered. The conceptual model represents the input adapters by one process that uses a single input dispatcher to place packets in one of the $N \times N$ input buffers. The switch fabric transports the buffered packets to their respective output adapters where buffers flatten out temporary peaks in traffic. Packets leaving the Internet packet switch finally disappear in the sinks.

Four methods (Table 6.9) deserve special attention because they exemplify the scalability issues encountered in earlier implementations of the simulator.

Sources Process `Sources` runs $N \times M$ methods `TransmitPacket` in parallel. Each method uses an infinite loop to produce packets. `CreatePacket` (line 5) returns a packet and calculates the remaining time until the next packet is to be produced. The packet is sent (line 6) to the input adapter along with information that identifies the input port (`srcID`). Since each packet carries its own destination, the switch can transport the packet to the correct output port. Finally, the source waits for the next moment to send a packet (line 7).

Input Adapters Process `InputAdapters` uses a single method `DispatchInput` to route incoming packets to one of the input buffers. In line 4 a packet is received and stored —with an immediate data expression— in global variable `Queues` that refers to a matrix of $N \times N$ FIFO queues. Each input and output pair has its own FIFO queue to buffer incoming packets (virtual output queuing [27]); the destination stored in the packet and the source (`srcID`) determine in which buffer the packet is stored. The input dispatcher uses a tail-recursive method call (line 5) to process the next packet.

Every input buffer is emptied by an output handler (method `HandleOutput`) as follows. The guard on line 3 blocks while the buffer is empty. As soon as a packet is present, the message-send statement (line 4) will send it to the switch fabric. Because the parameter expressions of a message-send statement must be side-effect free, the packet is retrieved with data method `inspect`, which leaves `Queues` untouched. Immediately after the communication has taken place, the packet is removed by the message's immediate data expression. The output handler uses tail-recursion (line 5) to transfer any remaining packets to the switch fabric.

Switch Fabric Process `SwitchFabric` always reserves one method `Transfer-Packet` to await (line 4) a packet from any of the input adapters. After receiving a packet, the method splits up into two concurrent activities, one that handles the packet and another that awaits additional packets. Packet handling is modelled by delaying for a constant time `SwitchingTime` after which the packet is sent to the appropriate output adapter.

To give an idea of the size of the model, we mention that most of the simulations where performed with typical values for N ranging from 32 through 64 and M usually being 16. This amounts to approximately[3] 2,500 through 9,200 concurrent activities. Larger switches being simulated were performed with N in excess of 256 and M equal to 64, amounting to over 147,000 concurrent activities.

[3]The approximations are based on high-load traffic, resulting in few empty input buffers. In this typically measured situation the switch fabric is running at peak performance and has $N \times N$ branches to handle packet transport, amounting to a total of about $N \cdot M + N \cdot N + N \cdot N$ concurrent activities for the entire model.

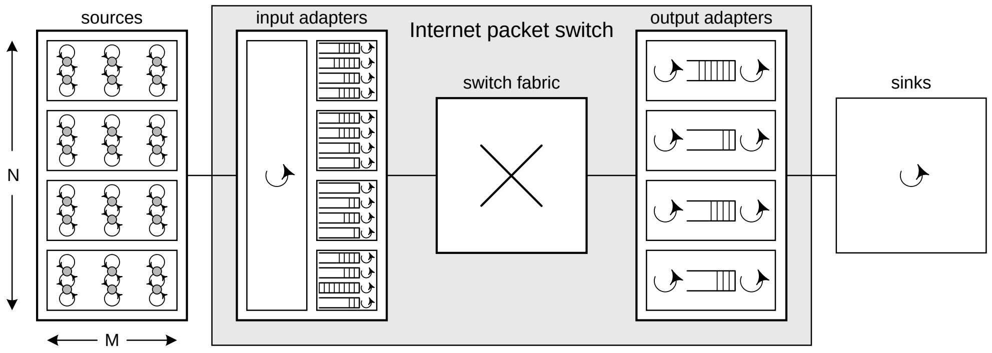


Figure 6.8: Internet packet routing switch.

Table 6.9: Methods of the Internet packet switch that cause scaling problems.

Process class `Sources`:

```
TransmitPacket(srcID: Integer)()
| p: Packet, t: Real |

while true do
    CreatePacket()(p, t);
    out!packet(srcID, p);
    delay t
od.
```

Process class `InputAdapters`:

```
DispatchInput()()
| srcID: Integer, p: Packet |

in?packet(srcID, p) {Queues put(srcID, p)};
DispatchInput()().
```

Process class `InputAdapters`:

```
HandleOutput(srcID, dstID: Integer)()

[Queues notEmpty(srcID, dstID)]
    out!packet(dstID, Queues inspect(srcID, dstID)) {Queues remove(srcID, dstID)};
HandleOutput(srcID, dstID)().
```

Process class `SwitchFabric`:

```
TransferPacket()()
| dstID: Integer, p: Packet |

in?packet(dstID, p);
par
    delay SwitchingTime;
    out!packet(dstID, p)
and
    TransferPacket()()
rap.
```

6.7.2 Optimisations

Many test cases have been used during the development of rotalumis, to check the effect of the optimisations discussed in this chapter. The packet-switch model is one of those test cases that showed the need for optimisations to obtain scalability. It is illustrative to examine the model and discuss those parts that would blow up without these accelerations.

Communication The case study confirmed that the initial algorithm for finding matching communication statements does not scale well. The $N \times N$ output handlers of the input adapters trying to communicate with the switch fabric cause a slowdown dependent on the size of the switch. For typical values of N this overhead was dominating for simulation speed: changing the switches' size from 32 ports to 64 ports would increase the simulation time sixteen times (doubling the size quadruples the workload, and the number of silent requests constructed and removed per synchronisation is also four times larger).

The second implementation for communication prevented the construction of requests for each possible combination, but it was asymmetric. We will now consider the $N \times M$ (say k) sources trying to synchronise with the single input dispatcher of process `InputAdapters`. We conveniently discard the other processes and abstract from time and message data: line 6 of method `TransmitPacket` then contains `out!packet;` and line 4 of method `DispatchInput` becomes simply `in?packet;`. (We will call this the first case.)

Now consider the situation where line 6 of method `TransmitPacket` is replaced by `out?packet;` while line 4 of method `DispatchInput` is changed into `in!packet;`. (This is the second case.) Both processes still synchronise as expected; functionally, nothing has changed. However, this subtle and seemingly innocuous alteration does boost simulation speed by a factor of up to k (actually, in the first case performance was up to k times too slow and has now been restored).

What has happened? The first case has a large scheduling overhead because of the at least $k-1$ transmitters trying to synchronise with a single receiver. Because message-send nodes issue a request at the scheduler, there are equally many blocking requests whenever a communication has just taken place. For the subsequent transition, the simulator will have to find the single request that allows the input dispatcher to perform a method call and issue the next message-receive request at the channel. Only then will the message-send requests become executable and can the next synchronisation take place. Actually, the scheduler can successively perform several other requests, namely those originating from the source that just has communicated (being related to `TransmitPacket`'s lines 4, 5 in this order), but eventually this source blocks at line 6 leaving only a single executable request: that of the receiver. Most of the time the scheduler picks inexecutable requests and has a low probability of finding executable requests; whence the large scheduling overhead (proportional to k).

In the second case (many receivers, single transmitter) this overhead is not present, because the k message-receive nodes issue their requests at the channel tree. The request list of the scheduler now contains only executable requests and, consequently, the simulator runs at full speed. As explained before, this asymmetrical behaviour of communication primitives is unacceptable; in the case study it slowed down execution speed approximately 500 to 1000 times ($N \in \{32, \ldots, 64\}$ and $M = 16$).

Because rotalumis employs the implementation with message representatives to match communication primitives, there will not be scheduled any blocking communication requests in this case.

Delays The sources in Table 6.9 exhibit the worst-case behaviour mentioned in Section 6.4 that delays have to be adjusted often. This behaviour is introduced by the delay requests (from the delay on line 7) of the $N \times M$ concurrently running methods `TransmitPacket`. Because the sources are statistically independent, it is likely that delay requests are started at different model times, and will end at different model times. In this situation, performing delay transitions is expensive: for each delay transition, the scheduler must adjust the almost $N \times M$ delay requests. For typical values of $N \in \{32, \ldots, 64\}$ and $M = 16$, the overhead of adjusting 500 to 1000 requests for each delay transition would become problematic with the initial approach. However, because the delay statements are unguarded, they will be invariable from the moment their expression has been evaluated — the execution engine will employ the faster approach with the binary heap.

Guards Consider the guard in line 3 of method `HandleOutput` in Table 6.9. Method `notEmpty` of data object `Queues` (a matrix of FIFO queues) is called to check if the buffer at position (`srcID`, `destID`) contains any packets. Section 6.5 explained that guards are expensive. Since the packet switch has $N \times N$ input adapters, there may be equally many blocking guards. In such a situation, the scheduler will encounter many blocking requests while trying to find the next executable transition. Here, only rewriting the model to avoid using this many guards will help — the actual packet switch model creates a new concurrent activity to handle output only if there is a packet in the corresponding queue, using a method similar to the one discussed in Section 6.6.

6.7.3 Actual performance

The theoretical behaviour is of order $O(M \times N + N \times N)$. Since N determines the dominant factor, we will vary that parameter while keeping M fixed at its typical value sixteen. The theoretical complexity of the model is then of order $O(N + N^2)$. For lower values of N, the execution time of the model should scale proportionate to N, but for larger values of N the execution time will be of order $O(N^2)$.

The theoretic behaviour is reflected in practice; the execution time (for simulating one second) for the actual packet-switch model is shown in Figure 6.10.

For $N \leq 32$ the execution time is almost linearly dependent on N, while for $N > 32$ it scales with $O(N^2)$. The slight additional increase in execution speed is about a factor of two for $N = 1024$, and is caused by slower memory access (see Figure C.26 in Appendix C) — at that point approximately two million concurrent processes are active. Notice that for higher loads, the amount of data in use by the model is higher, and thus memory access is slower.

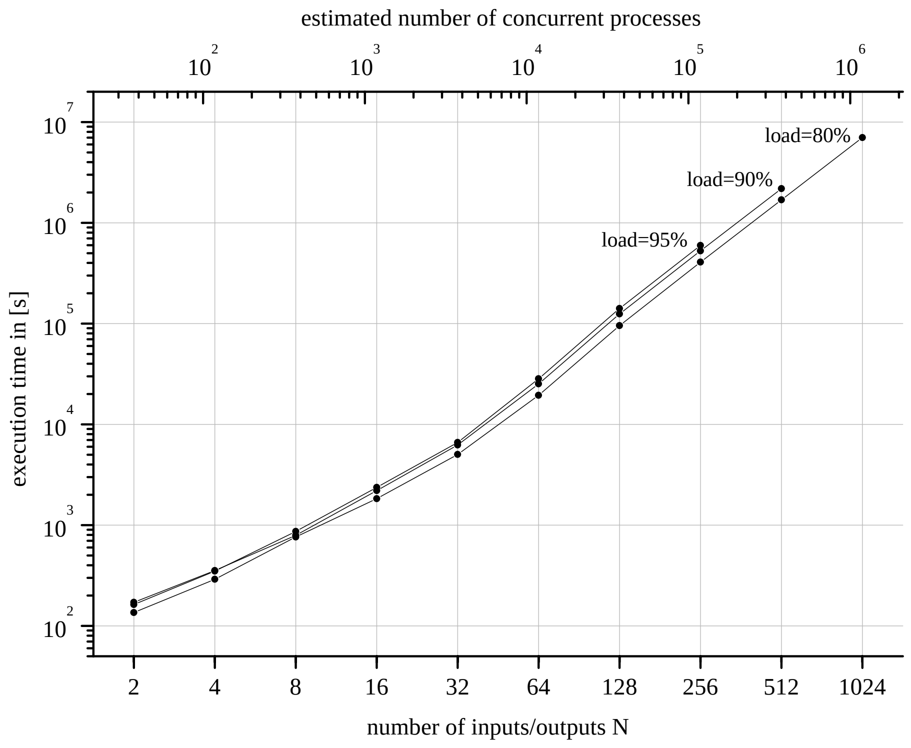


Figure 6.10: Execution time of the Alcatel packet switch model (for simulating one second).

6.8 Summary

To offer designers the full power of the language, an effort has been taken to optimise the execution speed of (combinations of) the language primitives. Industrial problems tend to be of such large dimensions that omission of these speedups would result in extremely long simulation times that it would render the execution engine unusable.

The most significant source of these scalability problems is formed by blocking statements. After identifying and discussing the different kinds of blocking statements, the implemented optimisations were explained, which alleviate most of the distress these blocking statements cause.

An important speedup has been achieved for communication primitives, using both information that is available at compile-time (such as channel topology) as well as information available at run-time (storing messages in categories based on their signatures, for efficient matching).

After having identified two kinds of delays —variable and invariable— an enhanced algorithm was devised that allows efficient handling of invariable delays whose expiry time can be precalculated. The required adjustments after each time transition for variable delays (which are either guarded or interruptible) remain because they can not be averted.

The main culprit is and remains the `guard`. Its semantics requires evaluation of the guarding expression before the scheduler can decide whether a guarded statement is eligible for execution. Combined with the action urgency requirement of POOSL's semantics, the scheduler must evaluate all guards before it can decide whether to let model time pass by. When the amount of blocking statements becomes relatively large, the execution speed may become so slow that it becomes problematic; in that case the modeler is required to restructure the model to either minimise or avoid the use of guards. Further research into the use of guards may lead to speedier execution of guards in special cases.

Dynamic restructuring of execution trees allows reducing their height, thus improving the simulation speed, which is related to the path length from request (leaf node) to root.

The measurements presented in this chapter show that rotalumis is a scalable, efficient execution engine for POOSL.

Chapter 7

Prototyping

7.1 Overview

The simulator can be adapted for prototyping. This allows a model to control hard- and software and to react to actual events from the outside world while satisfying timing constraints. Figure 7.1 gives an overview of how communication with the environment (real world) can be accomplished. Processes can use data objects to encapsulate part of the environment, for instance a hardware device such as a computer display. Such data objects define a clean interface towards the model (their instance methods), and internally use a device driver to access or control the hardware. The device driver could be as simple as a set of library functions that allow interaction with software. The next section discusses a case study to illustrate the design alternatives that were encountered during the implementation of the real-time variant of the simulator (rotalumis-rt).

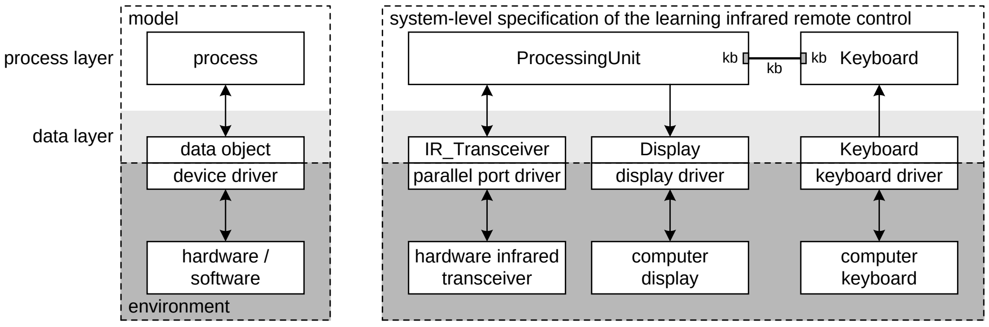


Figure 7.1: The model (left image) can interact with the environment through data objects that encapsulate external hardware devices or software and use device drivers to access or control those devices. The image on the right depicts the example that will be explained in the next section.

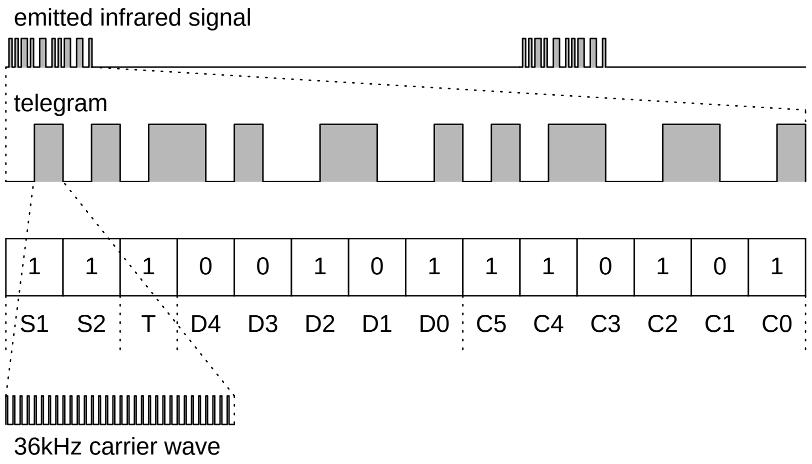


Figure 7.2: Example of a signal transmitted by an RC-5 compliant remote control of a video recorder while pressing its **play** button. From top to bottom: the entire signal, a single telegram, the encoded bit pattern and finally the carrier wave.

7.2 Case Study: Learning Infrared Remote Control

A case study has been carried out to discover the common difficulties in real-time simulation. The case study encompasses the implementation of a prototype of an infrared remote control with learning capabilities. A short introduction to existing protocols for infrared remote controls might help the reader in better understanding the specification of the learning infrared remote control (LIRC).

7.2.1 Protocols for Infrared Remote Controls

Remote controls can use different protocols to transmit information by means of infrared light; examples of such protocols are RC-5 and SIRCS. Figure 7.2 shows an example of the signal transmitted by a video recorder's remote control —compliant with the RC-5 standard— when its **play** button is pressed. The signal is a carrier wave of 36 kHz whose amplitude is modulated by a bitstream of 1125 Hz. The bitstream that toggles the carrier wave on and off is a repetition of a *telegram*, which is an encoded version of the message that is being transmitted. The message contains a command and an identifier that indicates which device should react. It is phase-shift encoded; each bit in the message is translated to a pair of bits in the telegram ($0 \longrightarrow 10$ and $1 \longrightarrow 01$). The original message (00101110101) contains five bits to identify the device (00101 for **video recorder**) and six bits for the command (110101 to select **play**). The telegram begins with two start bits (always 1), followed by a toggle bit that flips each time a key is pressed. Then the five device bits are sent, most significant bits first, followed by the six command bits. Each 113.778 ms the telegram is repeated.

Other remote controls using different protocols exhibit similar features, but may differ in for instance number of device/command bits or use different encodings, like pulse-length coding or space-length coding instead of phase-shift coding.

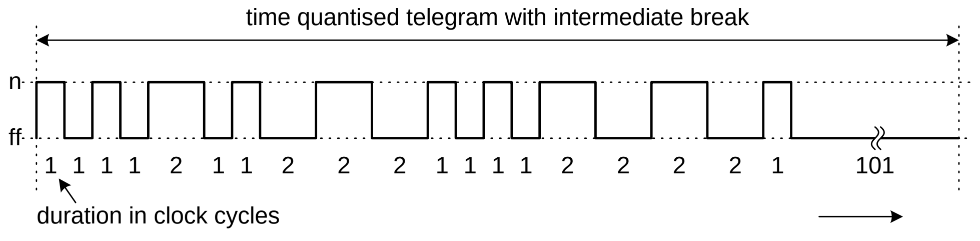


Figure 7.3: The LIRC will be restricted to deal with telegrams of this form.

7.2.2 Specification of the Learning Infrared Remote Control

The following specification is not intended to be complete and precise, but merely to give a general idea of the prototype that serves as an example to explore real-time simulation.

The learning infrared remote control (LIRC) will be restricted to handle telegrams with the following characteristics:

- the smallest duration between two signal transitions defines the clock frequency; other durations are a multiple of this duration;
- telegrams are repeated at a constant frequency and repetitive telegrams are identical.

The telegrams will be sent using a 36 kHz carrier wave. The LIRC should be capable of mimicking signals that were sent by remote controls producing similar telegrams. The LIRC should be fitted with a display to give the user feedback or instructions. A keyboard allows the user to communicate with the remote control's internal processor that performs signal (de)composition and storage and controls the display and an infrared transceiver.

Figure 7.3 shows the same[1] telegram as in Figure 7.2, but now viewed as a collection of transitions separated by an integer number of clock cycles. The clock frequency of the bitstream defined by RC-5 is 1125 Hz.

7.2.3 Implementation

The prototype consists of a hardware infrared (IR) transceiver controlled by a POOSL model that is being executed by the real-time variant of the simulator on a personal computer. The following sections will discuss the common problems that were encountered: synchronising model time with real time, reacting to events from the environment (for example from sensors) and producing output (for instance to control actuators).

The carrier wave will be generated by hardware (a prototype IR-transceiver) to relax the timing constraints for the real-time simulator. The IR-transceiver (Figure 7.4) also contains a demodulator to recover the enveloping signal that might contain telegrams

[1]The learning remote control assumes that the first off-to-on transition defines the beginning of the telegram. The telegram in Figure 7.2 actually starts one clock cycle earlier, but the asynchronously operating receiving unit (in this case the video recorder) will not be able to notice the difference.

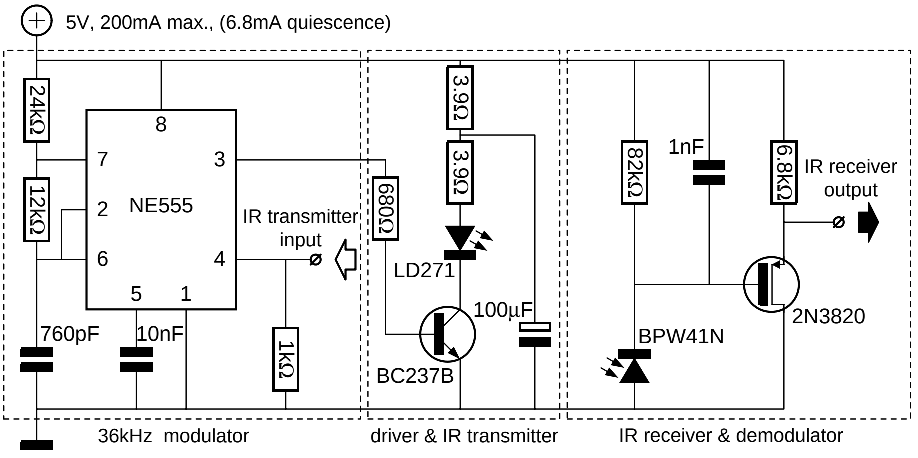


Figure 7.4: The electrical scheme of the hardware infrared transceiver.

sent by another remote control. The rest of the remote control is modelled in POOSL as is shown on the right in Figure 7.1. The architecture of the model contains two processes: one modelling a keyboard and the other modelling a processing unit with display and infrared transceiver. The keyboard is connected to the processing unit by a channel allowing it to offer the processing unit information on pressed keys.

As was mentioned before, the processing unit uses two data classes, `Display` and `IR_Transceiver`, to encapsulate the actual devices. `Display`, for instance, contains the primitive method `writeln` whose parameter is a string that will be displayed on the monitor of the computer the model is running on. Data classes can be viewed as wrapper classes whose primitive methods can encapsulate functions in a native programming language.

For the actual transmission of infrared signals, the hardware IR-transceiver connected to the parallel port of the computer (LPT1) must be controlled from within the POOSL model. Data class `IR_Transceiver` represents this hardware IR-transceiver. For transmission only one method is of real interest: method `Out`. The parameter to this method is either zero or one, to turn the IR-transmitter off and on respectively. When the transmitter is switched on, it will begin sending the carrier wave generated by its oscillator.

Two aspects of the model deserve special attention because they reveal the general problems of performing input and output from a system-level model while satisfying timing constraints: transmitting telegrams (output) and learning new telegrams (input). The methods will only be introduced here and discussed more thoroughly in the subsequent sections.

Process class `ProcessingUnit` offers method `TransmitTelegram` (Table 7.5) to send telegrams to the infrared transceiver. The method reads the telegram characteristics from a file, starting with the clock frequency, followed by the amount of transitions

Table 7.5: Method `TransmitTelegram` of process class `ProcessingUnit`.

```
TransmitTelegram(FileName:  String)()
| FSamples:  FileIn,
  i, NrOfTelegramsSent, NrOfTransitions, ClockFrequency:  Integer,
  Samples:  Array |

// Read the telegram from file.
FSamples := new(FileIn) source(FileName) open;
ClockFrequency := FSamples readInteger;
NrOfTransitions := FSamples readInteger;
i := 0; Samples := new(Array) size(NrOfTransitions);
while (i := i + 1) <= NrOfTransitions do Samples put(i, FSamples readInteger) od;

// Transmit the telegram.
LCD writeln("Transmitting....");
i := NrOfTelegramsSent := 0;
while
    if i == NrOfTransitions then NrOfTelegramsSent := NrOfTelegramsSent + 1 fi;
    i := i modulo(NrOfTransitions) + 1;
    NrOfTelegramsSent < 5
do
    IR Out(i & 1);
    delay Samples get(i) / ClockFrequency
od.
```

and finally the durations between the consecutive transitions. By definition, the first transition will switch the transmitter on. In order to reduce the error sensitivity, the receiving device usually only accepts a message after having received several correct telegrams; `TransmitTelegram` therefore repeats the telegram five times (this is an arbitrary number).

Method `ReceiveTelegram` (also of process class `ProcessingUnit`) is shown in Table 7.6. It writes a message to the display (`LCD`) to give the user instructions. Then a data object of class `IRSignal` is constructed, which will store the samples (signal transitions) and implements a signal-analysis algorithm for recovering the telegram from the samples. After enabling input events, the method will wait for the transceiver's input signal to reach one (=on) before informing the user that it has detected the signal of the sending remote control. A loop will keep waiting for other transitions, adding the time of occurrence and signal state to `Samples`. A watchdog, formed by the delay statement will abort scanning after `SampleTime`. Then the input events are disabled, the user is informed that sampling has ended and the samples are analysed (and implicitly stored).

7.3 Generating External Events

As was mentioned in the overview (Section 7.1), the model can interact with the environment through primitive methods of data objects. In general, primitive methods encapsulate behaviour that cannot be captured by POOSL expressions but is provided in a native language. Here, these methods are used for defining and implementing an interface to the real world. To generate external events, a primitive method can call library functions, use device drivers or even access hardware directly.

As an example, Table 7.7 shows the implementation of the primitive method `Out` of `IR_Transceiver` in C. First the operand is checked, since the parameter to method

Table 7.6: Method `ReceiveTelegram` of process class `ProcessingUnit`.

```
ReceiveTelegram(FileName:  String, SampleTime:  Real)()
| FSamples:  FileOut, Samples:  IRSignal |

LCD writeln("Aim your original remote control (ORC) to the LIRC,")
    writeln("and press the key on the ORC you want the LIRC to learn.");

Samples := new(IRSignal) Initialise;
IR EnableInputEvents;

// Wait for the input signal.
[IR In == 1] LCD writeln("Signal detected, please keep the key pressed.");

// Start scanning transitions on the input of the IR transceiver.
abort
    while true do
        [IR InputHasChanged] Samples AddSample(currentTime, IR In)
    od
with
    delay SampleTime;

IR DisableInputEvents;
LCD writeln("You can now release the key.");

Samples CorrectForDutyCycle(1.0E-5) AnalyseSignal.
```

Table 7.7: Implementation of primitive method `Out` of `IR_Transceiver` in C.

```
static PDO *PDM_Out(PDO **LV)
{
    if (LV[1]->Class != PDC_Integer)
        DisplayError("Operand of IR_Transceiver.Out is not an Integer.");
    _outp(0x387, (uint8)LV[1]->i);
    return LV[0];
}   // end of PDM_Out().
```

`Out` should be of class `Integer`. Then the parameter's value (`LV[1]->i`) is written to the data register of the parallel port interface (assumed to be present at port address `0x378`). Finally a reference to the executing object itself is returned (`LV[0]`).

7.4 Synchronising with Real Time

The core of method `TransmitTelegram` is a loop that flips the output of the transmitter and waits for the next transition (lines 16–23 in Table 7.5). But how does the model time relate to the *real* time? In a normal (non-real time) simulation, when the next moment in model time has been computed, the simulation clock is updated and the simulation proceeds immediately. For real-time simulations this situation is impractical, because external processes in the real world depend on physical time, which cannot make such leaps instantaneously. Hence, to allow interaction with the environment, the model time should be synchronised with the real time.

A model introduces moments in model time by its delay statements. Such moments or *instants* will be denoted by mt_i where $i \in \mathbb{N}_0$. The periods between instants are called epochs. The corresponding instants in real time will be denoted by rt_i. The model time will be *synchronised* at instant mt_i with real time if $mt_i - mt_0 = (rt_i - rt_0 + \varepsilon) \cdot c$ for some $0 \leq \varepsilon \leq \varepsilon_{max}$ and $\varepsilon_{max} \in \mathbb{R}$. The scheduler, responsible for synchronising the model time with real time introduces the inevitable timing error ε because it can only read a clock or timer at discrete moments in real time (it is running on a synchronous processor). Notice that $\varepsilon \geq 0$ because the clock readout always lags behind. Speed factor c allows model time to progress faster than real time ($c > 1$), equally fast ($c = 1$) or slower than real time ($c < 1$).

The semantics prescribes that actions take no model time, but in the real world they certainly do consume physical time. Two approaches have been identified that cope with this discrepancy without violating POOSL's semantics.

The first approach is to have a discrete model time and synchronise at any possible instant. This requires all actions, scheduled at a particular instant mt_i, to be finished before $mt_i + 1$. This imposes very stringent timing constraints on the execution: the computationally most expensive actions must satisfy the same constraint as the cheapest actions. On the other hand, the grain of the simulation clock is always known: parallel composition cannot introduce worse timing constraints simply because there are no instants between mt_i and $mt_i + 1$.

In the second approach, the timing constraints can be relaxed by synchronising the model time only when it is actually needed, that is on the instants defined by the model. Model time is kept constant during mt_i and mt_{i+1} as is shown in Figure 7.8. If the unit time of the simulation clock is small compared to the minimal time between any two instants imposed by the model, then the timing constraints are much easier to satisfy (with respect to the first approach), while still offering a high time resolution. This approach does not require the time domain to be discrete. In contrast to the first approach, the minimal time between instants is not known beforehand, but the first approach seems to be too restrictive for prototyping; the second approach is preferred and has been implemented.

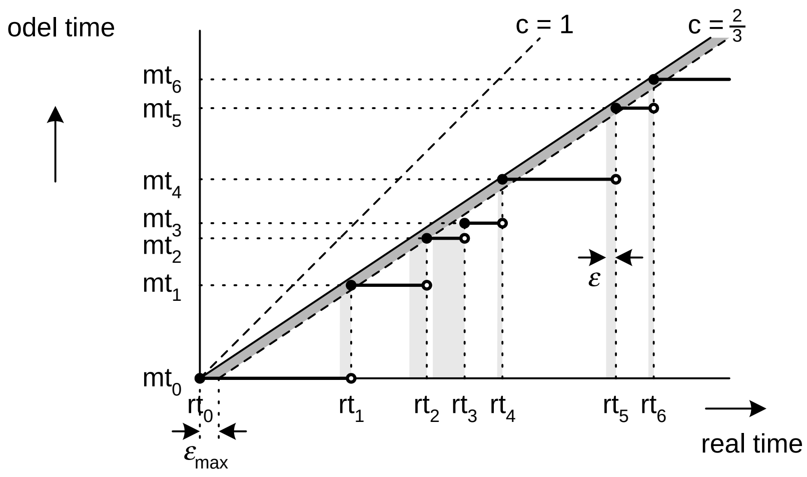


Figure 7.8: Synchronising model time with real time.

Both approaches are comparable to the important *synchrony hypothesis* [4, 3] of synchronous languages such as ESTEREL [2], LUSTRE [21], ARGOS, SIGNAL [15] and STATECHARTS [23] or zero-delay models used in hardware design. The synchrony hypothesis states that each reaction is instantaneous (here: actions take no model time) and only temporal statements take time (here: only delay statements consume model time).

A closely related question is when synchronisation has failed. The synchronisation formula uses ε to express the timing error, allowing a maximum timing error of ε_{max}. Synchronisation has failed if $\varepsilon > \varepsilon_{max}$. Ideally ε_{max} would be zero, but since this cannot be achieved in a practice, it is chosen as small as needed. The exact value is determined by both the computer on which the real-time simulator is running and the timing tolerance of the environment.

Real-time computing comes in two forms: *hard* real time and *soft* real time. Hard real-time computing does not tolerate any timing constraint being violated, as this might result in fatal incidents. Soft real-time computing (best-effort computing) is allowed to occasionally miss a temporal deadline as long as it catches up eventually. The real-time variant of the simulator acts like the latter type, and just counts the number of deadlines it has missed so far and proceeds.

7.5 Reacting to External Events

How and when can an external event influence the running simulation? To answer this question, the statement `[IR InputHasChanged] Samples AddSample(currentTime, IR In)` inside the scanning loop of `ReceiveTelegram` (Table 7.6, line 16) needs closer examination. Process instance variable `IR` encapsulates the infrared transceiver. Sending it method `InputHasChanged` will either return *false* when the transceiver's input has not changed or, more interestingly, return *true* when the input did change. The statement behind the guard that stores the time and input level in data object `Samples`, will therefore only be executed when the transceiver's input has changed.

Suppose that `ReceiveTelegram` is the only executing method, with the transceiver's input still at its initial level. The scheduler will test the guard and conclude that it blocks. Since no further action requests are available, it will let model time progress and wait until simulation time and real time are again in sync. Effectively, this means that after the guard has become blocking, no additional events will be detected. This is clearly an unsatisfactory situation. If the scheduler would have reevaluated the guarding expressions, it could have detected any guards becoming nonblocking and perform the accompanying statements. Before making plausible that reevaluating the guards is indeed a sensible implementation choice, we will compare this behaviour with interprocess communication.

The example in Table 7.9 shows two processes for which we will assume that they can communicate through their ports `IR`. Process `Q` is used in this example for simulating transitions on the infrared transceiver, and does so by sending —in the course of time— several messages `InputHasChanged` to process `P` and, along with it, the simulated input level. Process `P` is willing to wait for any transition during five units of

Table 7.9: Example of two synchronising processes.

Process P:

```
abort
    while true do
        IR?InputHasChanged(IR_In)
            {Samples AddSample(currentTime, IR_In)}
    od
with
    delay 5
```

Process Q:

```
delay 1.7;
IR!InputHasChanged(1);
delay 0.3;
IR!InputHasChanged(0);
delay 0.9;
IR!InputHasChanged(1)
```

time. (Notice the strong resemblance with lines 14–19 of method `ReceiveTelegram` in Table 7.6). In the example, `Q` defines intermediate points in time at which `P` can perform an action, namely at 1.7, 2.0 and 2.9. In case of the scanning loop of method `ReceiveTelegram`, the intermediate points in time are defined by an external process in much the same way as `Q` does for `P`. The main difference is that the very moment at which the guard will become nonblocking is now unknown to the scheduler.

Attempts to have the model generate the instants at which the guard should be checked will be futile[2] because the time at which an external event occurs is (in general) unpredictable. The scheduler, however, can pretend as if the external event has performed a delay request that expires *exactly* when the event occurs. At that instant, the scheduler will reexamine the guard and may find that it has become nonblocking and execute the accompanying action.

The view that external events introduce an additional moment in model time is quite natural as shown by means of the example in Table 7.9 and will be adopted by the real-time simulator. The working scheme is thus as follows. The data class that encapsules some event-generating device installs an *event handler* to notify the scheduler in case an event occurs. As long as the scheduler can perform an action transition, it can ignore the notifications of the event handlers, because guards will eventually be reevaluated before allowing model time to pass. If on the other hand, the scheduler is waiting for time synchronisation, it *will* react to a notification by adjusting the model time to reflect the current[3] instant at which the event is taking place, and retry performing an action transition.

After an event has been detected, it should be "consumed" to prevent it from being used over and over again. Line 16 in Table 7.6 illustrates the need for consuming events. The guard may have to be evaluated several times, so we cannot presume that the event is no longer needed after `IR InputHasChanged` has been evaluated. Instead, the `IR In` consumes the event (only once) after the scheduler has decided to execute the statement in line 16. So, the model implicitly notifies the device driver that the event has been consumed.

An event handler can either be passive —explicitly called by the scheduler to poll for an event— or active, like an interrupt. The real-time simulator currently uses only

[2]In parallel with the scanning loop, another loop could be used for creating all possible instants in time, effectively forcing the scheduler to reevaluate the guard time and again. Needless to say, this is such an inefficient method that it will not be considered realistic.

[3]The time that would have resulted from synchronising model time with the current physical time.

the first kind of event handlers, for simplicity reasons only. Since the scheduler will only poll the event handlers while it is waiting for time to synchronise, the overall speed of the simulator is not compromised.

The case study of the learning infrared remote control has demonstrated successfully that prototyping with POOSL is feasible after the suggested modifications have been made to the simulator (rotalumis-rt).

7.6 Extending the Semantics for Prototyping

This section will formalise the ideas described in Section 7.2.3. We define the set of external variables $EVar$ with typical elements χ, and the set of external events $ExtEvent = \{\lightning\}$. The set of external variables states $Z = EVar \hookrightarrow DObj$ has typical elements ζ that map external variables to their values.

Definition 44 (event transitions)
We let $\cdot \xrightarrow[ExtEvent]{\cdot} \cdot \subseteq Z \times ExtEvent \times Z$ *denote external event transitions.*

Definition 45 (time transitions)
We let $\cdot \xrightarrow[Time^+]{\cdot} \cdot \subseteq Z \times Time^+ \times Z$ *denote time transitions.*

The external processes (processes in the real world) are modelled by the timed labelled transition system $(Z, ExtEvent, \xrightarrow[ExtEvent]{}, Time, \xrightarrow[Time^+]{})$. We assume that this transition system is given, is deterministic and that after each transition the state is changed (to guarantee that the system is time-closed). We let $\zeta \xrightarrow{\lightning} \zeta'$ (Definition 44) denote the occurrence of an external event while being in state ζ, and yielding the terminal state ζ'. The state remembers the current time whose value $\zeta.currentTime$ is assumed to be zero when a simulation is started.

In the simulator, the transition system that models the real world is generated on-the-fly by device drivers that monitor external processes. When an external process generates an event, the device driver will record the event in its variables (elements of $EVar$) and notify the scheduler that an event transition has happened.

For allowing POOSL models to interact with external processes, we define the following transition system to model their composition:

$$(Conf^e, Act, \xrightarrow[Act]{}, Time, \xrightarrow[Time^+]{}, ExtEvent, \xrightarrow[ExtEvent]{}).$$

The set of extended configurations $Conf^e = Z \times Conf$ is defined by:

$$c^e \quad = \quad \zeta \parallel\!\mid c$$

where the interleaving operator $\parallel\!\mid$ denotes the interleaving of internal transitions with external transitions. Its meaning is defined by semantic rules 63 (PAR$_8$), 64 (PAR$_9$), 65 (PAR$_{10}$) in appendix A.4. Rule 63 (PAR$_8$) gives priority to action transitions the POOSL model can perform. The yielded extended probability function is defined as follows.

Definition 46 (extended substochastic probability function)
The set of extended substochastic probability functions is denoted by $\mathcal{P}(Conf^e) = \left\{ \boldsymbol{\pi}^e \in Conf^e \rightarrow \mathcal{R} \;\middle|\; \sum_{c^e \in Conf^e} \boldsymbol{\pi}^e.c^e \leq 1 \right\}$.

If the POOSL model is waiting for the model time to synchronise with the real time (rule 64, PAR_9), the composed system can react to an external event. Without giving a detailed definition of function $IncorporateExtData \in Conf \times Z \rightarrow Conf$, we assume that this function incorporates the external data (stored in ζ', representing the device drivers' external variables) into the model by only modifying the values of global or local variables referring to primitive objects whose data reflects part of the outer world. Rule PAR_9 allows the transition system to update its image of the external processes only if it cannot perform any actions.

The composed system can let time advance only if there are no events within t units of model time (rule 65, PAR_{10}). To this end the urgency predicate $Urgent(\zeta, t)$ is used. An external process ζ is urgent, written as $Urgent(\zeta)$, if it can perform an event transition (denoted by $\zeta \xrightarrow{\not{t}}$). An external process ζ is urgent within time t, denoted by $Urgent(\zeta, t)$, if there exists a $t' \leq t$ and external process ζ' such that $\zeta \xrightarrow{t'} \zeta'$ and ζ' is urgent.

Notice that *currentTime* defined in ζ equals the model time of the POOSL model. In the implementation, the moments in model time induced by the combined transition system are called instants, denoted by mt_i where $i, j \in \mathbb{N}_0$ and such that $mt_i \leq mt_j$ iff $i \leq j$ (see Figure 7.8). The corresponding instants in real time will be denoted by rt_i. The model time will be assumed to be *synchronised* with real time at any instant. An instant mt_i is synchronised if $mt_i - mt_0 = (rt_i - rt_0 + \varepsilon) \cdot c$ for some $0 \leq \varepsilon \leq \varepsilon_{max}$. Constant ε takes care of (inevitable) timing errors; the maximum allowable error is $\varepsilon_{max} \in \mathbb{R}$. The constant c can be interpreted as speed factor, allowing model time to progress faster than real time ($c > 1$), equally fast ($c = 1$) or slower than real time ($c < 1$).

7.7 Summary

The simulator has been extended for prototyping — rotalumis-rt. In a real-time simulation, part of the system is formed by the prototype (the executable model), while the rest is formed by elements in the real world (the external processes). Normally the simulator can make leaps in model time, but now external processes obey physical time. To keep the prototype in pace with its environment, the model time must be synchronised with physical time. This imposes time constraints on the actions performed by the prototype: the semantics prescribes actions to be timeless, but the prototype consumes physical time to execute them. Two approaches have been identified for dealing with this discrepancy without violating POOSL's semantics. The first approach introduces a discrete model time domain and requires actions to be executed before the next epoch starts. The second approach only synchronises time on instants defined by the model or the environment. Between two synchronisation points, model time is considered constant. Both approaches are similar to the important synchrony hypothesis in synchronous languages such as ESTEREL, LUSTRE, ARGOS, SIGNAL and STATECHARTS or zero-delay models used in hardware design.

Data classes function as abstractions of part of the external environment (the real world). Their primitive methods are viewed of as device drivers to communicate with hardware (actual devices) or software (library functions) and offer a clean interface towards the model. Events from external processes introduce additional instants if the scheduler is idling for time synchronisation. This implementation has been chosen because it is similar to the case where one process is awaiting a message from another internal process, but does not know when the message will come — the latter process introduces an instant in time that did not exist for the former process before.

A successful case study of a learning infrared remote control has demonstrated the applicability of rotalumis-rt to prototyping by letting a POOSL model emulate software that interacts with an actual hardware prototype.

Chapter 8

Conclusions and Future Work

Conclusions

This thesis develops a new formal semantics for POOSL (Parallel Object-Oriented Specification Language) [41] and describes a constructive approach to building an efficient execution engine for that language based on its formal semantics.

The new semantics is developed for several reasons. Modelling practice has led to the desire for incorporating a few additional modelling concepts into POOSL, such as the concept of time [17]. The semantics now supports the new theory for performance analysis based on Markov chains developed in [54, 53, 52, 55] by equipping expressions and statements with probabilistic features. The presented two-layered semantics consists of a denotational semantics for the data layer of POOSL and a structural operational semantics for the process and architecture layer. It unifies the concepts presented in [41] and [17] adding:

- probabilistic features;
- real-time;
- inheritance;
- concurrency within processes;
- immediate (indivisible) expressions;
- dynamic communication ports.

The semantics is an initial definition and verifying its desired properties is future research. This thesis has elaborated on a constructive approach to implementing platforms that can execute POOSL primitives efficiently and in such a way that their behaviour described by a formal semantics is respected.

To obtain *executable* models, the proposed constructive approach has successfully been applied to POOSL, yielding the execution engine rotalumis[1]. This engine contains

[1]Rotalumis can be downloaded from `www.ics.ele.tue.nl/~lvbokhov/poosl/rotalumis`.

a compiler that transforms human readable POOSL specifications into bytecode, a constructor that creates the architecture described in a model, and two platforms for the execution of dynamic behaviour. The platform that executes expressions (the data layer of POOSL) is a virtual machine with an instruction set that contains `push`, `pop`, `call`, `jump` and `test` instructions. To reclaim storage occupied by unreachable data objects, the virtual machine is equipped with a hybrid garbage collector that relies on reference counting and Baker's Treadmill. The state and dynamic behaviour of processes is captured by execution trees [18]. This dissertation has presented an efficient implementation of execution trees.

Rotalumis has been optimised for handling large industrial-sized models. The optimisations are described in this thesis since they may also be applicable to execution engines for other languages. Two industrial case studies have successfully demonstrated that rotalumis can cope with large models: an ATM-packet switch (in conjunction with IBM Research at Zürich) and a packet routing switch for the Internet (in association with Alcatel/Bell at Antwerp).

The execution engine always relinquishes control after having performed an execution step, and can therefore be used as a slave component in a larger simulation framework.

Additionally, rotalumis has been extended to function as a prototype implementation of (part of) the designed product. This requires the execution engine to cooperate with actual processes (either in hardware or software) and synchronise model time with the real time. A case study of a learning infrared remote control has shown the viability of this extension.

Future Work

To increase the ease-of-use of rotalumis, the following items will have to be dealt with:

- coupling to the graphical user interface of SHESim [16];
- debugging support (breakpoints, inspection of the model's state);
- additional compiler optimisations on expressions (although the dynamic method call severely reduces the optimisation possibilities).

A framework of observers, which monitor performance properties of a POOSL model, is desired for facilitating performance analyses. Part of such a framework is a language for stating performance measurements in the form of reward expressions.

Because of the floating-point representation of model time (in case of a dense time domain), small rounding errors can accumulate that may compromise the correct execution of the model — this deserves attention.

Distributed simulation offers the perspective of being able to simulate models that are too large to fit within a single machine. Such distribution is nontrivial, requiring semantics-preserving synchronisation protocols for time and communication. Because failure of network connections or machines is bound to happen, marshalling (part of) the simulation state is needed, to enable saving and restoring that state.

Some desired properties of the formal semantics have not yet been verified. Examples of such properties are time-determinism and time-additivity. Mathematical proofing techniques could also be applied to construct a set of reduction rules that enable further optimisations either at compile-time or at run-time.

Finally, the industry's interest in synthesis to software and hardware for implementation requires, among many other issues, resolving nondeterminism in models.

Appendix A

The Semantics of POOSL

A.1 The Data Layer of POOSL

The semantic rules in this section establish the semantic function (see Definition 30) $[\![\cdot]\!]_{\cdot} \in \mathit{Exp} \to \mathcal{S}^n \to \mathcal{S}$ for each construct in Exp listed in Section 3.2.1. The rules implicitly assume $f \in \mathcal{S}^n$, $s \in \mathit{State}$ and $s' \in \mathit{State}'$.

1. *Primitive data objects*

$$[\![\underline{\gamma}]\!]_f.s.s' = \begin{cases} 1 & \text{if } s' = (s, \gamma) \\ 0 & \text{otherwise.} \end{cases}$$

2. *Instance variables*

$$[\![x]\!]_f.s.s' = \begin{cases} 1 & \text{if } s' = (s, \sigma.\delta.x) \text{ where } s = (\sigma, \lambda, \tau),\ \lambda = (\delta, \psi), \\ & \sigma.\delta \neq \underline{\text{undef}} \text{ and } \sigma.\delta.x \neq \underline{\text{undef}} \\ 0 & \text{otherwise.} \end{cases}$$

3. *Local variables*

$$[\![u]\!]_f.s.s' = \begin{cases} 1 & \text{if } s' = (s, \psi.u) \text{ where } s = (\sigma, \lambda, \tau),\ \lambda = (\delta, \psi) \\ & \text{and } \psi.u \neq \underline{\text{undef}} \\ 0 & \text{otherwise.} \end{cases}$$

4. *Self*

$$[\![\texttt{self}]\!]_f.s.s' = \begin{cases} 1 & \text{if } s' = (s, \delta) \text{ where } s = (\sigma, \lambda, \tau),\ \lambda = (\delta, \psi) \\ & \text{and } \delta \neq \mathit{proc} \\ 0 & \text{otherwise.} \end{cases}$$

5. *Current time*

$$[\![\texttt{currentTime}]\!]_f.s.s' = \begin{cases} 1 & \text{if } s' = (s, \sigma.\mathit{proc}.\mathit{currentTime}) \text{ where} \\ & s = (\sigma, (\mathit{proc}, \psi), \tau) \\ 0 & \text{otherwise.} \end{cases}$$

6. *Object creation*

$$[\![\texttt{new}(C)]\!]_f.s.s' = \begin{cases} 1 & \text{if } \mathit{CDList} \equiv \mathit{CD}_1 \cdots \mathit{CD}_i \cdots \mathit{CD}_n \\ & \text{and } \mathit{CD}_i \equiv \begin{array}{ll} \;\;\textbf{data class} & C \\ [\,\textbf{extends} & C_{super}\,] \\ \;\;\textbf{instance variables} & x_1 \cdots x_n \\ \;\;\textbf{instance methods} & \mathit{MD}_1 \cdots \mathit{MD}_k \end{array} \\ & \text{and } s = (\sigma, \lambda, \tau),\ n = \mathit{MaxId}(\sigma) + 1,\ \mathrm{Dom}(\phi) = \mathcal{V}(C) \\ & \text{such that } \phi(x) = \mathit{nil} \text{ for all } x \in \mathrm{Dom}(\phi),\ \sigma' = \sigma\{\phi/\widehat{n}\}, \\ & \tau' = \tau\{C/\widehat{n}\} \text{ and } s' = \big((\sigma', \lambda, \tau'), \widehat{n}\big) \\ 0 & \text{otherwise.} \end{cases}$$

7. *Assignment to instance variables*

$$[\![x\texttt{:=}E]\!]_f.s.s' = \sum_{t':\mathrm{P1}} [\![E]\!]_f.s.t'$$

$$\mathrm{P1}: \quad t' = \big(s'', \beta\big);$$

if $s'' = (\sigma, \lambda, \tau)$, $\lambda = (\delta, \psi)$ and $s' = \big((\sigma', \lambda, \tau), \beta\big)$ such that $\sigma' = \sigma\big\{\sigma.\delta\{\beta/x\}/\delta\big\}$, $\sigma.\delta \neq \underline{\text{undef}}$ and $\sigma.\delta.x \neq \underline{\text{undef}}$.

8. *Assignment to local variables*

$$[\![u\texttt{:=}E]\!]_f.s.s' = \sum_{t':\mathrm{P1}} [\![E]\!]_f.s.t'$$

$$\mathrm{P1}: \quad t' = (s'', \beta);$$

if $s'' = (\sigma, \lambda, \tau)$, $\lambda = (\delta, \psi)$ and $s' = \big((\sigma, \lambda', \tau), \beta\big)$ where $\lambda' = \big(\delta, \psi\{\beta/u\}\big)$ and $\psi.u \neq \underline{\text{undef}}$.

9. *Sequential composition*

$$[\![E_1\texttt{;}E_2]\!]_f.s.s' = \sum_{t':\mathrm{P1}} [\![E_1]\!]_f.s.t' \times [\![E_2]\!]_f.t.s'$$

$$\mathrm{P1}: \quad t' = (t, \beta).$$

10. *If*

$$[\![\texttt{if}\ E_c\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{fi}]\!]_f.s.s' = \sum_{t'_T:\mathrm{P1}} [\![E_c]\!]_f.s.t'_T \times [\![E_1]\!]_f.t_T.s' + \sum_{t'_F:\mathrm{P2}} [\![E_c]\!]_f.s.t'_F \times [\![E_2]\!]_f.t_F.s'$$

$$\mathrm{P1}: \quad t'_T = (t_T, \mathit{true});$$
$$\mathrm{P2}: \quad t'_F = (t_F, \mathit{false}).$$

11. *While*

$$[\![\texttt{while}\ E_c\ \texttt{do}\ E\ \texttt{od}]\!]_f = \mathrm{FIX}\,\mathcal{X}_f$$

where $\mathcal{X}_f(g).s.s' = \sum_{t'_T:\mathrm{P1}} \sum_{t':\mathrm{P2}} [\![E_c]\!]_f.s.t'_T \times [\![E]\!]_f.t_T.t' \times g.t.s' + \sum_{t'_F:\mathrm{P3}} [\![E_c]\!]_f.s.t'_F$

$$\mathrm{P1}: \quad t'_T = (t_T, \mathit{true});$$
$$\mathrm{P2}: \quad t' = (t, \beta);$$
$$\mathrm{P3}: \quad t'_F = (t_F, \mathit{false}) \text{ and } s' = (t_F, \mathit{nil}).$$

12. *Dynamic method call*

$$[\![E\ m(E_1,\dots,E_n)]\!]_f.s.s' = \sum_{t_0':\mathrm{P0}}\cdots\sum_{t_n':\mathrm{P}n}\sum_{u:\mathrm{Q}}[\![E]\!]_f.s.t_0' \times [\![E_1]\!]_f.t_0.t_1' \times\cdots\times [\![E_n]\!]_f.t_{n-1}.t_n' \times \mu.u.w'$$

$$\begin{array}{ll} \mathrm{P0}: & t_0' = (t_0,\beta_0) \text{ and } t_0 = \big(\sigma_0,(\delta,\psi_0),\tau_0\big);\\ \mathrm{P}j: & t_j' = (t_j,\beta_j) \text{ and } t_j = \big(\sigma_j,(\delta,\psi_j),\tau_j\big) \text{ for } j\in\{1,\dots,n\};\\ \mathrm{Q}: & u = \big(\sigma_n,(\beta_0,\psi''),\tau_n\big),\ \psi'' = \psi'''\{\beta_1/u_1\}\cdots\{\beta_n/u_n\}\{nil/z_1\}\cdots\{nil/z_m\}\\ & \quad\text{and } w' = \big(\big(\sigma',(\beta_0,\psi'),\tau'\big),\beta'\big), \text{ where } \mathrm{Dom}(\psi''') = \varnothing; \end{array}$$

if $s = \big(\sigma,(\delta,\psi),\tau\big)$ and $s' = \big(\big(\sigma',(\delta,\psi_n),\tau'\big),\beta'\big)$ and where

$$\mu = \begin{cases} f_i & \text{where } i = \mathcal{I}.\tau_0(\beta_0).m \text{ if } \mathcal{M}.\tau_0(\beta_0).m \equiv \begin{array}{l} m(u_1,\dots,u_n)\\ |z_1\cdots z_m|\\ E' \end{array}\\ \mathcal{D}.\tau_0(\beta_0).m & \text{if } \mathcal{M}.\tau_0(\beta_0).m \equiv \begin{array}{l} m(u_1,\dots,u_n)\\ |z_1\cdots z_m|\\ \texttt{primitive} \end{array}\\ \perp_S & \text{otherwise.} \end{cases}$$

13. *Static method call*

$$[\![E\ m_C(E_1,\dots,E_n)]\!]_f.s.s' = \sum_{t_0':\mathrm{P0}}\cdots\sum_{t_n':\mathrm{P}n}\sum_{u:\mathrm{Q}}[\![E]\!]_f.s.t_0' \times [\![E_1]\!]_f.t_0.t_1' \times\cdots\times [\![E_n]\!]_f.t_{n-1}.t_n' \times \mu.u.w'$$

$$\begin{array}{ll} \mathrm{P0}: & t_0' = (t_0,\beta_0) \text{ and } t_0 = \big(\sigma_0,(\delta,\psi_0),\tau_0\big);\\ \mathrm{P}j: & t_j' = (t_j,\beta_j) \text{ and } t_j = \big(\sigma_j,(\delta,\psi_j),\tau_j\big) \text{ for } j\in\{1,\dots,n\};\\ \mathrm{Q}: & u = \big(\sigma_n,(\beta_0,\psi''),\tau_n\big),\ \psi'' = \psi'''\{\beta_1/u_1\}\cdots\{\beta_n/u_n\}\{nil/z_1\}\cdots\{nil/z_m\}\\ & \quad\text{and } w' = \big(\big(\sigma',(\beta_0,\psi'),\tau'\big),\beta'\big), \text{ where } \mathrm{Dom}(\psi''') = \varnothing; \end{array}$$

if $s = \big(\sigma,(\delta,\psi),\tau\big)$ and $s' = \big(\big(\sigma',(\delta,\psi_n),\tau'\big),\beta'\big)$ and where

$$\mu = \begin{cases} f_i & \text{where } i = \mathcal{I}.C.m \text{ and } \mathcal{M}_s.C.m \equiv \begin{array}{l} m(u_1,\dots,u_n)\\ |z_1\cdots z_m|\\ E' \end{array}\\ \mathcal{D}.C.m & \text{if } \mathcal{M}_s.C.m \equiv \begin{array}{l} m(u_1,\dots,u_n)\\ |z_1\cdots z_m|\\ \texttt{primitive} \end{array}\\ \perp_S & \text{otherwise.} \end{cases}$$

A.2 The Process Layer of POOSL

The semantic rules in this section establish the timed probabilistic labelled transition system $(\mathit{Conf}, \mathit{Act}, \xrightarrow[\mathit{Act}]{}, \mathit{Time}, \xrightarrow[\mathit{Time}^+]{})$ defined in Section 4.2.3. The transition system is completed by the rules in Appendix A.3.

14. *Expression*

$$\frac{-}{\left([E]^{\psi}_{C}, \sigma, \tau\right) \xrightarrow{\tau} \boldsymbol{\pi}}\ \textsc{exp}$$

where

$$\boldsymbol{\pi}.c = \begin{cases} [\![E]\!].s.s' \\ \qquad \text{for } c = \left([\surd]^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{if } s = \left(\sigma, (\mathit{proc}, \psi), \tau\right) \text{ and } s' = \left((\sigma', (\mathit{proc}, \psi'), \tau'), \beta\right) \\ 0 \qquad \text{otherwise.} \end{cases}$$

15. *If*

$$\frac{-}{\left([\texttt{if } E \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}]^{\psi}_{C}, \sigma, \tau\right) \xrightarrow{\tau} \boldsymbol{\pi}}\ \textsc{if}$$

where

$$\boldsymbol{\pi}.c = \begin{cases} [\![E]\!].s.s' \\ \qquad \text{for } c = \left([S_1]^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{if } s = \left(\sigma, (\mathit{proc}, \psi), \tau\right) \text{ and } s' = \left((\sigma', (\mathit{proc}, \psi'), \tau'), \mathit{true}\right) \\ [\![E]\!].s.s' \\ \qquad \text{for } c = \left([S_2]^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{if } s = \left(\sigma, (\mathit{proc}, \psi), \tau\right) \text{ and } s' = \left((\sigma', (\mathit{proc}, \psi'), \tau'), \mathit{false}\right) \\ 0 \qquad \text{otherwise.} \end{cases}$$

16. *While*

$$\frac{-}{\left([\texttt{while } E \texttt{ do } S \texttt{ od}]^{\psi}_{C}, \sigma, \tau\right) \xrightarrow{\tau} \boldsymbol{\pi}}\ \textsc{while}$$

where

$$\boldsymbol{\pi}.c = \begin{cases} [\![E]\!].s.s' \\ \qquad \text{for } c = \left([S\texttt{;while } E \texttt{ do } S \texttt{ od}]^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{if } s = \left(\sigma, (\mathit{proc}, \psi), \tau\right) \text{ and } s' = \left((\sigma', (\mathit{proc}, \psi'), \tau'), \mathit{true}\right) \\ [\![E]\!].s.s' \\ \qquad \text{for } c = \left([\texttt{nil}]^{\psi'}_{C}, \sigma', \tau'\right) \\ \qquad \text{if } s = \left(\sigma, (\mathit{proc}, \psi), \tau\right) \text{ and } s' = \left((\sigma', (\mathit{proc}, \psi'), \tau'), \mathit{false}\right) \\ 0 \qquad \text{otherwise.} \end{cases}$$

17. *Guarded command 1*

$$\frac{\big([S]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}'}{\big([\texttt{[}E\texttt{]}S]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}}\ \text{GRD}_1$$

where

$$\boldsymbol{\pi}.c=\begin{cases}\boldsymbol{\pi}'.\big([S']^{\psi'}_{C},\sigma',\tau'\big)\\ \qquad\text{for } c=\big([S']^{\psi'}_{C},\sigma',\tau'\big)\\ \qquad\text{if } [\![E]\!].s.(s,\mathit{true})=1,\ \mathit{Timeless}(E,s)\text{ and } a\neq f,\\ \qquad\text{where } s=\big(\sigma,(\mathit{proc},\psi),\tau\big)\\ \boldsymbol{\pi}'.\big([S']^{\psi'}_{C},\sigma',\tau'\big)\\ \qquad\text{for } c=\big([\surd(\texttt{[}E\texttt{]}S')]^{\psi'}_{C},\sigma',\tau'\big)\\ \qquad\text{if } [\![E]\!].s.(s,\mathit{true})=1,\ \mathit{Timeless}(E,s)\text{ and } a=f,\\ \qquad\text{where } s=\big(\sigma,(\mathit{proc},\psi),\tau\big)\\ 0\qquad\text{otherwise.}\end{cases}$$

18. *Parallel composition 1*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}'}{\big([\textbf{par}\ S_1\ \textbf{and}\ S_2\ \textbf{rap}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}}\ \text{PAR}_1$$

where

$$\boldsymbol{\pi}.c=\begin{cases}\boldsymbol{\pi}'.\big([S_1']^{\psi'}_{C},\sigma',\tau'\big)\\ \qquad\text{for } c=\big([\surd(\textbf{par}\ S_1'\ \textbf{and}\ S_2\ \textbf{rap})]^{\psi'}_{C},\sigma',\tau'\big)\\ 0\qquad\text{otherwise.}\end{cases}$$

19. *Parallel composition 2*

$$\frac{\big([S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}'}{\big([\textbf{par}\ S_1\ \textbf{and}\ S_2\ \textbf{rap}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}}\ \text{PAR}_2$$

where

$$\boldsymbol{\pi}.c=\begin{cases}\boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C},\sigma',\tau'\big)\\ \qquad\text{for } c=\big([\surd(\textbf{par}\ S_1\ \textbf{and}\ S_2'\ \textbf{rap})]^{\psi'}_{C},\sigma',\tau'\big)\\ 0\qquad\text{otherwise.}\end{cases}$$

20. *Sequential composition 1*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}'}{\big([S_1\,\texttt{;}\,S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{a}\boldsymbol{\pi}}\ \text{SEQ}_1$$

where

$$\boldsymbol{\pi}.c=\begin{cases}\boldsymbol{\pi}'.\big([S_1']^{\psi'}_{C},\sigma',\tau'\big)\\ \qquad\text{for } c=\big([\surd(S_1'\,\texttt{;}\,S_2)]^{\psi'}_{C},\sigma',\tau'\big)\\ 0\qquad\text{otherwise.}\end{cases}$$

21. *Select 1*

$$\frac{\big([S_1]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\texttt{sel}\; S_1\; \texttt{or}\; S_2\; \texttt{les}]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}} \;\textsc{sel}_1$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S_1']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([S_1']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a \neq f \\ \boldsymbol{\pi}'.\big([S_1']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(\texttt{sel}\; S_1'\; \texttt{or}\; S_2\; \texttt{les})]^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a = f \\ 0 \qquad \text{otherwise.} \end{cases}$$

22. *Select 2*

$$\frac{\big([S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\texttt{sel}\; S_1\; \texttt{or}\; S_2\; \texttt{les}]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}} \;\textsc{sel}_2$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a \neq f \\ \boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(\texttt{sel}\; S_1\; \texttt{or}\; S_2'\; \texttt{les})]^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a = f \\ 0 \qquad \text{otherwise.} \end{cases}$$

23. *Abort 1*

$$\frac{\big([S_1]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\texttt{abort}\; S_1\; \texttt{with}\; S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}} \;\textsc{abort}_1$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S_1']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(\texttt{abort}\; S_1'\; \texttt{with}\; S_2)]^{\psi'}_{C}, \sigma', \tau'\big) \\ 0 \qquad \text{otherwise.} \end{cases}$$

24. *Abort 2*

$$\frac{\big([S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\texttt{abort}\ S_1\ \texttt{with}\ S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}}\ \textsc{abort}_2$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a \neq f \\ \boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(\texttt{abort}\ S_1\ \texttt{with}\ S_2')]^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a = f \\ 0 \qquad \text{otherwise.} \end{cases}$$

25. *Interrupt 1*

$$\frac{\big([\texttt{interrupt}\ S_1\ \texttt{with}\ S_2, S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}}{\big([\texttt{interrupt}\ S_1\ \texttt{with}\ S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}}\ \textsc{intr}_1$$

26. *Interrupt 2*

$$\frac{\big([S_1]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\texttt{interrupt}\ S_1\ \texttt{with}\ S_2, S_3]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}}\ \textsc{intr}_2$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S_1']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(\texttt{interrupt}\ S_1'\ \texttt{with}\ S_2, S_3)]^{\psi'}_{C}, \sigma', \tau'\big) \\ 0 \qquad \text{otherwise.} \end{cases}$$

27. *Interrupt 3*

$$\frac{\big([S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\texttt{interrupt}\ S_1\ \texttt{with}\ S_2, S_3]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}}\ \textsc{intr}_3$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(S_1\ \texttt{interrupted by}\ S_2', S_3)]^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a \neq f \\ \boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(\texttt{interrupt}\ S_1\ \texttt{with}\ S_2', S_3)]^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } a = f \\ 0 \qquad \text{otherwise.} \end{cases}$$

28. *Interrupt 4*

$$\frac{\big([S_2]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([S_1 \; \texttt{interrupted by} \; S_2, S_3]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}} \; \text{INTR}_4$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S_2']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\surd(S_1 \; \texttt{interrupted by} \; S_2', S_3)]^{\psi'}_{C}, \sigma', \tau'\big) \\ 0 \qquad \text{otherwise.} \end{cases}$$

29. *Method call 1*

$$\frac{-}{\big([m_{C'}(E_1, \ldots, E_n)(w_1', \ldots, w_k')]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{f} \boldsymbol{\pi}} \; \text{MC}_1$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \sum\limits_{s_1:\text{P}1} \cdots \sum\limits_{s_n:\text{P}n} [\![E_1]\!].s.s_1' \times [\![E_2]\!].s_1.s_2' \times \cdots \times [\![E_n]\!].s_{n-1}.s_n' \\ \qquad \text{where } s = (\sigma, (\mathit{proc}, \psi), \tau) \\ \qquad\qquad \text{P}i \;:\; s_i = (\sigma_i, (\mathit{proc}, \psi_i), \tau_i) \text{ and } s_i' = \big(s_i, \beta_i\big) \text{ for } i \in \{1, \ldots, n\} \\ \qquad \text{for } c = \big([\, m_{C'}(\,)(w_1', \ldots, w_k')[S^b]^{\psi'}]^{\psi_n}_{C}, \sigma_n, \tau_n\big) \text{ if } k \neq 0 \\ \qquad \text{or } c = \big([\, [S^b]^{\psi'}]^{\psi_n}_{C}, \sigma_n, \tau_n\big) \text{ if } k = 0, \\ \qquad \text{if } \; \mathcal{M}.C'.m \quad \equiv \quad m(u_1, \ldots, u_n)(w_1, \ldots, w_k) \\ \qquad\qquad\qquad\qquad\quad |z_1 \cdots z_m| \\ \qquad\qquad\qquad\qquad\quad S^b \\ \qquad \text{and } \mathrm{Dom}(\psi') = \{u_1, \ldots, u_n, w_1, \ldots, w_k, z_1, \ldots, z_m\}, \; \psi'.v = \mathit{nil} \text{ for} \\ \qquad v \in \{w_1, \ldots, w_k, z_1, \ldots, z_m\} \text{ and } \psi'.u_i = \beta_i \text{ for } i \in \{1, \ldots, n\} \\ 0 \qquad \text{otherwise.} \end{cases}$$

30. *Method call 2*

$$\frac{\big([S]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\, m_{C'}(\,)(w_1', \ldots, w_k')[S]^{\psi}]^{\psi''}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}} \; \text{MC}_2$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S']^{\psi'}_{C}, \sigma', \tau'\big) \\ \qquad \text{for } c = \big([\, m_{C'}(\,)(w_1', \ldots, w_k')[S']^{\psi'}]^{\psi''}_{C}, \sigma', \tau'\big) \\ 0 \qquad \text{otherwise.} \end{cases}$$

31. *Method call 3*

$$\frac{\big([S]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}'}{\big([\,[S]^{\psi}\,]^{\psi''}_{C}, \sigma, \tau\big) \xrightarrow{a} \boldsymbol{\pi}} \ \text{MC}_3$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.\big([S']^{\psi'}_{C}, \sigma', \tau'\big) & \\ \qquad \text{for } c = \big([\,\surd(\mathbb{V}([S']^{\psi'}))\,]^{\psi''}_{C}, \sigma', \tau'\big) & \\ 0 \quad \text{otherwise.} & \end{cases}$$

32. *Method call 4*

$$\frac{-}{\big([\,m_{C'}()(w'_1, \ldots, w'_k)[\surd]^{\psi''}\,]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{f} \boldsymbol{\pi}} \ \text{MC}_4$$

where

$$\boldsymbol{\pi}.c = \begin{cases} 1 \quad \text{for } c = \big([\surd]^{\psi_k}_{C}, \sigma_k, \tau\big) \\ \quad \text{if } \mathcal{M}.C'.m \ \equiv \ m(u_1, \ldots, u_n)(w_1, \ldots, w_k) \\ \qquad\qquad\qquad |z_1 \cdots z_m| \\ \qquad\qquad\qquad S^b \\ \quad \text{where } \sigma_i = \begin{cases} \sigma_{i-1}\big\{\sigma_{i-1}.proc\{\psi''(w_i)/w'_i\}/proc\big\} & \text{if } w'_i \in \text{Dom}(\sigma) \\ & \text{and } w'_i \notin \text{Dom}(\psi) \\ \sigma_{i-1} & \text{otherwise} \end{cases} \\ \quad \text{and} \quad \psi_i = \begin{cases} \psi_{i-1}\{\psi''(w_i)/w'_i\} & \text{if } w'_i \in \text{Dom}(\psi) \\ \psi_{i-1} & \text{otherwise} \end{cases} \\ \quad \text{for } i \in \{1, \ldots, k\} \text{ with } \sigma_0 = \sigma,\ \psi_0 = \psi \\ 0 \quad \text{otherwise.} \end{cases}$$

33. *Message send 1*

$$\frac{-}{\big([E_p\,!\,m(E_1, \ldots, E_n)\{E\}]^{\psi}_{C}, \sigma, \tau\big) \xrightarrow{f} \boldsymbol{\pi}} \ \text{COMM}_1$$

where

$$\boldsymbol{\pi}.c = \begin{cases} [\![E_p]\!].s.s' \\ \quad \text{for } c = \big([\underline{\widetilde{p}}\,!\,m(E_1, \ldots, E_n)\{E\}]^{\psi'}_{C}, \sigma', \tau'\big) \\ \quad \text{if } s = \big(\sigma, (proc, \psi), \tau\big) \text{ and } s' = \big((\sigma', (proc, \psi'), \tau'), p\big) \\ 0 \quad \text{otherwise.} \end{cases}$$

34. *Message receive 1*

$$\frac{-}{\big([E_p\,?m(v_1,\ldots,v_n|E_{rc})\{E\}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{f}\boldsymbol{\pi}}\ \text{COMM}_2$$

where

$$\boldsymbol{\pi}.c=\begin{cases}[\![E_p]\!].s.s' & \\ \quad\text{for } c=\big([\underset{\sim}{p}?m(v_1,\ldots,v_n|E_{rc})\{E\}]^{\psi'}_{C},\sigma',\tau'\big) & \\ \quad\text{if } s=\big(\sigma,(proc,\psi),\tau\big)\text{ and } s'=\big((\sigma',(proc,\psi'),\tau'),p\big) & \\ 0\quad\text{otherwise.}\end{cases}$$

35. *Message send 2*

$$\frac{-}{\big([\underset{\sim}{p}\,!\,m(E_1,\ldots,E_n)\{E\}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{p\,!\,m[data]}\boldsymbol{\pi}}\ \text{COMM}_3$$

where

$$\boldsymbol{\pi}.c=\begin{cases}[\![E]\!].s.s' & \\ \quad\text{for } c=\big([\surd]^{\psi'}_{C},\sigma',\tau'\big)\text{ if } [\![E_1]\!].s.(s,\beta_1)\times\cdots\times[\![E_n]\!].s.(s,\beta_n)=1, & \\ \quad Timeless(E_i,s)\text{ for } i\in\{1,\ldots,n\},\ \underline{p}\,!\,m(\underline{n})\text{ in } MSS(C), & \\ \quad s=\big(\sigma,(proc,\psi),\tau\big),\ s'=\big((\sigma',(proc,\psi'),\tau'),\beta\big)\text{ and} & \\ \quad data=\big(DC(\beta_1,\sigma,\tau),\ldots,DC(\beta_n,\sigma,\tau)\big) & \\ 0\quad\text{otherwise.}\end{cases}$$

36. *Message receive 2*

$$\frac{-}{\big([\underset{\sim}{p}?m(v_1,\ldots,v_n|E_{rc})\{E\}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{p\,?m[data]}\boldsymbol{\pi}}\ \text{COMM}_4$$

where

$$\boldsymbol{\pi}.c=\begin{cases}[\![E]\!].t.s' & \\ \quad\text{for } c=\big([\surd]^{\psi'}_{C},\sigma',\tau'\big)\text{ if } [\![E_{rc}]\!].s.(s,true)=1,\ Timeless(E_{rc},s), & \\ \quad \underline{p}?m(\underline{n})\text{ in } MSS(C),\ s=\big(\sigma,(proc,\psi),\tau\big), & \\ \quad s'=((\sigma',(proc,\psi'),\tau'),\beta),\ data=\big((\beta_1,\sigma_1,\tau_1),\ldots,(\beta_n,\sigma_n,\tau_n)\big) & \\ \quad\text{and } (\beta_i',\sigma_i',\tau_i')=\text{Relabel}_{+\text{MaxId}(\sigma''_{i-1})}(\beta_i,\sigma_i,\tau_i)\text{ for } i\in\{1,\ldots,n\} & \\ \quad\text{where } \sigma_i''=\begin{cases}(\sigma''_{i-1}\cup\sigma_i') & \text{if } v_i\in\text{Dom}(\sigma)\\ \quad\big\{(\sigma''_{i-1}\cup\sigma_i').proc\{\beta_i'/v_i\}/proc\big\} & \text{and } v_i\notin\text{Dom}(\psi)\\ \sigma''_{i-1}\cup\sigma_i' & \text{otherwise}\end{cases} & \\ \quad\text{and}\quad \psi_i''=\begin{cases}\psi''_{i-1}\{\beta_i'/v_i\} & \text{if } v_i\in\text{Dom}(\psi)\\ \psi''_{i-1} & \text{otherwise}\end{cases} & \\ \quad\text{and}\quad \tau_i''=\tau''_{i-1}\cup\tau_i' & \\ \quad\text{with } \sigma_0''=\sigma,\ \psi_0''=\psi,\ \tau_0''=\tau\text{ and } t=(\sigma_n'',(proc,\psi_n''),\tau_n'') & \\ 0\quad\text{otherwise.}\end{cases}$$

37. *Delay 1*

$$\frac{-}{\big([\texttt{delay}\ E]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{f}\boldsymbol{\pi}}\ \text{DELAY}_1$$

where

$$\boldsymbol{\pi}.c=\begin{cases}[\![E]\!].s.s' & \\ \quad\text{for } c=\big([\surd(\texttt{delay}\,\widetilde{t}\,)]^{\psi'}_{C},\sigma',\tau'\big)\text{ if } s=\big(\sigma,(proc,\psi),\tau\big) \\ \quad\text{and } s'=\big((\sigma',(proc,\psi'),\tau'),t\big)\text{, such that } t\in Time \\ 0 \quad \text{otherwise.}\end{cases}$$

38. *Terminated statement*

$$\frac{-}{\big([\surd]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd]^{\psi}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}\ \text{TERM}$$

39. *Message send 3*

$$\frac{-}{\big([\underline{\widetilde{p}}!m(E_1,\ldots,E_n)\{E\}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\underline{\widetilde{p}}!m(E_1,\ldots,E_n)\{E\}]^{\psi}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}\ \text{COMM}_5$$

40. *Message receive 3*

$$\frac{-}{\big([\underline{\widetilde{p}}?m(v_1,\ldots,v_n|E_{rc})\{E\}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\underline{\widetilde{p}}?m(v_1,\ldots,v_n|E_{rc})\{E\}]^{\psi}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}\ \text{COMM}_6$$

41. *Method call 5*

$$\frac{\big([S]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S']^{\psi}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}{\big([\,m_{C'}(\,)(w'_1,\ldots,w'_k)[S]^{\psi}\,]^{\psi'}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\,m_{C'}(\,)(w'_1,\ldots,w'_k)[S']^{\psi}\,]^{\psi'}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}\ \text{MC}_5$$

42. *Method call 6*

$$\frac{\big([S]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S']^{\psi}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}{\big([\,[S]^{\psi}\,]^{\psi'}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd([S']^{\psi})]^{\psi'}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}\ \text{MC}_6$$

43. *Sequential composition 2*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S'_1]^{\psi}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}{\big([S_1\,\texttt{;}\,S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd(S'_1\,\texttt{;}\,S_2)]^{\psi}_{C},\sigma\!\upharpoonleft_{t},\tau\big)}\ \text{SEQ}_2$$

44. *Sequential composition 3*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t_1}\big([\surd]^{\psi}_{C},\sigma\!\upharpoonleft_{t_1},\tau\big),\quad \big([S_2]^{\psi}_{C},\sigma\!\upharpoonleft_{t_1},\tau\big)\xrightarrow{t_2}\big([S'_2]^{\psi}_{C},\sigma\!\upharpoonleft_{t_1+t_2},\tau\big)}{\big([S_1\,\texttt{;}\,S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t_1+t_2}\big([S'_2]^{\psi}_{C},\sigma\!\upharpoonleft_{t_1+t_2},\tau\big)}\ \text{SEQ}_3$$

45. *Select 3*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_1']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big),\quad \big([S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_2']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}{\big([\texttt{sel}\ S_1\ \texttt{or}\ S_2\ \texttt{les}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd(\texttt{sel}\ S_1'\ \texttt{or}\ S_2'\ \texttt{les})]^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}\ \text{SEL}_3$$

46. *Guarded command 2*

$$\frac{-}{\big([[E]S]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([[E]S]^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}\ \text{GRD}_2$$

if $[\![E]\!].s.(s,\mathit{false})=1$ and $\mathit{Timeless}(E,s)$ where $s=\big(\sigma,(\mathit{proc},\psi),\tau\big)$.

47. *Guarded command 3*

$$\frac{\big([S]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}{\big([[E]S]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd([E]S')]^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}\ \text{GRD}_3$$

if $[\![E]\!].s.(s,\mathit{true})=1$ and $\mathit{Timeless}(E,s)$ where $s=\big(\sigma,(\mathit{proc},\psi),\tau\big)$.

48. *Abort 3*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_1']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big),\quad \big([S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_2']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}{\big([\texttt{abort}\ S_1\ \texttt{with}\ S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd(\texttt{abort}\ S_1'\ \texttt{with}\ S_2')]^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}\ \text{ABORT}_3$$

49. *Interrupt 5*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_1']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big),\quad \big([S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_2']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}{\big([\texttt{interrupt}\ S_1\ \texttt{with}\ S_2,S_3]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd(\texttt{interrupt}\ S_1'\ \texttt{with}\ S_2',S_3)]^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}\ \text{INTR}_5$$

50. *Interrupt 6*

$$\frac{\big([S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_2']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}{\big([S_1\ \texttt{interrupted by}\ S_2,S_3]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd(S_1\ \texttt{interrupted by}\ S_2',S_3)]^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}\ \text{INTR}_6$$

51. *Parallel composition 3*

$$\frac{\big([S_1]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_1']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big),\quad \big([S_2]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([S_2']^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}{\big([\texttt{par}\ S_1\ \texttt{and}\ S_2\ \texttt{rap}]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t}\big([\surd(\texttt{par}\ S_1'\ \texttt{and}\ S_2'\ \texttt{rap})]^{\psi}_{C},\sigma\!\uparrow_t,\tau\big)}\ \text{PAR}_3$$

52. *Delay 2*

$$\frac{-}{\big([\texttt{delay}\ \widetilde{t}\,]^{\psi}_{C},\sigma,\tau\big)\xrightarrow{t'}\big([\surd(\texttt{delay}\ \widetilde{t-t'})]^{\psi}_{C},\sigma\!\uparrow_{t'},\tau\big)}\ \text{DELAY}_2$$

if $t'\leq t$.

A.3 The Architecture Layer of POOSL

The semantic rules in this section complete the timed probabilistic labelled transition system $(Conf, Act, \xrightarrow[Act]{}, Time, \xrightarrow[Time^+]{})$ defined in Sections 4.2.3 and 5.2.3.

53. *Process initialisation*

$$\frac{-}{C(PE_1,\ldots,PE_r) \xrightarrow{\tau} \boldsymbol{\pi}}\ \text{PROC}$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \sum\limits_{\text{P1}} \cdots \sum\limits_{\text{P}r} [\![PE_1]\!].s.s'_1 \times [\![PE_2]\!].s_1.s'_2 \times \cdots \times [\![PE_r]\!].s_{r-1}.s'_r \\ \qquad \text{where } s = (\sigma, (proc, \psi), \tau),\ \mathrm{Dom}(\sigma) = \mathrm{Dom}(\psi) = \mathrm{Dom}(\tau) = \varnothing \text{ and} \\ \qquad\qquad \text{P}i\ :\quad s_i = (\sigma_i, (proc, \psi), \tau_i),\ s'_i = \big(s_i, \beta_i\big) \text{ for } i \in \{1,\ldots,r\} \\ \qquad \text{for } c = \big([m_{C'}(E_1,\ldots,E_q)(\,)]^{\psi}_{C}, \sigma', \tau'\big) \\ \qquad \text{if } CDList \equiv CD_1 \cdots CD_i \cdots CD_n \\ \qquad \text{and } CD_i \equiv \begin{array}{ll} \textbf{process class} & C(y_1,\ldots,y_r) \\ [\,\textbf{extends} & C_{super}\,] \\ \textbf{port interface} & \underline{p_1} \cdots \underline{p_g} \\ \textbf{message interface} & \underline{ms_1} \cdots \underline{ms_h} \\ \textbf{instance variables} & x_1 \cdots x_n \\ \textbf{initial method call} & m_{C'}(E_1,\ldots,E_q)(\,) \\ \textbf{instance methods} & MD^p_1 \cdots MD^p_k \end{array} \\ \qquad \text{with } \mathrm{Dom}(\sigma') = \mathcal{V}(C) \\ \qquad \text{and } \sigma' = \sigma_r\big\{\sigma_r.proc\{nil/x\}/proc\big\} \text{ for } x \in \mathcal{V}(C) \backslash \{y_1,\ldots,y_r\} \\ \qquad \text{and } \sigma' = \sigma_r\big\{\sigma_r.proc\{\beta_i/y_i\}/proc\big\} \text{ for } i \in \{1,\ldots,r\} \text{ and } \tau' = \tau_r \\ 0 \qquad \text{otherwise.} \end{cases}$$

54. *Cluster initialisation*

$$\frac{-}{C(PE_1,\ldots,PE_r) \xrightarrow{\tau} \boldsymbol{\pi}}\ \text{CLUS}$$

where

$$\boldsymbol{\pi}.c = \begin{cases} 1 \qquad \text{for } c = BSpec[\![PE_1/y_1,\ldots,PE_r/y_r]\!] \\ \qquad \text{and } CDList \equiv CD_1 \cdots CD_i \cdots CD_n \\ \qquad \text{and } CD_i \equiv \begin{array}{ll} \textbf{cluster class} & C(y_1,\ldots,y_r) \\ \textbf{port interface} & \underline{p_1} \cdots \underline{p_g} \\ \textbf{message interface} & \underline{ms_1} \cdots \underline{ms_h} \\ \textbf{behaviour specification} & BSpec^b \end{array} \\ 0 \qquad \text{otherwise.} \end{cases}$$

55. *Parallel composition 4*

$$\frac{BSpec_1 \xrightarrow{a} \boldsymbol{\pi}_1}{BSpec_1 \parallel BSpec_2 \xrightarrow{a} \boldsymbol{\pi}}\ \text{PAR}_4$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}_1.BSpec'_1 \\ \qquad \text{for } c = BSpec'_1 \parallel BSpec_2 \\ 0 \qquad \text{otherwise.} \end{cases}$$

56. *Parallel composition 5*

$$\frac{BSpec_2 \xrightarrow{a} \boldsymbol{\pi}_2}{BSpec_1 \parallel BSpec_2 \xrightarrow{a} \boldsymbol{\pi}} \text{ PAR}_5$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}_2.BSpec_2' \\ \qquad \text{for } c = BSpec_1 \parallel BSpec_2' \\ 0 \qquad \text{otherwise.} \end{cases}$$

57. *Parallel composition 6*

$$\frac{BSpec_1 \xrightarrow{\ell} \boldsymbol{\pi}_1, BSpec_2 \xrightarrow{\overline{\ell}} \boldsymbol{\pi}_2}{BSpec_1 \parallel BSpec_2 \xrightarrow{\tau} \boldsymbol{\pi}} \text{ PAR}_6$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}_1.BSpec_1' \times \boldsymbol{\pi}_2.BSpec_2' \\ \qquad \text{for } c = BSpec_1' \parallel BSpec_2' \\ 0 \qquad \text{otherwise.} \end{cases}$$

58. *Port hiding 1*

$$\frac{BSpec \xrightarrow{a} \boldsymbol{\pi}'}{BSpec \setminus L \xrightarrow{a} \boldsymbol{\pi}} \text{ HIDE}_1$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.BSpec' \\ \qquad \text{for } c = BSpec' \setminus L \text{ if } Port(a) \notin L \\ 0 \qquad \text{otherwise.} \end{cases}$$

59. *Port relabelling 1*

$$\frac{BSpec \xrightarrow{a} \boldsymbol{\pi}'}{BSpec[f] \xrightarrow{f(a)} \boldsymbol{\pi}} \text{ RELAB}_1$$

where

$$\boldsymbol{\pi}.c = \begin{cases} \boldsymbol{\pi}'.BSpec' \\ \qquad \text{for } c = BSpec'[f] \\ 0 \qquad \text{otherwise.} \end{cases}$$

60. *Parallel composition 7*

$$\frac{BSpec_1 \xrightarrow{t} BSpec_1'\!\upharpoonleft_t, \quad BSpec_2 \xrightarrow{t} BSpec_2'\!\upharpoonleft_t}{BSpec_1 \parallel BSpec_2 \xrightarrow{t} BSpec_1'\!\upharpoonleft_t \parallel BSpec_2'\!\upharpoonleft_t} \text{ PAR}_7$$

if $\neg Urgent(BSpec_1 \parallel BSpec_2, t)$.

61. *Port hiding 2*

$$\frac{BSpec \xrightarrow{t} BSpec'{\upharpoonleft}_t}{BSpec \setminus L \xrightarrow{t} BSpec'{\upharpoonleft}_t \setminus L}\ \textsc{hide}_2$$

62. *Port relabelling 2*

$$\frac{BSpec \xrightarrow{t} BSpec'{\upharpoonleft}_t}{BSpec[f] \xrightarrow{t} BSpec'{\upharpoonleft}_t[f]}\ \textsc{relab}_2$$

A.4 Extension for Prototyping with POOSL

The semantic rules in this section extend the semantics for prototyping as discussed in Section 7.6.

63. *Parallel composition 8*

$$\frac{\mathit{BSpec} \xrightarrow{a} \boldsymbol{\pi}}{\zeta \parallel \mathit{BSpec} \xrightarrow{a} \boldsymbol{\pi}^e} \ \mathrm{PAR}_8$$

where

$$\boldsymbol{\pi}^e . c^e = \begin{cases} \boldsymbol{\pi}.c & \text{for } c^e = \zeta \parallel c \\ 0 & \text{otherwise.} \end{cases}$$

64. *Parallel composition 9*

$$\frac{\zeta \xrightarrow{\not{t}} \zeta'}{\zeta \parallel \mathit{BSpec} \xrightarrow{\not{t}} \zeta' \parallel \mathit{BSpec}'} \ \mathrm{PAR}_9$$

if $\neg \mathit{Urgent}(\mathit{BSpec})$ and $\mathit{BSpec}' = \mathit{IncorporateExtData}\big(\mathit{BSpec}, \zeta'\big)$.

65. *Parallel composition 10*

$$\frac{\zeta \xrightarrow{t} \zeta', \quad \mathit{BSpec} \xrightarrow{t} \mathit{BSpec}' {\upharpoonleft}_t}{\zeta \parallel \mathit{BSpec} \xrightarrow{t} \zeta' \parallel \mathit{BSpec}' {\upharpoonleft}_t} \ \mathrm{PAR}_{10}$$

if $\neg \mathit{Urgent}(\zeta, t)$.

Appendix B

Concrete Syntax of POOSL

B.1 Extended BNF

This section gives a brief summary of the syntactic metalanguage EBNF, standardised in [26].

The following *syntax rule* defines a language containing strings ε (the empty string), `b a`, `b b a a`, `b b b a a a`, . . . :

```
    A = | "b", A, "a";
```

It defines two alternatives, separated by `|`. The first alternative is the empty sequence represented by the (invisible) empty sequence between `=` and `|`. The second alternative is any sequence starting with `b`, followed by another string produced by `A`, and terminated by `a`.

The list of symbols used in EBNF are (with decreasing precedence): `-` exception symbol, `,` concatenate symbol, `|` definition separator symbol, `=` defining symbol, `;` terminator symbol. The normal precedence is overridden by the following bracket pairs: `' '` first quote pair, `" "` second quote pair, `( )` grouping, `[ ]` optional sequences, `{}` repetition, and `? ?` special sequences.

Examples:

```
    B = {"a", "b"}-;
```

defines the sequences consisting of a repetition of subsequence `ab`, except the empty sequence (`a b, a b a b, a b a b a b, ...`).

```
    C = ?ISO 6429 character Form Feed? | '"';
```

defines the sequence consisting of a single form feed character as defined in [25] or a quotation mark (`"`).

In general, a language is defined by a set of syntax rules. The following sections present the syntax for POOSL specifications, represented by the start symbol `system specification`.

B.2 Keywords

`abort`	`do`	`les`	`port`	`system`
`and`	`else`	`message`	`process`	`then`
`behaviour`	`extends`	`method`	`primitive`	`variables`
`call`	`fi`	`methods`	`rap`	`while`
`class`	`if`	`new`	`return`	`with`
`cluster`	`initial`	`nil`	`sel`	
`currentTime`	`instance`	`od`	`self`	
`data`	`interface`	`or`	`skip`	
`delay`	`interrupt`	`par`	`specification`	

B.3 Operator Precedence

precedence	operator	equivalent method name
highest	`-`	`unaryMinus`
higher	`*`	`multiply`
	`/`	`divide`
medium	`+`	`add`
	`-`	`subtract`
	`&`	`and`
	`|`	`or`
lowest	`=`	`equals`
	`!=`	`equalsNot`
	`==`	`isIdenticalWith`
	`!==`	`isNotIdenticalWith`
	`<`	`lessThan`
	`<=`	`lessOrEqual`
	`>`	`moreThan`
	`>=`	`moreOrEqual`

B.4 Characters, Literals and Identifiers

The characters and special control sequences used in this section have been standardised in [24] and [25] respectively.

```
letter
    = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
    | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
    | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
    | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";

binary digit
    = "0" | "1";

octal digit
    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";

decimal digit
    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

hexadecimal digit
    = decimal digit
    | "a" | "b" | "c" | "d" | "e" | "f" | "A" | "B" | "C" | "D" | "E" | "F";
```

```
new line
    = [?ISO 6429 character Carriage Return?], ?ISO 6429 character Line Feed?
    | ?ISO 6429 character Carriage Return?, [?ISO 6429 character Line Feed?];

horizontal tab
    = ?ISO 6429 character Horizontal Tab?;

white space
    = " " | new line | horizontal tab;

other character
    = "!" | ’"’ | "#" | "$" | "%" | "&" | "’" | "(" | ")" | "*" | "+" | ","
    | "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "\"
    | "]" | "^" | "_" | "‘" | "{" | "|" | "}" | "~";

identifier
    = letter, {letter | decimal digit | "_"};

class name
    = identifier;

class names
    = class name, {",", class name};

message name
    = identifier;

method name
    = identifier;

port name
    = identifier;

port names
    = port name, {",", port name};

variable
    = identifier;

boolean
    = "true" | "false";

integer
    = {decimal digit}-
    | "0b", {binary digit}-
    | "0o", {octal digit}-
    | "0x", {hexadecimal digit}-;

real
    = {decimal digit}-, ".", {decimal digit}-,
      [("e" | "E"), ["+" | "-"], {decimal digit}-];

string
    =’"’, {letter | decimal digit | white space | other character - ’"’ | ’""’}, ’"’;
```

B.5 Expressions

```
variables
    = variable, {",", variable};

data method call
    = ["^"], method name, ["(", [list of expressions], ")"];

declaration
    = variables, ":", class name;

declarations
    = declaration, {",", declaration};

operator
    = "+" | "-" | "*" | "/" | "!=" | "==" | "!=="
    | "&" | "|" | "<" | ">" | "<=" | ">=" | "=";

primary
    = variable
    | boolean | integer | real | string
    | "nil" | "self" | "currentTime"
    | "new", "(", class name, ")"
    | "(", expressions, ")";

expression
    = ["-"], primary, {data method call}
    | expression, operator, expression
    | variable, ":=", expression
    | "if", expressions, "then", expressions, ["else", expressions], "fi"
    | "while", expressions, "do", expressions, "od"
    | "return", expression;

expressions
    = expression, {";", expression};

list of expressions
    = expressions, {",", expressions};

port expression
    = port name
    | expression
    | "{", expressions, "}";
```

B.6 Data Methods and Classes

```
data method definition
    = (method name | operator), ["(", [declarations], ")"], ":", class name,
      ["|", [declarations], "|"],
      ("primitive" | expressions), ".";

data class definition
    = "data", "class", class name,
      ["extends", class name],
      "instance", "variables", [declarations],
      "instance", "methods", {data method definition};
```

B.7 Statements

```
message
    = port expression, "?", message name,
      ["(", [variables], ["|", expressions], ")"], ["{", expressions, "}"]
    | port expression, "!", message name,
       ["(", [list of expressions], ")"], ["{", expressions, "}"];

process method call
    = method name, "(", [list of expressions], ")", "(", [variables], ")";

statement
    = expression
    | "{", expressions, "}"
    | "delay", expression
    | message
    | process method call
    | "skip"
    | "[", expressions, "]", statement
    | "if", expressions, "then", statements, ["else", statements], "fi"
    | "while", expressions, "do", statements, "od"
    | "par", statements, {"and", statements}-, "rap"
    | "sel", statements, {"or", statements}-, "les"
    | "abort", statements, "with", statement
    | "interrupt", statements, "with", statement
    | "(", statements, ")";

statements
    = statement, {";", statement};
```

B.8 Process Methods and Classes

```
instantiation parameters
    = declarations;

message signature
    = port name, "?", message name, ["(", [class names], ")"]
    | port name, "!", message name, ["(", [class names], ")"];

message signatures
    = message signature, {",", message signature};

process method definition
    = method name, "(", [declarations], ")", "(", [declarations], ")",
      ["|", [declarations], "|"],
      statements, ".";

process class definition
    = "process", "class", class name, ["(", [instantiation parameters], ")"],
      ["extends", class name],
      "port", "interface", [port names],
      "message", "interface", [message signatures],
      "instance", "variables", [declarations],
      "initial", "method", "call", process method call,
      "instance", "methods", {process method definition};
```

B.9 Cluster Classes

```
hidings
    = "\", "{", [port names], "}";

relabelling
    = port name, "/", port name;

relabellings
    = "[", [relabelling, {",", relabelling}], "]";

specification primary
    = identifier, ":", class name, ["(", [list of expressions], ")"],
      [hidings], [relabellings];

specification primaries
    = specification primary, {"||", specification primary};

behaviour specification
    = specification primaries
    | "(", specification primaries, ")", [hidings], [relabellings];

cluster class definition
    = "cluster", "class", class name, ["(", [instantiation parameters], ")"],
      "port", "interface", [port names],
      "message", "interface", [message signatures],
      "behaviour", "specification", behaviour specification;
```

B.10 System Specification

```
class definition
    = cluster class definition
    | data class definition
    | process class definition;

system specification
    = "system", "specification", class name,
      "behaviour", "specification", behaviour specification,
      {class definition};
```

Appendix C

Performance of Rotalumis

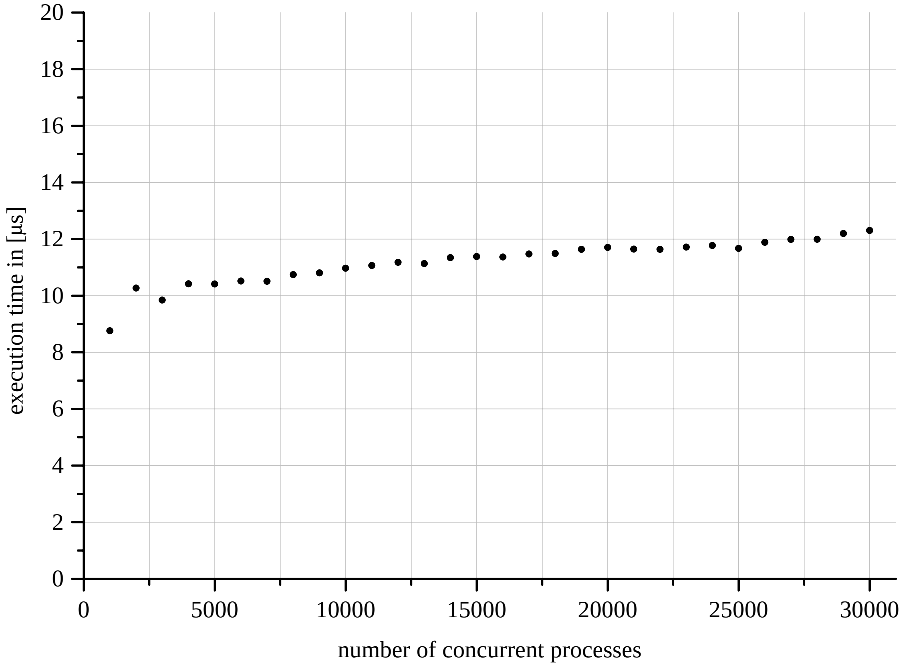


Figure C.1: Performing a communication in the presence of other processes.

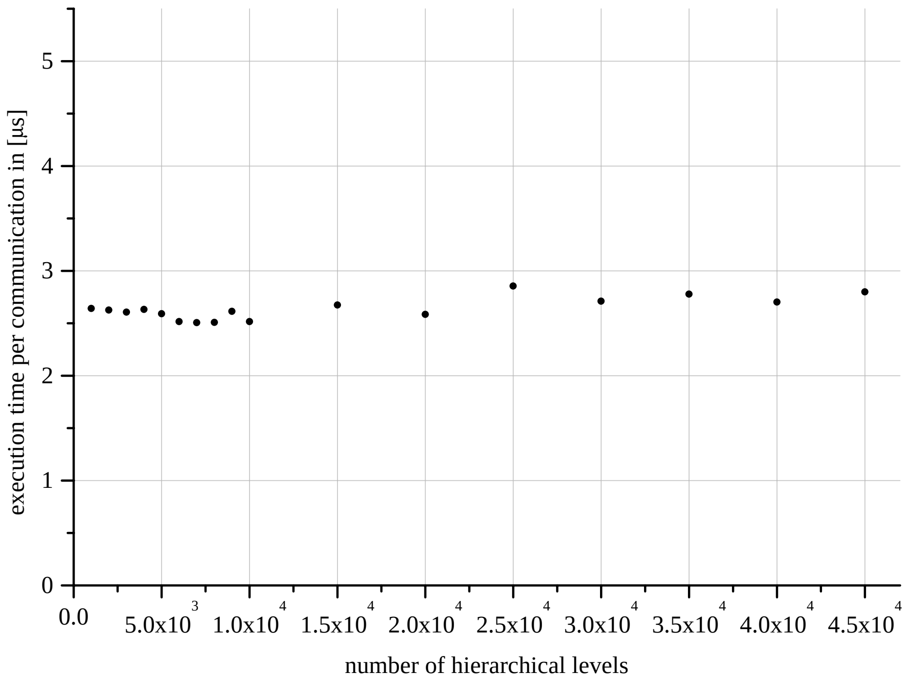


Figure C.2: Communicating through hierarchical levels.

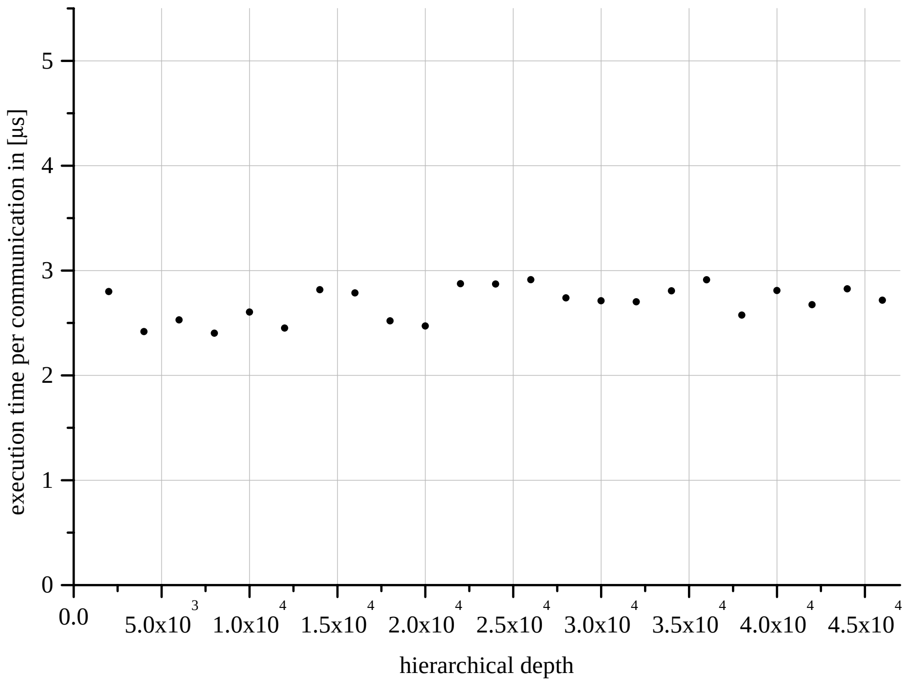


Figure C.3: Communicating at various hierarchical depths.

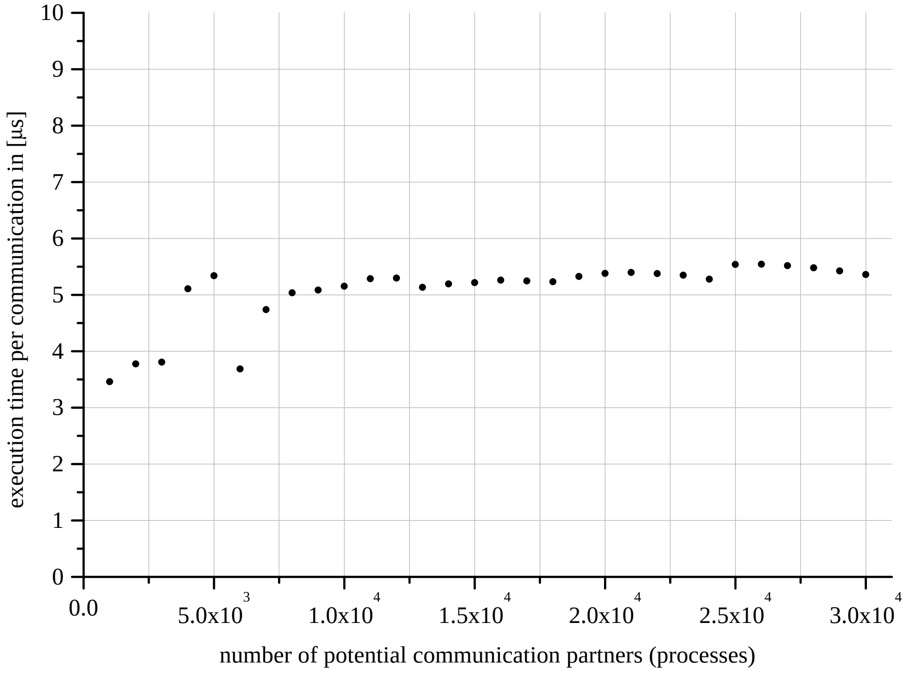


Figure C.4: Transmitting messages to nonblocking processes connected to a single channel.

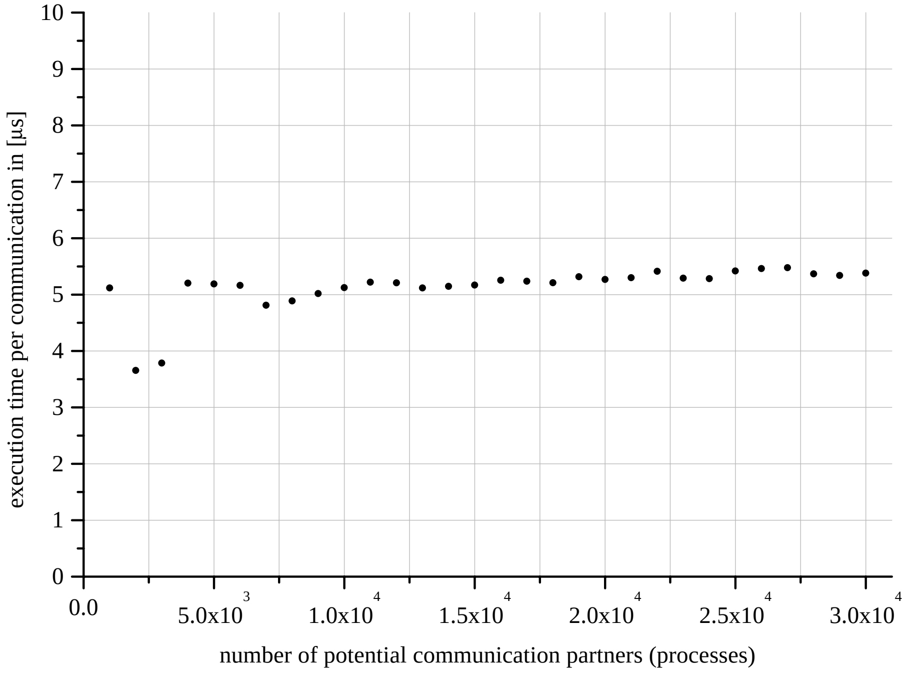


Figure C.5: Receiving messages from nonblocking processes connected to a single channel.

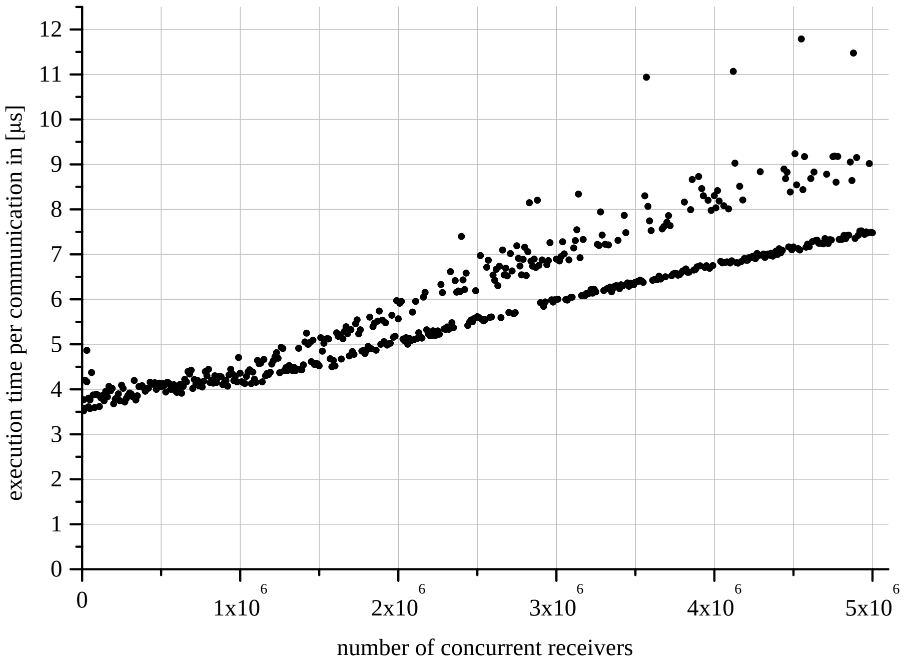


Figure C.6: Transmitting messages to concurrent receivers within a single process.

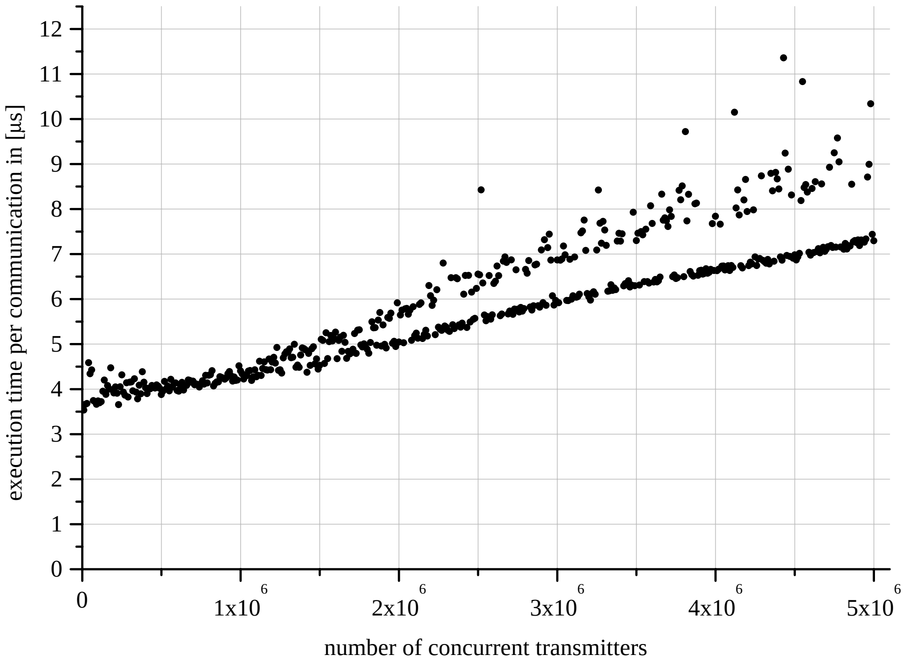


Figure C.7: Receiving messages from concurrent transmitters within a single process.

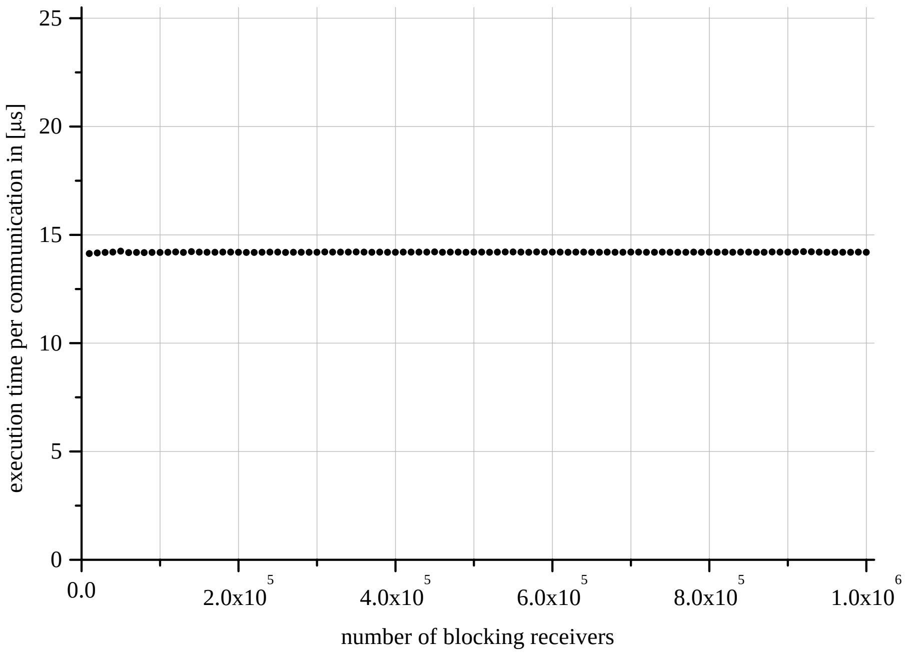


Figure C.8: Conditionally transmitting messages to a single receiver in the presence of blocking receivers.

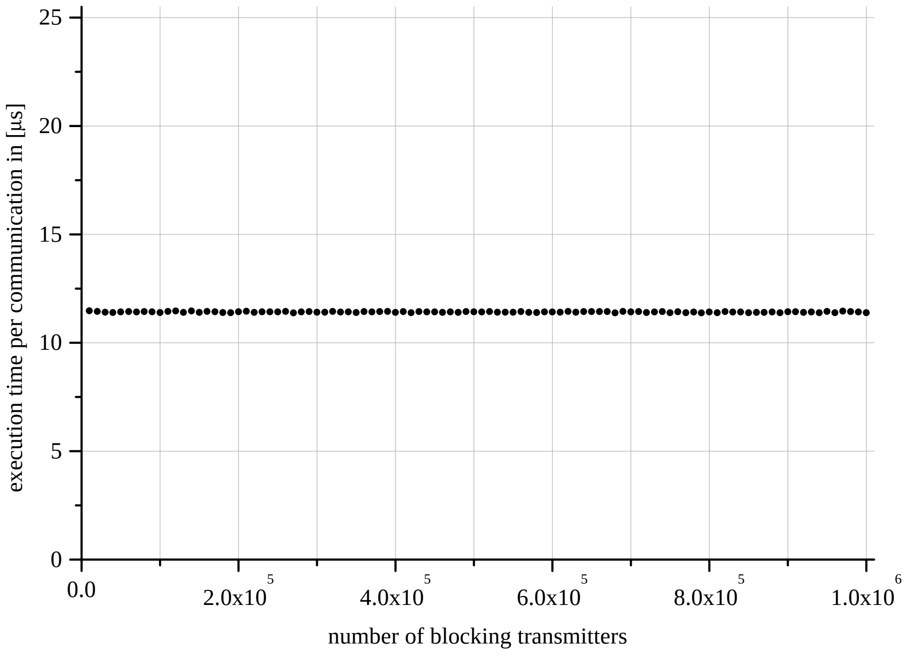


Figure C.9: Conditionally receiving messages from a single transmitter in the presence of blocking transmitters.

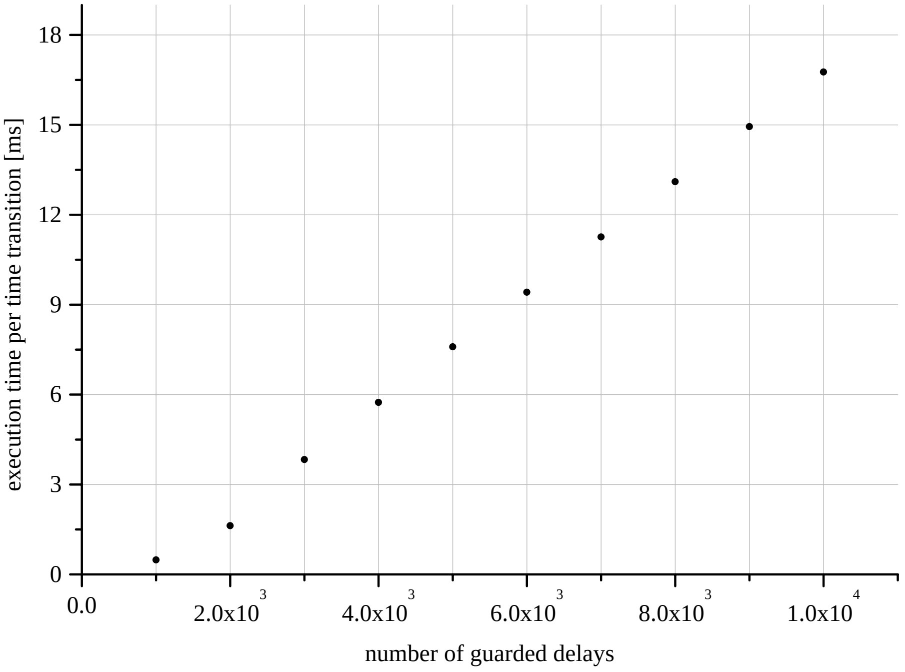


Figure C.10: Advancing time in the presence of guarded delays.

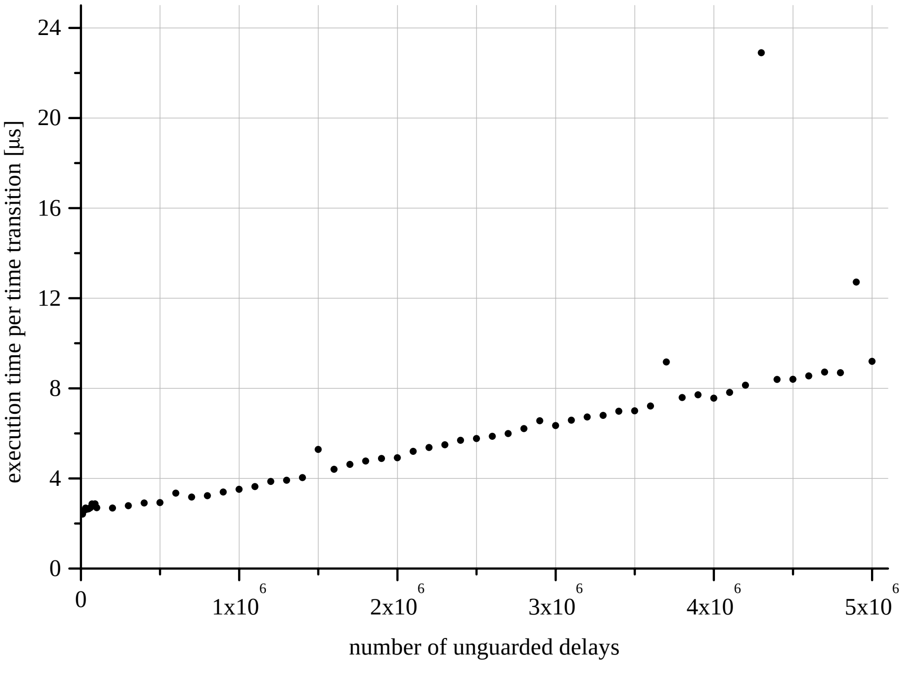


Figure C.11: Advancing time in the presence of unguarded delays.

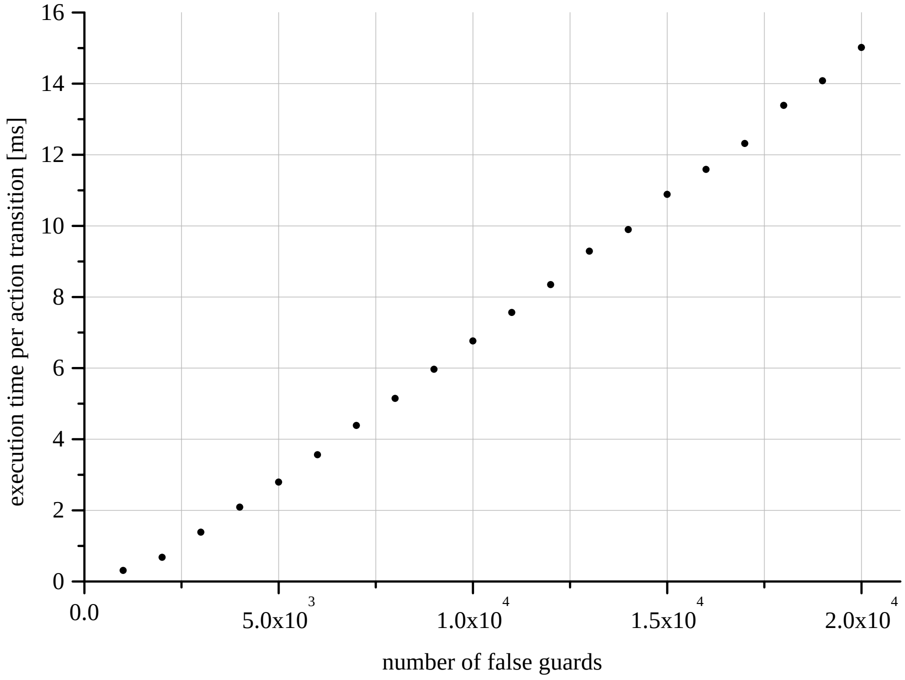


Figure C.12: Executing a task in the presence of false guards.

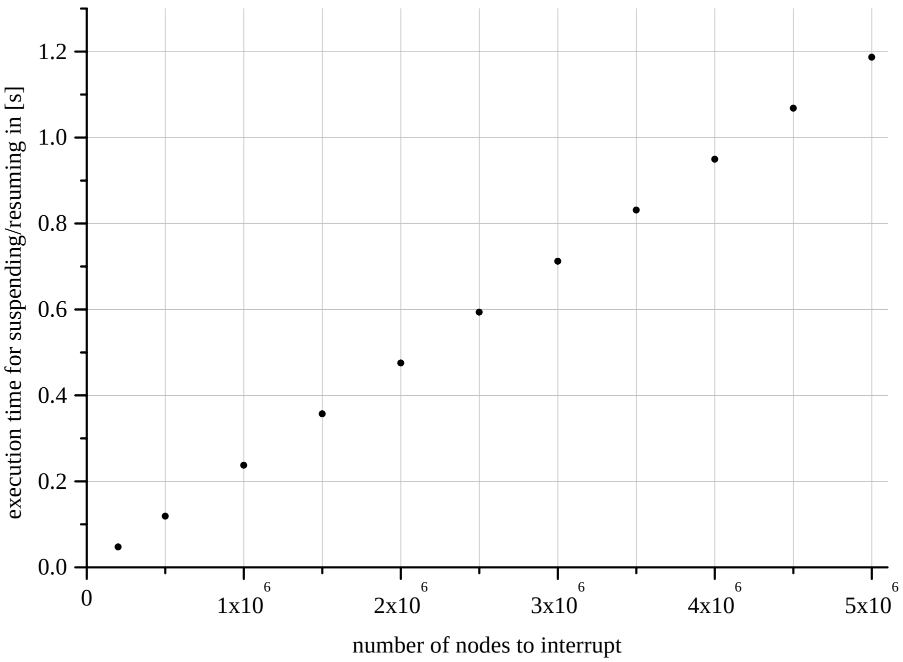


Figure C.13: Suspending or resuming statements with `interrupt`.

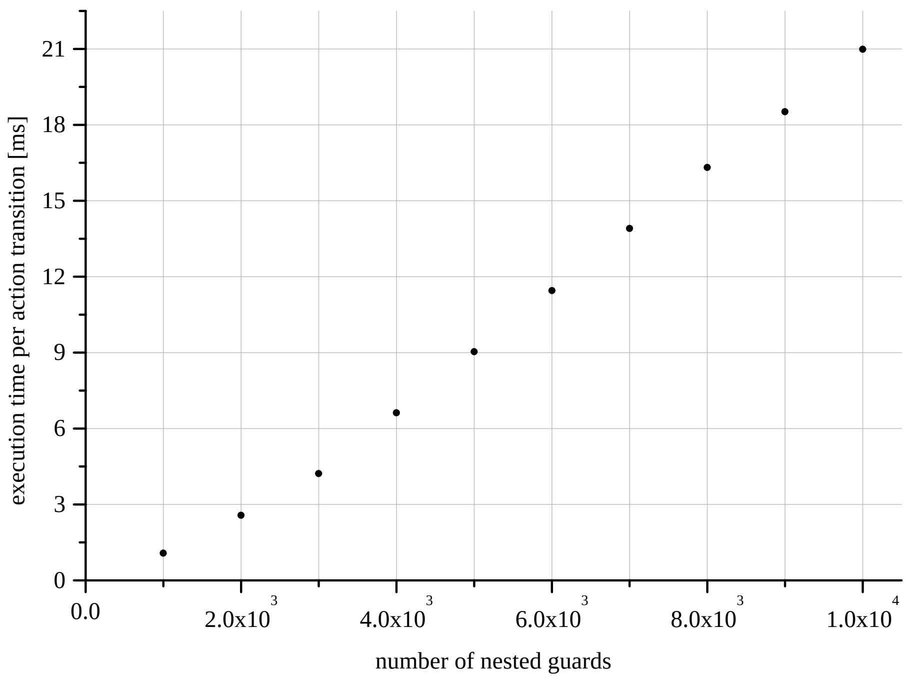


Figure C.14: Executing a task that is nested in guards.

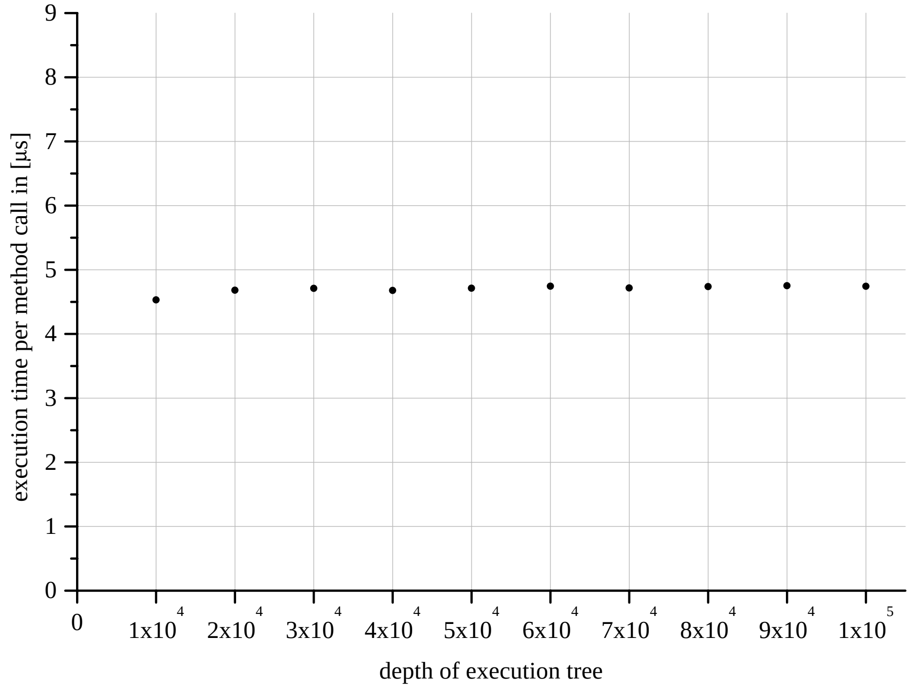


Figure C.15: Calling a method at various depths of the execution tree.

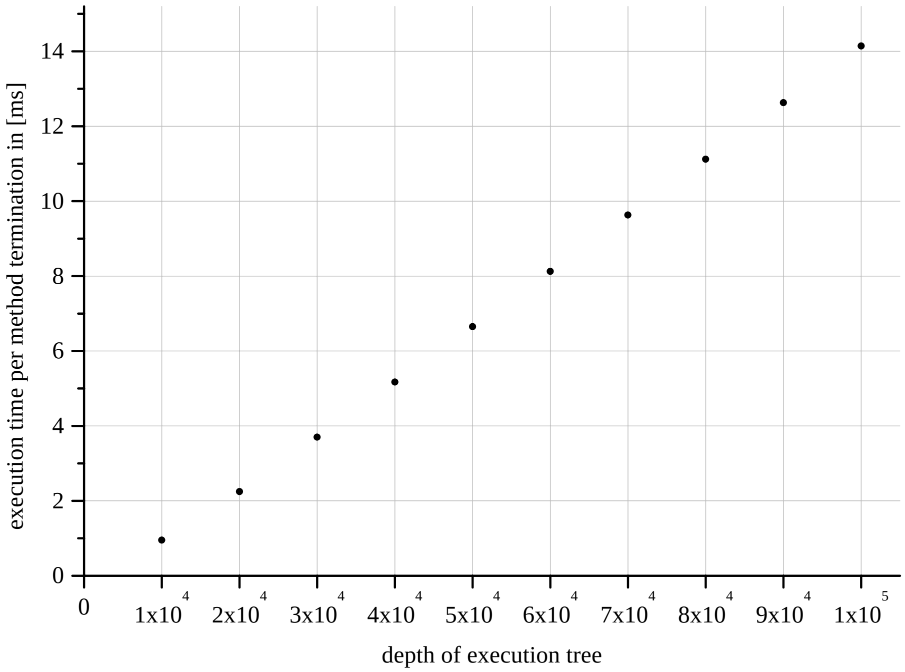


Figure C.16: Terminating a method with return parameters at various depths of the execution tree.

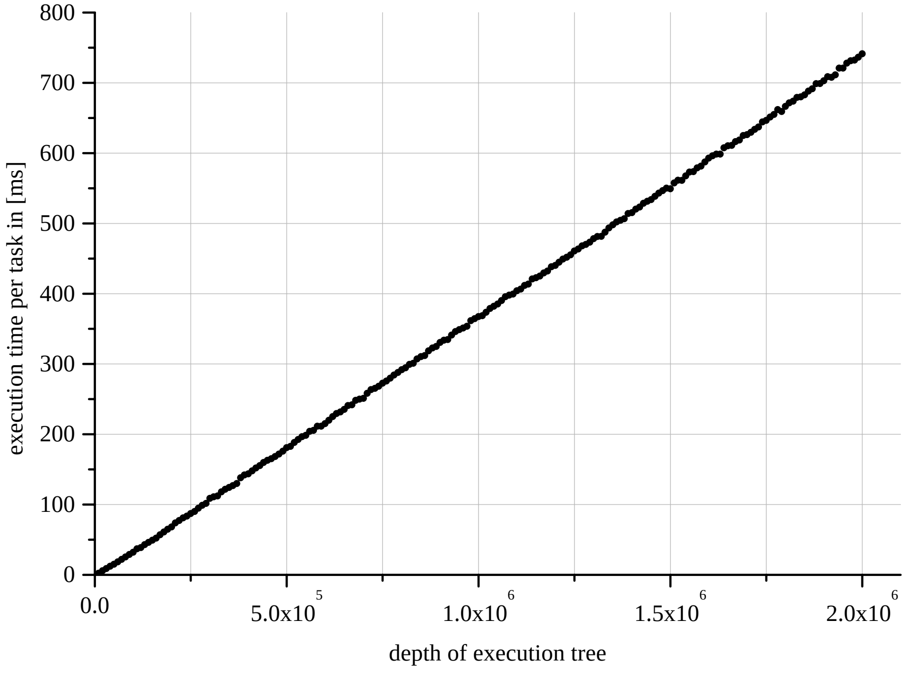


Figure C.17: Executing a task at various depths in the execution tree.

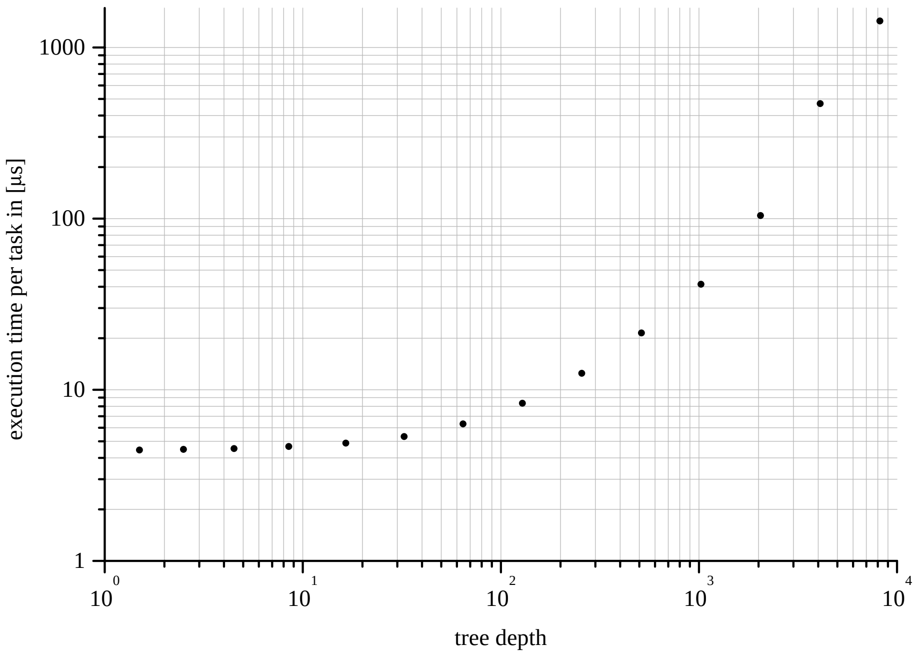


Figure C.18: Executing a task at various depths in the execution tree (without tree height reduction).

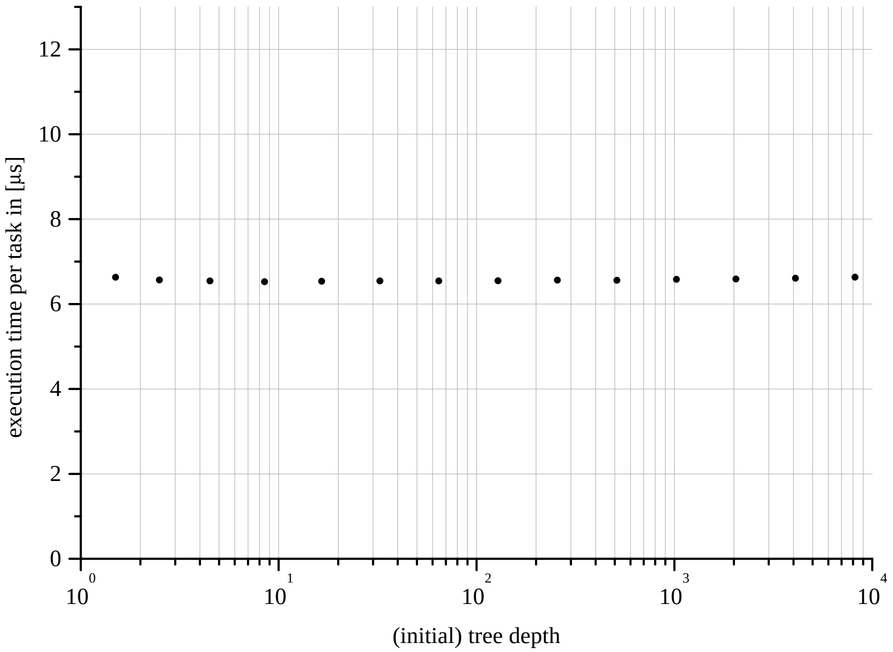


Figure C.19: Executing a task at various depths in the execution tree (with tree height reduction).

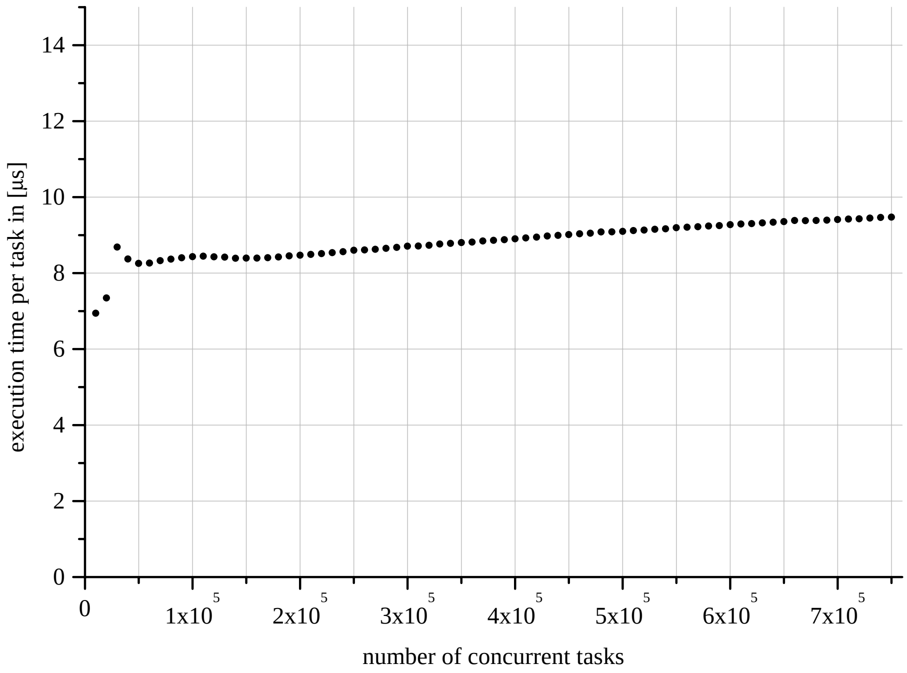


Figure C.20: Concurrently executing computational tasks.

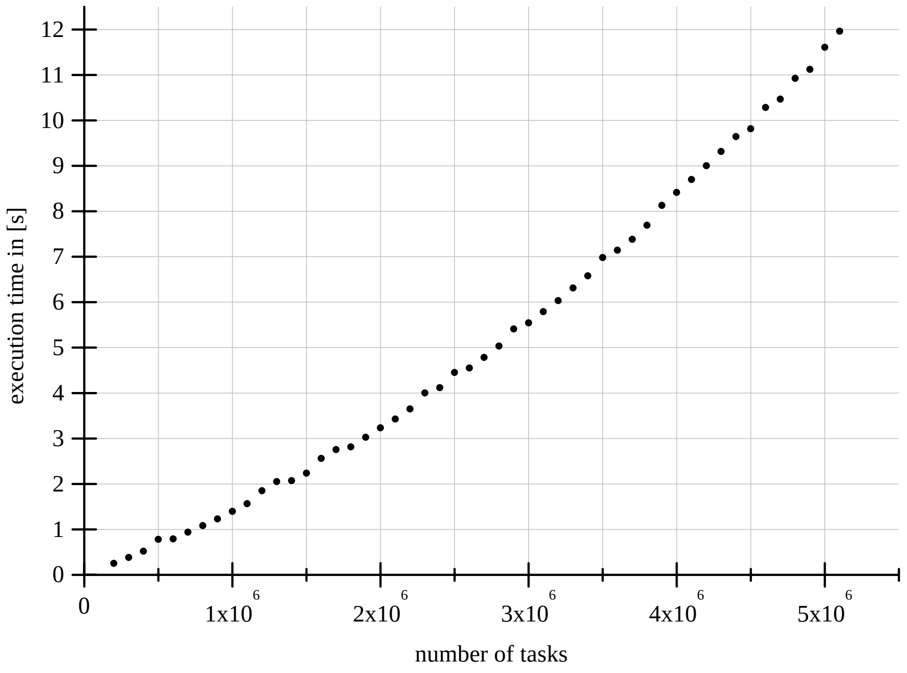


Figure C.21: Creating tasks in `abort`.

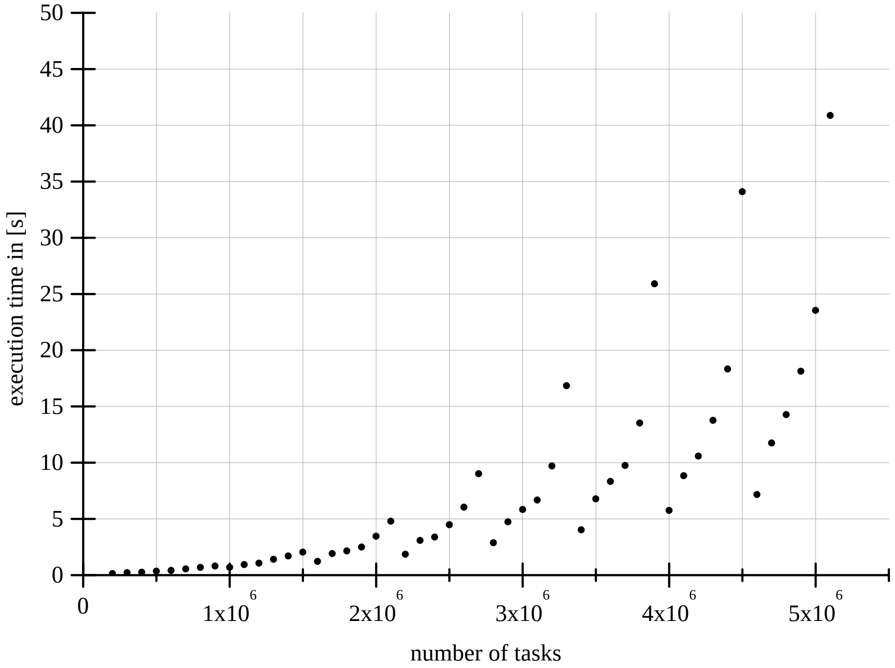


Figure C.22: Destroying tasks in `abort`.

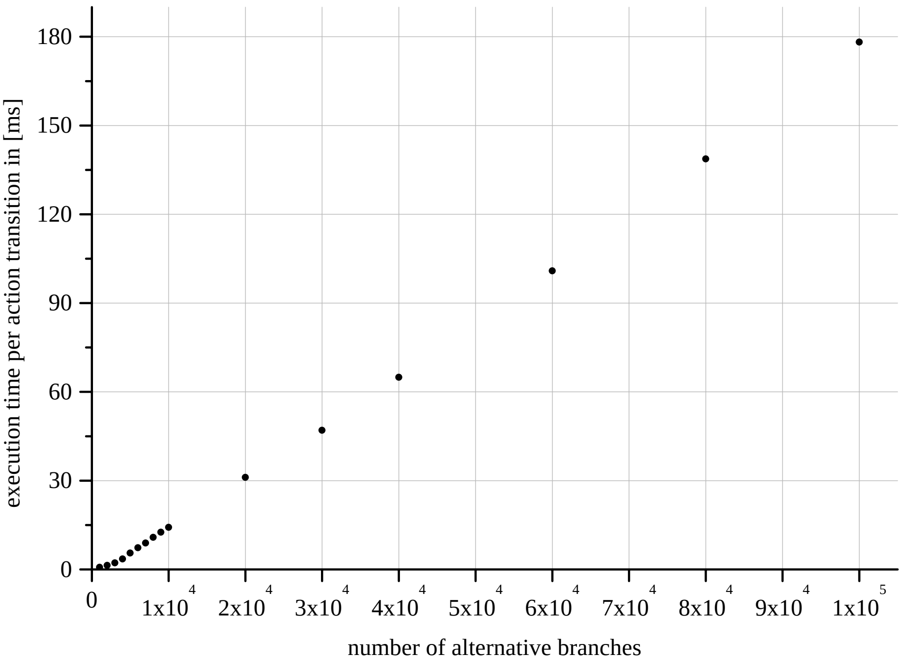


Figure C.23: Choosing a branch from a `select`.

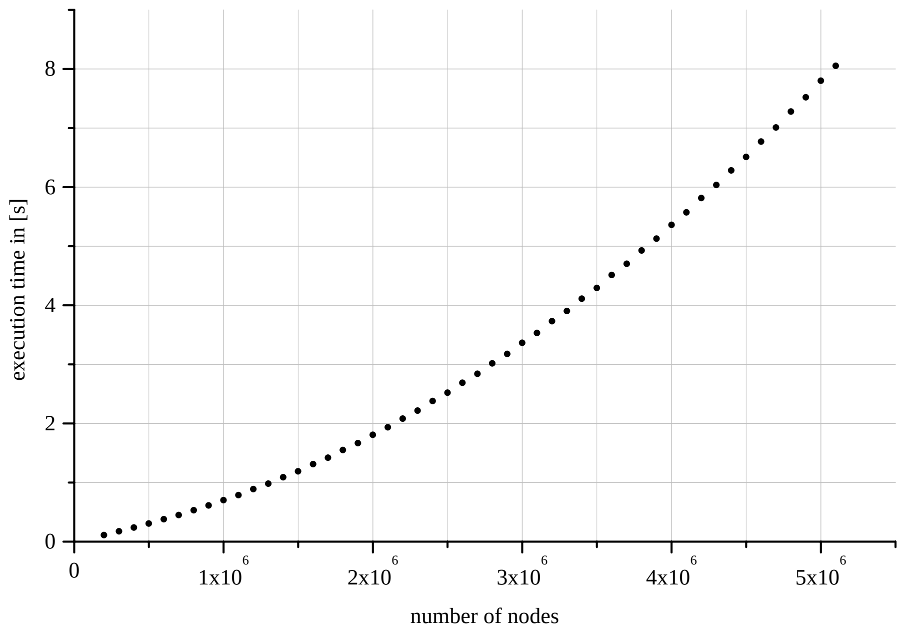


Figure C.24: Creating nodes in C++.

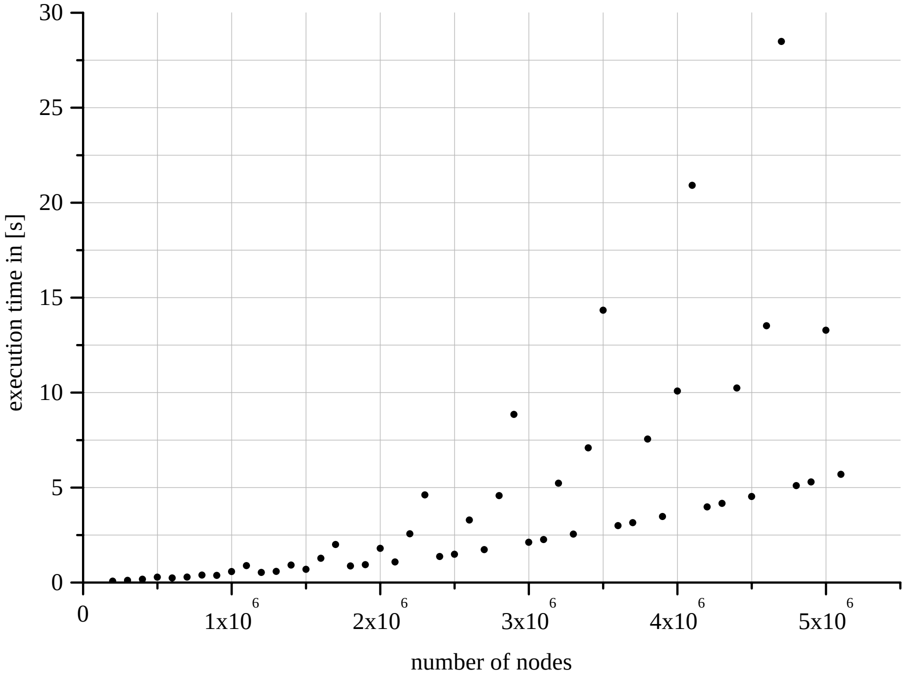


Figure C.25: Destroying nodes in C++.

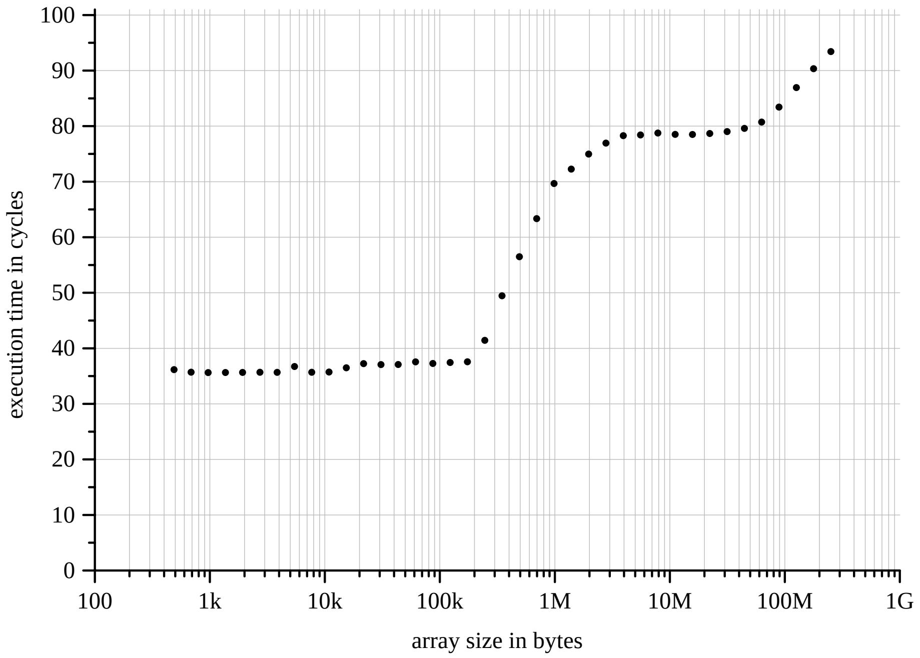


Figure C.26: Incrementing an array element in random order in C++ (including loop overhead).

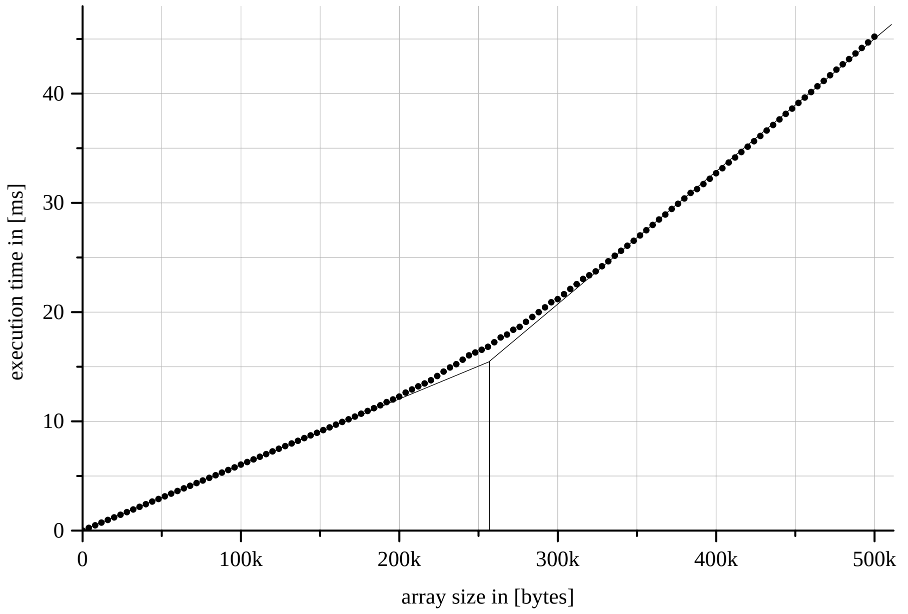


Figure C.27: Incrementing an array element in random order in C++ (detailed view).

Bibliography

[1] H.G. Baker. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. *ACM Sigplan Notices*, 27(3):66–70, March 1992.

[2] G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. Rapports de recherche 842, Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay, 1988.

[3] G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] G. Berry. Real-Time Programming: Special Purpose or General Purpose Languages. *Information Processing 89*, pages 11–17, 1989.

[5] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation Can't Be Traced. *Journal of the ACM (JACM)*, 42(1):232–268, January 1995.

[6] L.J. van Bokhoven, J.P.M. Voeten, and M.C.W. Geilen. Real-time Simulation Using Process Execution Trees. In J.P. Veen, editor, *Proceedings of the ProRISC / IEEE Workshop on Circuits, Systems and Signal Processing (CSSP98) and STW's Workshop on Semiconductor Advances for Future Electronics (SAFE98)*, pages 51–55, Mierlo, The Netherlands, 25-27 November 1998. STW, Technology Foundation.

[7] L.J. van Bokhoven, J.P.M. Voeten, and M.C.W. Geilen. Software Synthesis for System Level Design Using Process Execution Trees. In B. Werner, editor, *Proceedings of the 25th EUROMICRO Conference (EUROMICRO 99).*, pages 463–467, Los Alamitos, California, USA, 8-10 September 1999. IEEE, IEEE Computer Society.

[8] V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 2002.

[9] R. Cleaveland and D. Yankelevich. An Operational Framework for Value-Passing Processes. In *Proceeding of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 326–338, New York, 1994. ACM.

[10] G.E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[11] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[12] P.H.J. van Eijk. The Design of a Simulator Tool. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Language LOTOS*, pages 351–390. Elsevier Science Publishers B.V., North-Holland, 1989.

[13] P.H.V. van Eijk, C.A. Visser, and M. Diaz, editors. *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project.* Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1989.

[14] P. van Eijk. A Comparison of Behavioural Language Simulators. In B. Sarikaya and G. von Bochmann, editors, *Proceedings of PSTV86 International Workshop on Protocol Specification, Testing and Verification, June 10-13, Montreal, Quebec, Canada*, pages 85–96, North-Holland, 1986. Elsevier Science Publishers B.V.

[15] T. Gautier, P. le Guernic, and L. Besnard. SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems. Rapports de recherche 761, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, 1987.

[16] M.C.W. Geilen, J.P.M. Voeten, P.H.A. van der Putten, L.J. van Bokhoven, and M.P.J. Stevens. Object-Oriented Modelling and Specification Using SHE. *Journal of Computer Languages*, 27(1-3):19–38, April-October 2001.

[17] M.C.W. Geilen. Real-Time Concepts for Software/Hardware Engineering. Master's thesis, Department of Electrical Engineering, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1996.

[18] M.C.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems.* PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 2002.

[19] J.F. Groote. *Process Algebra and Structured Operational Semantics.* PhD thesis, Universiteit van Amsterdam, The Netherlands, 1991.

[20] J.F. Groote. The Syntax and Semantics of Timed μCRL. Technical Report SEN-R9709, Centrum voor Wiskunde en Informatica, The Netherlands, 1997.

[21] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[22] J.I. den Hartog. Verifying Probabilistic Programs Using a Hoare-like Logic. In P.S. Thiagarajan and R. Yap, editors, *Proceeding ASIAN '99*, pages 113–125, LNCS 1742, 1999. Springer.

[23] C. Huizing, R.T. Gerth, and W.P. de Roever. *A Compositional Semantics for STATECHARTS*. Computing Science Notes; 8715. Technische Universiteit Eindhoven, The Netherlands, 1987.

[24] ISO/IEC. Information Technology — ISO 7-bit Coded Character Set for Information Interchange. Technical Report 646, 1991.

[25] ISO/IEC. Information Technology — Control Functions for Coded Character Sets. Technical Report 6429, 1992.

[26] ISO/IEC. Information Technology — Syntactic Metalanguage — Extended BNF. Technical Report 14977, 1996.

[27] S. Keshav. *An Engineering Approach to Computer Networking; ATM Networks, the Internet and the Telephone Network*. Addison-Wesley, Reading, Massachusetts, U.S.A., 1998.

[28] D.H. Lehmer. Mathematical Methods in Large-Scale Computing Units. In *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery, 1949*, pages 141–146, Cambridge, Mass., 1951. Harvard University Press.

[29] L.M.B. Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, University of Porto, 1999.

[30] L. Lopes, F. Silva, and V. Vasconcelos. A Virtual Machine for a Process Calculus. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 244–260. Springer-Verlag, 1999.

[31] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator. *ACM Transactions on Modelling and Computer Simulation*, 8(1):3–30, January 1998.

[32] J.H. McBeth. On the Reference Counter Method. *Communications of the ACM*, 6(9):575, September 1963.

[33] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part. *Communications of the ACM*, 3(4):184–195, April 1960.

[34] R. Milner. A Calculus of Communicating Systems. In G. Goos, J. Hartmanis, and J. Van Leeuwen, editors, *Lecture Notes in Computer Science : 92*. Springer-Verlag, Berlin, Germany, 1980.

[35] R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[36] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1999.

[37] X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In K. Larsen and A. Skou, editors, *Proceedings CAV'91 3rd International Workshop Computer Aided Verification, Ålborg, Denmark , July 1991 (LNCS 575)*, pages 376–398, Berlin, 1992. Springer Verlag.

[38] H.R. Nielson and F. Nielson. *Semantics with Applications — A Formal Introduction.* John Wiley & Sons Ltd., England, 1992.

[39] S.K. Park and K.W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[40] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, Aarhus, Denmark, September 1981.

[41] P.H.A. van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware / Software Systems - The Method Software / Hardware Engineering (SHE).* PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1997.

[42] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Massachusetts Institute of Technology, June 1995.

[43] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[44] R.D. Tennent. *Semantics of Programming Languages.* Prentice Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Herfordshire HP2 4RG, 1991.

[45] B.D. Theelen, J.P.M. Voeten, L.J. van Bokhoven, G.G. de Jong, A.M.M. Niemegeers, P.H.A. van der Putten, M.P.J. Stevens, and J.C.M. Baeten. System-Level Modelling and Performance Analysis. In *Proceedings of PROGRESS'00*, pages 141–147, Utrecht, The Netherlands, October 2000. STW Technology Foundation.

[46] B.D. Theelen, J.P.M. Voeten, L.J. van Bokhoven, P.H.A. van der Putten, G.G. Jong, and A.M.M. Niemegeers. Performance Modeling in the Large: A Case Study. In *Proceedings of the European Simulation Symposium (ESS) 2001*, October 2001.

[47] B.D. Theelen and J.P.M. Voeten. Modelling the MPSR Switch System using POOSL; Performance Analysis for System-Level Design. Confidential report of the Alcatel – TUE frame project: Performance Analysis, Deliverable 1.1, Technische Universiteit Eindhoven, The Netherlands, March 2000.

[48] B.D. Theelen and J.P.M. Voeten. Simulating the Evolved POOSL Model of the MPSR Switch System; Performance Analysis for System-Level Design. Confidential report of the Alcatel – TUE frame project: Performance Analysis, Deliverable 1.2, Technische Universiteit Eindhoven, The Netherlands, May 2000.

[49] K.J. Turner, editor. *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL.* John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1993.

[50] I. Ulidowski and I. Phillips. Formats of Ordered SOS Rules with Silent Actions. In M. Bidoit and M. Dauchet, editors, *Proceedings of the 7th International Conference on Theory and Practice of Software Development TAPSOFT'97, Lille, France, LNCS 1214.* Springer Verlag, 1997.

[51] I. Ulidowski and S. Yuen. Extending Process Languages with Time. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology AMAST'97 Sydney, Australia, LNCS 1349*. Springer Verlag, 1997.

[52] J.P.M. Voeten, M.C.W. Geilen, L.J. van Bokhoven, P.H.A. van der Putten, and M.P.J. Stevens. A Probabilistic Real-Time Calculus for Performance Evaluation. In G. Horton, D. Möller, and U. Rüde, editors, *Proceedings of the 11th European Simulation Symposium (ESS'99) (Erlangen, Germany, October 26–28)*, pages 608–617, Delft, the Netherlands, 1999. SCS-Europe.

[53] J.P.M. Voeten, P.H.A. van der Putten, M.C.W. Geilen, and M.P.J. Stevens. Towards System Level Performance Modelling. In J.P. Veen, editor, *Proceedings of ProRISC'98 Mierlo, The Netherlands, November 25–27*, pages 593–597, Utrecht, The Netherlands, 1998. STW Technology Foundation.

[54] J.P.M. Voeten, I.G. Stappers, M.C.W. Geilen, L.J. van Bokhoven, P.H.A. van der Putten, and M.P.J. Stevens. An Analytical Approach towards System Level Performance Analysis. In J.P. Veen, editor, *Proceedings of ProRISC'99 (Mierlo, The Netherlands, November 24–26)*, pages 569–576, Utrecht, The Netherlands, 1999. STW Technology Foundation.

[55] J.P.M. Voeten. Temporal Rewards for Performance Evaluation. In *Proceedings of the 8th International Workshop on Process Algebra and Performance Modelling PAPM'00 (Geneva, Switzerland, July 15)*, pages 511–522, Waterloo, Ontario, Canada, 2000. Carleton Scientific.

[56] F. van Wijk, J.P.M. Voeten, and A.J.W.M. ten Berg. An Abstract Modeling Approach towards System-Level Design-Space Exploration. In *Proceedings of the Forum on Specification and Design Languages (FDL'02)*, Marseille, France, September 24-27, 2002.

[57] P.R. Wilson. Uniprocessor Garbage Collection Techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings Memory Management*, pages 1–42, Berlin, Germany, 19 September 1992. International Workshop IWMM 92, Springer Verlag.

[58] W. Yi. Real-Time Behaviour of Asynchronous Agents. In *Proceedings of CONCUR90 (International Conference on Concurrency Theory), Amsterdam, LNCS 458*. Springer Verlag, 1990.

[59] B. Zorn. The Measured Cost of Conservative Garbage Collection. *Software — Practice and Experience*, 23(7):733–756, July 1993.

Glossary of Symbols

Symbols in the following chapter-wise glossaries have been sorted alphabetically, based on their pronunciation. Symbols whose pronunciation is probably unknown have been collected and listed at the beginning of each list. If a symbol is not present in the glossary of a particular chapter, it might already have been listed in one of the previous glossaries.

Chapter 2 Mathematical Preliminaries

Symbol	Page	Meaning
$\mathbf{2}^A$	16	powerset of A
$\perp$	17	least element of a poset (pronounced "bottom")
$\sqsubseteq$	16	partial order
$\bigsqcup d$	17	least upper bound of chain d
$\restriction$	15	restriction operator
$A \times B$	16	Cartesian product
$A \hookrightarrow B$	15	set of partial functions from A to B
$A \rightarrow B$	15	set of total functions from A to B
$(a_1, \ldots, a_n)$	16	ordered n-tuple
$(a_1, a_2, \ldots)$	16	sequence
A^n	16	$\{1, \ldots, n\} \rightarrow A$ (set of ordered n-tuples)
A^ω	16	$\mathbb{N} \rightarrow A$ (set of sequences)
$d \in D^\omega$	17	d is an ω-chain
$\mathrm{Dom}(f)$	15	domain of function f
$(D, \sqsubseteq)$	16	partially ordered set (poset) or
	17	chain-complete partial order (ccpo)
$f(a)$	15	image of a under function f
$f(a) = \underline{\mathrm{undef}}$	15	function f is undefined for a
f_a	15	image of a under function f
$f.a$	15	image of a under function f
$f\{w/v\}$	15	variant notation, function equal to f except that $f(v) = w$
$\mathrm{FIX}\,\mathcal{F}$	18	least fixed point of $\mathcal{F}$
$\mathbb{N}$	15	natural numbers: $\{1, 2, 3, \ldots\}$
$\mathbb{N}_0$	15	$\{0, 1, 2, 3, \ldots\}$
nil	26	single element of primitive data class *Nil*

Chapter 3 POOSL Data Layer

Symbol	Page	Meaning
$\triangleq$	30	equal by definition
$\equiv$	31	syntactic identity
$\perp_{\mathcal{P}}$	35	least element of ccpo $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$
$\perp_{\mathcal{R}}$	35	least element of ccpo $(\mathcal{R}, \leq)$

$\bot_{\mathcal{S}}$	35	least element of ccpo $(\mathcal{S}, \sqsubseteq_{\mathcal{S}})$
$\bot^n_{\mathcal{S}}$	35	least element of ccpo $(\mathcal{S}^n, \sqsubseteq^n_{\mathcal{S}})$
$\sqsubseteq$	33	partial order on functions $\mathit{State}' \rightarrow \mathcal{R}$
$\sqsubseteq_{\mathcal{P}}$	33	partial order on probability functions
$\sqsubseteq_{\mathcal{S}}$	34	partial order on functions of the semantic domain
$\sqsubseteq^n_{\mathcal{S}}$	34	partial order on an n-tuple of functions of the semantic domain
$[\![\cdot]\!]$	40	semantic function (defines the meaning of an expression)
$[\![\cdot]\!]_{.}$	35	conditional semantic function
$[\![E]\!]_f$	35	meaning of E conditional upon f
α	30	typical element of *NDObj*
β	30	typical element of *DObj*
$\mathcal{B}$	40	functional used to define the meaning of method bodies
$\mathbb{B}$	30	set of boolean objects
C	26	typical element of *CName*
$\mathcal{C}_{\beta,\sigma,\tau}$	42	function for calculating the set of β's (indirect) references
CD	26,57,81	typical element of *ClassDef*
ClassDef	26,57,81	syntactic category of class definitions
CName	26	syntactic category of class names
$\mathcal{D}$	40	function that defines the behaviour of primitive methods
DC	42	function for calculating the set of (indirect) references
δ	30	typical element of Δ
Δ	30	set of executing objects (process or data)
DObj	30	set of all data objects
E	27	typical element of *Exp*
Exp	27	syntactic category of data expressions
γ	30	typical element of *PDObj*
$\underline{\gamma}$	27	textual representation (direct naming) of γ
$\mathcal{I}$	32	maps nonprimitive method bodies to unique indices
$\mathbb{I}$	30	set of integer objects
IVar	26	syntactic category of instance variables
λ	31	typical element of Λ
Λ	31	set of local variables states
LVar	26	syntactic category of local variables
m	26	typical element of *MName*
$\hat{}\,m$	32,65	super method call (calls m in superclass)
$\mathcal{M}$	32,65	dynamic method lookup function
$\mathcal{M}_s$	31,65	static method lookup function
m_C	27	method m of data class C
$\mathrm{MaxId}(\sigma)$	31	returns the largest object identifier in σ
MD	26	typical element of *MethDef*
MethDef	26	syntactic category of data method definitions
MName	26	syntactic category of method names
$\widehat{n}$	30	identifier of a nonprimitive data object
NDObj	30	set of nonprimitive data objects
p	33	typical element of $\mathcal{P}$
$\mathcal{P}$	33	set of substochastic probability functions
PDObj	30	set of primitive data objects
ϕ	30	typical element of Φ
Φ	30	set of functions that assign values to instance variables
proc	30	executing process object
ψ	30	typical element of Ψ
Ψ	30	set of functions that assign values to local variables
$\mathbb{R}$	30	set of real objects
$\mathrm{Relabel}_{+k}$	42	relabelling function that increases object identifiers by k

ϱ	40	limits the range of random numbers to $\{1, \ldots, \varrho\}$
s	31	typical element of *State*
$\mathcal{S}$	34	semantic domain
$\mathbb{S}$	30	set of string objects
s'	31	typical element of *State*$'$
σ	30	typical element of Σ
Σ	30	set of global variables states
State	31	set of execution states
State$'$	31	set of pairs of execution state and object resulting from an expression evaluation
τ	31	typical element of T
T	31	set of functions that give the types of data objects
u	26	typical element of *LVar*
v	26	typical element of *Var*
$\mathcal{V}$	31,64	calculates the set of (inherited) instance variables of a class
Var	26	syntactic category of variables
w	26	typical element of *LVar*
x	26	typical element of *IVar*
$\mathcal{X}$	36	functional used to define the meaning of `while`
$\dot{\mathcal{X}}_f$	36	functional used to define the meaning of `while`
y	26	typical element of *IVar*
z	26	typical element of *LVar*

Chapter 4 POOSL Process Layer

Symbol	Page	Meaning
$c \xrightarrow{a} \boldsymbol{\pi}$	62	action transition in c producing probability function $\boldsymbol{\pi}$
$c \xrightarrow{t} c'$	62	time transition in c producing configuration c'
$[S]^{\psi}_{C}$	61	S executed in class C with local variables state ψ
$\left([S]^{\psi}_{C}, \sigma, \tau\right)$	82,61	configuration representing statement S running in class C in variables contexts ψ (local) and σ (global) with type information stored in τ
$\surd(S)$	63	terminated function applied to statement S
$\surd$	61	terminated statement (pronounced “tick”)
a	62	typical element of *Act*
Act	62	set of actions
c	61	typical element of *Conf*
CD	26,57,81	typical element of *ClassDef*
Conf	61,85	set of configurations
f	62	fix action
ℓ	62	typical element of $\mathcal{L}$
$\overline{\ell}$	62	complement of message ℓ
$\mathcal{L}$	62	set of communication actions
${}^{\uparrow}\mathcal{M}$	65	super method lookup function
MD^p	58	typical element of *MethDef*^{p}
MsgSignatures	58	syntactic category of message signatures
MethDef^p	58	syntactic category of process method definitions
ms	58	typical element of *MsgSignatures*
$\mathcal{P}(Conf)$	62	set of substochastic probability functions over configurations
$\boldsymbol{\pi}$	62	typical element of $\mathcal{P}(Conf)$
S	60	typical element of *Stat*
S^b	58	typical element of *Stat*^{b}
$\sigma\!\upharpoonright_t$	69,85	increases *currentTime* in global variables state σ by t
Stat	60	syntactic category of (extended) statements

$Stat^b$	58	syntactic category of basic statements
t	62	typical element of *Time*
$\widetilde{t}$	60	evaluated and fixed value t (used for time and port expressions)
τ	62	internal action (also called silent action)
$Time$	62	time domain
$Timeless(E, s)$	70	determines if E does not depend on *currentTime* in state s
$Urgent(BSpec)$	69	action urgency predicate
$Urgent(BSpec, t)$	69	states if an urgent action happens within t units of time
$\mathbb{V}$	66	function for cleaning up redundant local variables contexts

Chapter 5 POOSL Architecture Layer

Symbol	Page	Meaning
$\|\|$	82	parallel composition operator
$\upharpoonleft_t$	85	increases the model time by t
$BSpec$	82	typical element of *BSpecifications*
$BSpec^b$	82	typical element of $BSpecifications^b$
$BSpec\,[f]$	82	relabelling of ports listed in relabelling set f
$BSpec \setminus L$	82	hiding of ports listed in hiding set L
$BSpecifications$	82	syntactic category of extended behaviour specifications
$BSpecifications^b$	82	syntactic category of basic behaviour specifications
CD	81	typical element of *CDList*
$CDList$	81	syntactic category of class definitions
MSS	84	calculates the message signature sort
$\underline{p}$	82	direct naming (textual representation) of p
PE	83	typical element of *PExp*
$PExp$	83	syntactic category of parametric expressions
$Port$	83	port-identifier extractor function
PS	84	calculates the port sort
ς	85	typical element of *SyntSubst*
$SyntSubst$	85	set of syntactic substitution functions
$SSpec$	81	typical element of *SSpecifications*
$SSpecifications$	81	syntactic category of system specifications

Chapter 7 Prototyping

Symbol	Page	Meaning
$\lightning$	120	external event
$\|\|\|$	120	interleaving operator
$\zeta \xrightarrow{\lightning} \zeta'$	120	external event transition
c^e	120	typical element of $Conf^e$
$Conf^e$	120	set of extended configurations
ζ	120	typical element of Z
$EVar$	120	set of external variables
$ExtEvent$	120	set of external events
χ	120	typical element of *EVar*
$IncorporateExtData$	121	function to incorporate external data into a model
mt	117	instant in model time
$\mathcal{P}(Conf^e)$	121	set of extended substochastic probability functions
$\boldsymbol{\pi}^e$	121	typical element of $\mathcal{P}(Conf^e)$
rt	116	instant in real time
$Urgent(\zeta)$	121	external-event urgency predicate
$Urgent(\zeta, t)$	121	states if an urgent external event happens within t units of time
Z	120	set of external processes

Index

Curriculum Vitae

Leonardus Jakobus van Bokhoven was born on January 3, 1975, in Eindhoven, the Netherlands. From birth to finishing elementary school he lived in Nuenen. After moving to Waalre, Leo continued his education at the Hertog Jan College in Valkenswaard. Here, he discovered his interest in physics, chemistry and mathematics, so he enroled in Information Technology at the Technische Universiteit Eindhoven after graduating from VWO in 1993. In 1998, he obtained his MSc degree. At that time he had started as a PhD student at the Department of Electrical Engineering working on synthesis for system-level design methods — resulting in this dissertation.

During a practical training for his studies at the university, Leo has worked in the field of synthesis for hybrid controller architectures at Philips Research (NatLab.). Before that project, he had examined the feasibility of lossy image compression by subband coding with power-symmetric FIR-filters at the university. He carried out his final project at Océ in Venlo, working on a full image-processing path for colour copiers, run on a VLIW (very large instruction word) processor. The image-processing algorithms were successfully implemented in hand-optimised assembly, including colour-space conversions, filters, several image-enhancing algorithms, unsharp masking and error-dithering with subpixel positioning.

As of March 2002, Leo is working at the Research and Development department of Magma Design Automation on RTL synthesis.

Embedded, distributed, real-time, electronic systems are becoming more and more dominant in our lives. Hidden in cars, televisions, mp3-players, mobile phones and other appliances, these hardware / software systems influence our daily activities. Their design can be a huge effort and has to be carried out by engineers in a limited amount of time. Computer-aided modelling and design automation shorten the design cycle of these systems enabling companies to deliver their products sooner than their competitors.

The design process is divided into different levels of abstraction, starting with a vague product idea (abstract) and ending up with a concrete description ready for implementation. Recently, research has started to focus on the system level, being a promising new area at which the product design could start.

This dissertation develops a constructive approach to building tools for system-level design / description / modelling / specification languages, and shows the applicability of this method to the system-level language POOSL (Parallel Object-Oriented Specification Language). The formal semantics of this language is redefined and partly redeveloped, adding probabilistic features, real-time, inheritance, concurrency within processes, dynamic ports and atomic (indivisible) expressions, making the language suitable for performance analysis/modelling. The semantics is two-layered, using a probabilistic denotational semantics for stating the meaning of POOSL's data layer, and using a probabilistic structural operational semantics for the process layer and architecture layer.

The constructive approach has yielded the system-level simulation tool *rotalumis*, capable of executing large industrial designs, which has been demonstrated by two successful case studies — an ATM-packet switch (in conjunction with IBM Research at Zürich) and a packet routing switch for the Internet (in association with Alcatel / Bell at Antwerp). The more generally applicable optimisations of the execution engine (rotalumis) and the decisions taken in its design are discussed in full detail.

Prototyping, where the system-level model functions as a part of the prototype implementation of the designed product, is supported by *rotalumis-rt*, a real-time variant of the execution engine. The viability of prototyping is shown by a case study of a learning infrared remote control, partially realised in hardware and completed with a system-level model.

ISBN 90-386-1960-X