Specification and analysis of Internet applications

Citation for published version (APA):
Beek, van, H. M. A. (2005). *Specification and analysis of Internet applications*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR589889

DOI:
10.6100/IR589889

Document status and date:
Published: 01/01/2005

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Specification and Analysis of Internet Applications

Harm van Beek

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Beek, Henricus M.A. van

Specification and Analysis of Internet Applications /
by Henricus M.A. van Beek. -
Eindhoven : Technische Universiteit Eindhoven, 2005.
Proefschrift. - ISBN 90-386-0564-1
NUR 993
Subject Headings : computer networks ; protocols / Internet / semantics /
process algebra / programming languages /
software ; specifications / software verification
CR Subject Classification (1998) : F.4.3, D.3.1, F.3.2, D.2.4

Printed by University Press Facilities, Technische Universiteit Eindhoven

Cover design by ISAAC Web Solutions

Specification and Analysis of Internet Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 9 juni 2005 om 14.00 uur

door

Henricus Martinus Adrianus van Beek

geboren te Westerhoven

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. J.C.M. Baeten

Copromotor:
dr. S. Mauw

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2005-05

Preface

Until the end of my graduation I had never considered doing a Ph.D. However, when prof.dr. Jos Baeten and dr. Sjouke Mauw offered me this chance I did not have to think about it for too long. There was only one issue, namely that I was also working for my company, which was hard to combine with a full-time Ph.D. position. The opportunity to have a part-time position, three days a week for five years, was a perfect combination with my work for ISAAC.

Right now, you are reading the result of this five-year period, which would never have been accomplished without the help of many people, all of whom I would like to thank.

First of all, I would like to thank my supervisors Jos and Sjouke, without whom I would not even have thought of writing a Ph.D. thesis, let alone finishing it.

Furthermore, I would like to thank my colleagues at the formal methods group, Louis in particular, with whom I had many discussions and, not to forget, played many games of table-tennis which gave us time to relax. I thank the (Eindhoven) Embedded Systems Institute for offering me a very nice place of work and my colleagues over there for the nice working atmosphere. Furthermore, I thank Tim and Cas for their valuable comments on previous versions of this thesis.

Of course I thank prof.dr.ir. Loe Feijs, dr. Rob van Glabbeek and prof.dr. Wim Hesselink for taking part in the core committee and ms.prof.dr. Lynda Hardman and prof.dr. Martin Rem for joining the doctorate committee.

Apart from them, special thanks go to my colleagues at ISAAC, especially Mark and Max, for giving me the time to work on this thesis and for having the patience in busy times when I had to finish my thesis. Fortunately, their planning made the combination with my Ph.D. position possible. Of course, I also thank Max and Mark for being my paranimphs.

Finally, I would like to thank all my friends and relatives for supporting me during my Ph.D. period and for giving me enough distraction outside working hours.

Harm van Beek
Eindhoven, April 2005

Contents

Preface **v**

Contents **vii**

1 Introduction **1**
1.1 Trends on the Internet 1
1.2 Programming Internet Applications 3
1.3 Using Formal Methods 3
1.4 Goal of this Thesis 4
1.5 Outline 5

I Domain Analysis **7**

2 Introduction to Domain Analysis **9**
2.1 Introduction 9
2.2 Language-Driven System Design 10
2.2.1 Identification of the Problem Domain 11
2.2.2 Identification of the Problem Space 11
2.2.3 Formulation of the Language Definition 12

3 Domain Identification **13**
3.1 Internet Applications for Distributed Consensus 13
3.2 The Hypertext Transfer Protocol 14
3.3 Web-Based versus Window-Based Applications 15
3.4 Example Applications 18
3.4.1 Sinterklaaslootjes 18
3.4.2 Meeting Scheduler 20
3.4.3 Internet Vote 22
3.4.4 Internet Auction 24
3.5 Security Considerations 26

3.6 Access Control . . . 30
3.6.1 Role Based Access Control . . . 30
3.6.2 Task Based Authorisation Control . . . 32

4 Concepts **33**
4.1 Introduction . . . 33
4.2 Processes . . . 35
4.2.1 Transactions . . . 36
4.3 User Interaction . . . 38
4.3.1 Level of Abstraction . . . 38
4.3.2 The Interaction Primitives . . . 40
4.4 Users . . . 42
4.4.1 Client Identification . . . 42
4.4.2 Access Control . . . 44
4.5 Types and Calculations . . . 45
4.6 Time . . . 45
4.7 Overview . . . 46

II Modelling Internet Applications **47**

5 Process Algebra **49**
5.1 Alphabet . . . 49
5.2 Operators . . . 50
5.3 Axioms . . . 50
5.4 Deadlock . . . 52
5.5 The Empty Process . . . 52
5.6 Semantics . . . 53
5.7 Additional Operators . . . 54
5.7.1 Conditional Branching . . . 54
5.7.2 Conditional Repetition . . . 54
5.7.3 Conditional Disrupt . . . 56

6 Modelling States and Time **57**
6.1 Introduction . . . 57
6.2 States . . . 58
6.3 Scope Operator . . . 60
6.3.1 Semantics . . . 62
6.4 Adding a Time Component . . . 63
6.4.1 Semantics . . . 63

7 Modelling Internet Communication **67**
7.1 Alphabet . . . 67
7.2 Semantics . . . 70
7.3 Access Control . . . 72
7.3.1 Parallel Composition . . . 73
7.3.2 Anonymous Interaction . . . 74
7.3.3 Identification . . . 75
7.3.4 Registration . . . 76

8 Modelling Transactional Behaviour **79**
8.1 Introduction to Transactions . . . 80
8.2 A Process-Algebraic Approach . . . 83
8.2.1 Transactional Operator . . . 83
8.2.2 Locking and Unlocking Operators . . . 88
8.2.3 Merge Operator . . . 89
8.2.4 Overview of PAtrans . . . 91
8.3 Examples . . . 92
8.4 Properties of Process Algebra with Transactions . . . 99
8.5 Combining Transactions and States . . . 102
8.5.1 Introduction . . . 102
8.5.2 Rollbacks and Commits . . . 104
8.5.3 Executing Actions from within a Transaction . . . 105
8.5.4 Locking and Unlocking . . . 107
8.5.5 Identifiers and Scoping . . . 110
8.6 Degrees of Isolation . . . 111
8.7 Related Work . . . 113

9 Modelling Internet Applications **115**
9.1 Types . . . 116
9.2 Alphabet . . . 117
9.3 Operators . . . 118
9.4 Operational semantics . . . 121
9.4.1 Transition Labels . . . 121
9.4.2 Deduction Rules . . . 123
9.5 Example . . . 136
9.5.1 Properties . . . 138

III Tools and Applications 147

10 Conformance Testing of Internet Applications 149
10.1 Introduction . . . 149
10.2 Testing of Internet Applications . . . 150
10.3 Introduction to Conformance Testing . . . 152
10.4 Formal Model . . . 153
10.4.1 Labelled Transition Systems . . . 154
10.4.2 Request/Response Transition Systems . . . 156
10.4.3 Multi Request/Response Transition Systems . . . 156
10.5 Relating Multi Request/Response Transition Systems . . . 157
10.6 Test Derivation . . . 158
10.6.1 Adapting the Algorithm . . . 162
10.7 Example . . . 164
10.8 Using *DiCons* Specifications . . . 169
10.8.1 From *DiCons* Specifications to LTSs . . . 169
10.8.2 From *DiCons* LTSs to MRRTSs . . . 170
10.8.3 Example . . . 172
10.9 Related Work . . . 177
10.10 Conclusions . . . 179

11 Generation of Internet Applications 181
11.1 Programming Internet Applications . . . 181
11.1.1 Hypertext Documents . . . 182
11.1.2 Adding Input Parameters to Hypertexts . . . 184
11.1.3 Adding Output Parameters to Hypertexts . . . 189
11.1.4 Communicating Hypertext Documents . . . 191
11.2 Generating Executable Code . . . 195
11.2.1 Alphabet . . . 196
11.2.2 Operators . . . 197
11.3 Implementing the Compiler . . . 200
11.4 Future Work . . . 200
11.4.1 Using XML . . . 201
11.4.2 Adding Scope Control . . . 207
11.4.3 Adding Access Control . . . 207
11.4.4 Adding Transactional Processes . . . 208
11.5 Conclusions . . . 208

IV Conclusions **209**

12 Related Work **211**

13 Conclusions **215**

V Appendices **217**

A Overview of PAtrans **219**
A.1 Axioms of PAtrans 220
A.2 Deduction Rules for T(PAtrans) 221

B Proofs for PAtrans **223**
B.1 Soundness of PAtrans 223
B.2 Elimination of PAtrans to $\mathrm{BPA}_{\delta\varepsilon}$rec 232

C Proofs of Test Derivation Theory **235**
C.1 Proof of Lemma 10.6.5 235
C.2 Proof of Theorem 10.6.7 237
C.3 Proof of Lemma 10.6.9 239
C.4 Proof of Lemma 10.6.12 241
C.5 Proof of Theorem 10.6.13 241

Summary **243**

Samenvatting (Summary in Dutch) **245**

Curriculum Vitae **247**

Bibliography **249**

Index **261**

1

Introduction

In this thesis we develop *DiCons*, a formal language for specifying Internet applications. A *DiCons* specification can serve several goals, e.g., it can be used as input for a compiler, such that a running application can be produced from it, or it can be used by a tester to check whether a running application conforms to it.

We start this introduction by discussing some trends on Internet application development in Section 1.1. As a result of these trends some problems arise. One of these problems is the current state of programming languages available to developers of Internet applications. We have a short look at these languages and tools in Section 1.2. Next, we give a short introduction to formal methods, in which we explain what they can be used for and how we use them. This is done in Section 1.3. The use of formal methods for supporting the development of Internet applications is the basis and serves as a goal of this thesis. This goal is described in more detail in Section 1.4. Finally, we complete this chapter by giving an outline of the thesis in Section 1.5.

1.1 Trends on the Internet

Some trends concerning the growth of the Internet, and thus the number of Internet based applications, can be observed. Whereas in 2000, according to the *Internet World Stats* [Min05], 360 million people made use of the Internet, nowadays over 888 million people access the Internet on a regular basis. This shows the enormous popularity of the Internet. As a result, not only more and more companies start making use of the Internet for customer binding and for selling products, but also the applications developed for use via the Internet become more complex. Since the first to come with an interesting application sets the standard for that application area,

applications for the Internet are developed with a tremendous speed. Many new services are realised, for example applications which support shopping, auctions or voting via the Internet.

Apart from that, large portals arise next to the *old style* search engines. They provide functionality that goes beyond mere guidance through the Internet. The longer the visitor stays at the portal site and the more often he uses functionality provided by the portal, the higher the income from advertisements will be. Therefore, portals must offer interesting applications and must keep their functionality up to date. This does not only imply that portals must maintain a large set of applications, but also that they must be able to rapidly develop new services. Short *time to market* is an important asset.

A third observation is that since the Internet use is growing, of course the number of commercial transactions on the Internet is growing as well. To give an example, in the Netherlands only, nearly 4 million people spent 1.7 billion euro via the Internet in 2004 [Bla05]. This is an increase of 35% compared to the on-line sales in 2003, where 2.8 million people made a purchase via the Internet.

Nowadays, people are using the Internet as a prime source for finding information on all kinds of topics, and, even more importantly, the Internet is actively used for doing on-line calculations, shopping, reservation of flights and hotel rooms, auctions and even voting. All these on-line applications give rise to a new way of programming and a new way of software development.

Apart from the number of so-called Internet applications, the complexity of these applications increases too. This increasing complexity leads to a growing amount of errors in Internet applications, of which examples can be found at The Risks Digest [Neu05], amongst others. This increasing number of errors asks for better testing of the applications and, preferably, this testing should be automated.

Security and dependability are important factors at all levels of interaction. Apart from proper use of cryptographic encoding and decoding techniques, this also requires that the protocols by which information is exchanged are correct. A voting system, e.g., must guarantee that the winner is indeed the candidate who received most support.

We conclude that these trends lead to a need for the better understanding of Internet applications in general. More specific, a formalism for specifying these applications is desirable, such that formal reasoning about these applications is possible.

1.2 Programming Internet Applications

Currently, a mix of different languages, at different levels, with a low degree of formality is used for programming Internet applications, e.g. Perl, C# and Java. Apart from the programming language, most Internet applications make use of a language dedicated to the representation of transmitted data, like the Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS). The tools used for programming Internet applications combine these programming and representation languages into a mechanism such that both the logic and representation are combined into one language, e.g., Active Server Pages (ASP), PHP Hypertext Preprocessor (PHP) or Java Server Pages (JSP).

As a result, the specifications of such applications, which in most cases are the (annotated) source codes, are written in multiple languages and distributed over several files, possibly on several physically separated places on the Internet. This leads to implementations that are hard to maintain and hard to test. Even worse, it is almost impossible to give proofs of properties of vital parts of the applications, e.g., for the implementation of a voting protocol via the Internet. So, do the applications implement what we expect them to do?

1.3 Using Formal Methods

Aforementioned problems ask for a mechanism, i.e. a formal method, which enables us to specify the behaviour of Internet applications in an unambiguous way. So, making use of formal methods for specifying, reasoning about, testing and generating these Internet based systems is becoming more and more important.

Formal methods are a useful mechanism for giving *exact* specifications of any kind of systems. In general, a formal method consists of a specification language having a syntax and semantics. The syntax defines *how* to write down specifications. The semantics describes the meaning of *what* has been written down. The applications we focus on can be described as processes in which communication via the Internet plays a central role. So we need a method for writing down Internet communication processes. These processes are concurrent processes since several clients can interact with the application simultaneously. In [Pnu77], Pnueli introduced formal reasoning about concurrent processes using temporal logic. A graphical representation of concurrent processes using so-called Petri-nets, was introduced by Petri [Pet80]. More algebraic approaches to modelling these processes, called process algebras, were introduced by Hoare [Hoa78] and Milner [Mil80]. Process-algebraic specifications can be used for mathematical reasoning about processes. Furthermore, since process algebras are constructed from actions and operators, they can also be used for ex-

pressing how the process is implemented. As a result, in this thesis we introduce a mechanism for describing the processes using process algebra, which is nicely introduced in [BW90].

A process-algebraic specification of an Internet application can be used as a basis for e.g. a compiler, which can transform such a specification into a running application. Furthermore, such an algebraic specification can be used by a tester to test whether a running application conforms to this specification.

1.4 Goal of this Thesis

Building Internet applications is not an easy task. Given the many problems involved, like session management and multi-user interaction, it makes sense to investigate the use of formal methods. We think that formal methods can help to develop Internet applications more efficiently, and to improve the quality of applications.

In this thesis, we focus on answering the following questions: *Is it possible to give a useful formalism for the specification of Internet applications, such that formal reasoning, formal testing and generation of Internet applications is possible? Does this formalism lead to a better understanding of Internet applications?*

We answer these question by introducing a new specification language *DiCons (Distributed Consensus)* which is used for specifying Internet protocols for *distributed consensus*. The major characteristic of this class of protocols is that a number of users strive to reach a common goal (e.g. make an appointment, evaluate a paper, select a "winner"). The problem is that the users do not want to physically meet to reach their goal, nor will there be any synchronised communication between the users. A central system, i.e. an Internet application, is used to collect and distribute all relevant information.

Our goal is to develop a language in which the aforementioned protocols can easily be specified. The language must both be expressive enough and concrete. In order to be applicable to an appropriate range of problems, it must have the right expressive power. The language must be concrete enough, such that automatic generation of an executable is feasible.

The most important feature of *DiCons* is that it is geared towards the highest level of abstraction, the communication level, and that aspects of lower levels are treated in separate parts of the language.

The language *DiCons* needs a formal syntax so that we can use or build tools. Moreover, in order to avoid unclarities, ambiguities or misunderstandings, it is vital that also a formal semantics is developed. This makes it possible to establish formal prop-

erties of *DiCons* applications. For instance, in the case of Internet voting, we want to prove that each participant only gets to vote once, and that the candidate with the most votes is elected.

Since the work presented in this thesis is the result of a research project (which started as a Master's project), parts of this thesis has been presented before in [Bee00, BBM01c, BBM01a, BBM01b, Bee02, BM03].

1.5 Outline

Since we are new to the domain of Internet applications, we start by analysing the domain in Part I. Many techniques for doing a domain analysis are available. We introduce them in Chapter 2 and conclude that we make use of a language-driven system design. This approach asks for first identifying the problem domain, which is done in Chapter 3 by comparing Web-based to window-based applications, by inspecting some example applications and by examining security aspects. From this domain, a problem space is abstracted in Chapter 4.

The problem space consists of several concepts which serve as a basis for the language we develop. In Part II we formalise the concepts defined in the problem space. As our goal is to develop a process algebra, we start in Chapter 5 by giving the basis of many process-algebraic specification, viz. Basic Process Algebra. In the rest of this part, this algebra is extended with several features to model important concepts in the problem space of Internet applications for distributed consensus.

First of all, in Chapter 6, we explain how states can be specified in process algebra. Furthermore, we discuss how types and operations/calculations on them can be specified. Since all applications have an internal state, this is the first concept we formalise. Formalisation of all other concepts depends on states to some extent.

Client interaction with Internet applications does not correspond to interaction with other window-based applications. So, this asks for a formalisation which is dedicated to Internet communication. In Chapter 7 this formalisation of client interaction is given by modelling the communication primitives using process algebra.

In the domain analysis we conclude that an important mechanism when specifying Internet applications is the use of so-called transactions. In Chapter 8 we explain how transactions can be specified using process algebra.

To complete this part, we bring together all mechanisms in Chapter 9 where we give the process algebra that can be used for giving formal specifications of Internet applications for distributed consensus.

In Part III we show how the model can be put in practice. To show the usefulness

of the language developed, we explain how an algebraic specification can serve as a basis for the testing of running applications in Chapter 10. Apart from that, we describe in Chapter 11 how an algebraic specification can be compiled into a running application.

To complete this thesis, we discuss related work in Chapter 12 and draw some final conclusions in Chapter 13.

I

Domain Analysis

2

Introduction to Domain Analysis

When specifying a formal language, the exact domain of the language should be defined, in our case the domain of Internet applications for distributed consensus. That is, we need to identify the context and determine the boundaries of the domain of applications we want to specify. So, we start our research by exactly identifying this domain. In order to do this, we first need to analyse the domain, preferably by using proven techniques. Therefore, we first give a short and general introduction to domain-specific languages and domain analysis techniques in this chapter. Next, in Chapter 3 the actual domain analysis is done. We finalise this part by reducing the problem domain to a problem space in Chapter 4, as is explained in the remainder of this chapter.

In Section 2.1, we first have a look at domain analysis in general, after which we focus on a language-driven approach for designing domain-specific languages in Section 2.2.

2.1 Introduction

Many researchers have been working in the domain of so-called *domain-specific languages* (DSL) and *domain analysis*. A nice overview of different approaches to domain-specific languages is given by Van Deursen, Klint and Visser in [DKV00]. Domain analysis concepts are discussed in detail by Arango and Prieto-Díaz in [AP91]. Although their focus is on domain analysis to support reuse-based software specifications and implementations, the concepts discussed can be applied to domain analy-

sis in general.

From [DKV00] we learn that the development of a domain-specific language involves the following steps:

1. **Analysis**
   (a) identify the problem;
   (b) gather all relevant information in the domain;
   (c) cluster the knowledge in a handful of semantic notations and operations on them;
   (d) design a DSL that concisely describes applications in the domain.
2. **Implementation**
   (a) construct a library that implements the semantic notations;
   (b) design and implement a compiler that transforms DSL programs into a running application.
3. **Use**
   (a) write DSL programs.

In the remainder of this part we discuss points 1a and 1b. In Part II of this thesis, point 1c is discussed, so we introduce the different components of the formal model. Point 1d is discussed in detail in Chapter 9 where a formal model for specifying Internet applications, *DiCons*, is given. Part III of this thesis is concerned with points 2a and 2b. In this part we show how specifications can be used (point 3a) for the testing and generation of Internet applications, respectively.

2.2 Language-Driven System Design

As stated in points 1a and 1b, we need to identify the problem and gather all relevant information. To do this, Mauw, Wiersma and Willemse introduce a language-driven approach for designing domain-specific languages in [MWW04], which we follow to a large extent. Their approach consists of three steps:

1. identification of the problem domain;
2. identification of the problem space;
3. formulation of the language definition.

In the remainder of this chapter we shortly discuss these steps and we summarise how we handle them in this thesis.

2.2.1 Identification of the Problem Domain

In [AP91], the identifying of the problem and the gathering of all relevant knowledge concerning the problem domain is called *domain identification*. Most important sources for this identification are technical literature, existing applications and more general input like customer surveys.

In Chapter 3 we identify the problem domain using several techniques. First of all, in Section 3.1 we draw a distinction between several types of Internet applications. As mentioned, we restrict ourselves to a specific subset of all Internet applications. Next, in Section 3.3, we compare this set of Internet applications to window-based applications, i.e., to applications which are simply accessed using a graphical user interface on the machine on which the application is executed. After having a clear view of the type of applications we focus on, we give some examples in Section 3.4. Of course, as we are dealing with Internet communication, security is important. Therefore, we discuss security considerations in Section 3.5. This discussion results amongst other things in a need for access control. Therefore, access control models are discussed in Section 3.6.

2.2.2 Identification of the Problem Space

The outcome of identifying the problem domain is a collection of concepts which are related to the actual problem, in our case related to the domain of Internet applications for distributed consensus. These concepts are often too general and too large for actually solving the problem. Therefore, this set of concepts, i.e. the problem domain, should be restricted. This restricted domain is called the *problem space*.

As proposed in [MWW04], an approach for getting a restriction of the problem domain is classifying the concepts into three categories, viz. irrelevant concepts, variable concepts and fixed concepts. *Irrelevant concepts* are those concepts that do not play any part in the solution to the problem. *Variable concepts* are those concepts that vary depending on instances of the problem. In that case, the variable concepts are problem parameters. These so-called *statically variable concepts* influence the actual syntax of the domain-specific language, in this case of *DiCons*. Those variable concepts that vary within an instance of the problem domain determine the behaviour of the system. Therefore, these concepts, called *dynamically variable concepts*, appear in the state space of the Internet applications specified in *DiCons*. Finally, *fixed concepts* are those concepts that are identical to all instances of the problem.

The class of all variable and fixed concepts is referred to as the *problem space*. In Chapter 4 we give the concepts and their classification.

2.2.3 Formulation of the Language Definition

Giving a formal definition of the language is done in Part II of this thesis. We first give formalisms for the different concepts as defined in the problem space in Chapter 4. As mentioned before, the variable concepts define which concepts serve as a basis for the syntax of the language and which lead to a formal definition of the state space of Internet applications. In Chapter 4 we refer to the chapters in Part II for making the connection between the concepts and the model. The actual language, *DiCons*, is given in Chapter 9.

Since we also want to put *DiCons* in practice, which is called the *pragmatics* of the language, we show in Part III of this thesis how a *DiCons* specification can serve as a basis for the testing of Internet applications (see Chapter 10) and how it can be compiled into a running application which conforms to the specification (see Chapter 11).

3

Domain Identification

Our goal is to prove properties of Internet applications, test them and generate executable code from (formal) specifications. These goals are kept in mind while identifying the domain. In this section we sketch a general picture of the domain we are focusing on. After that, we discuss the similarities and differences between Web based and window-based applications. Furthermore, we inspect some example applications. Note that this is quite an informal approach. However, the goal of the problem domain analysis is to get (and give) an idea on what we aim at specifying, as is stated in points 1a and 1b on page 10.

In the next chapter we summarise the concepts that are fundamental to the domain of Internet applications for distributed consensus, based on the domain identification done in this chapter. This results in an informal notion of the actual problem space. The formal definition of the problem space is presented in Part II.

Since we aim at giving formal specifications of communication protocols, the domain analysis takes place from the Internet applications' point of view, i.e. a technical (protocol-based) point of view focussing on client interaction with the applications to be modelled.

3.1 Internet Applications for Distributed Consensus

Nowadays, many (software) systems make use of the Internet to communicate with other systems. These systems can be roughly divided into two groups, viz. clients and servers, which do not necessarily have to be disjoint.

A client uses a server, e.g., to do calculations, gather information, buy or sell products, or communicate with other clients via chat boxes, Web forums, etcetera. A

server, on the other hand, provides applications that can be accessed by one or more clients. These applications can be roughly split into three groups, based on the type of client interaction:

- Applications that are used for information retrieval, without any interaction between any of the clients that are making use of it, e.g. an application providing daily news;
- Applications that are used for synchronous interaction between its clients, e.g. a chat box;
- Applications that are used by multiple clients to reach a (common) goal. Examples are a forum in which clients can post questions that (asynchronously) can be answered by other clients, or a voting system where clients can (asynchronously) send their votes to a central application which calculates the results.

In this thesis we focus on this last group of applications, where several clients strive to reach a (common) goal without having to communicate synchronously, nor having to meet physically. We call this group "Internet applications for distributed consensus".

Most of these applications are based on the widely used *Hypertext Transfer Protocol* (HTTP) for interacting with the clients. All well-known Web browsers communicate with Web servers (which provide Internet applications) using the HTTP protocol. More information on the HTTP protocol can be found in Chapter 7, where we give a formal model for this protocol. We continue the domain analysis discussing the HTTP protocol in short, after which we compare Web-based, in this case this means HTTP-based, applications to window-based (desktop) applications.

3.2 The Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) [FGM+99] is a protocol used for communicating on the World Wide Web. It defines the precise manner in which clients (mainly Web browsers) communicate with Web servers. The protocol is request/response oriented: The client sends a request to the server after which the server sends a response.

HTTP is a *connection-less* protocol. This means that the client opens a connection, sends a request, receives a response, and closes the connection. As a result, each request/response pair requires its own connection. Furthermore, HTTP is a *stateless* protocol, which means that it has no memory of former connections and cannot distinguish one client's request from another. Every connection is a new request from

an anonymous client. There are however some strategies for adding state to HTTP. Since requests can be extended with parameters, one can include a so-called session identifier to interconnect consecutive requests and responses. This is discussed in more detail in Section 11.1.4.

As can be concluded from its name, the Hypertext Transfer Protocol is mainly used for sending hypertext documents to requesting clients. These documents can be considered text files containing information. However, apart from normal text documents, a hypertext can contain interactive elements, like links to other texts and forms that can be filled in and submitted. These interactive elements serve as a basis for all HTTP-based Web applications. To make up hypertext documents, one can make use of several techniques, of which the Hypertext Markup Language (HTML) [RLHJ99] is the most widely used. HTML contains a mechanism for describing forms that can be filled in and submitted, i.e. the values filled in in the form are sent to the application which provides the form. In this way, communication between applications and users take place. A more detailed description of the Hypertext Markup Language can be found in Section 11.1.1. The chosen markup language is irrelevant and therefore does not influence the *DiCons* language. In Chapter 7, the *DiCons* representation of HTTP interaction is given. In Section 4.3.2 it is shown how the HTTP protocol serves as a basis for the interaction primitives.

3.3 Web-Based versus Window-Based Applications

In general, Web-based applications, or Internet applications, behave like window-based (desktop) applications, i.e., like applications which are simply accessed using a graphical user interface on the machine on which the application is executed. They both communicate via a user interface with one or more clients. However, there are some major differences.

As mentioned before, the Internet applications we focus on are based on client/server communication via the Internet. The application runs on a server which is connected to the Internet. Via this connection, clients who are also connected to the Internet can interact with the application using prescribed protocols. Clients send requests over the Internet to the server on which the application runs. The server receives the requests and returns calculated responses.

In Figures 3.1 and 3.2 a schematic overview of the communication with Internet applications and window-based applications is given. Clients interacting with window-based applications are using a (graphical) user interface which is directly connected to the application. When interacting with Internet applications, the client sends an HTTP request [FGM+99] via the Internet, i.e. via some third parties, to the server. The server receives the request which subsequently is sent to the application. After

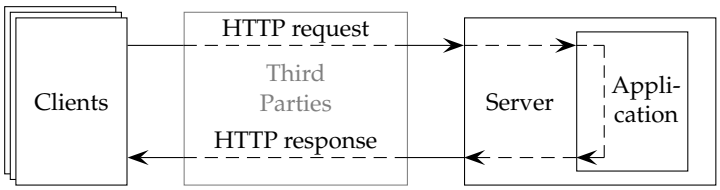


Figure 3.1: Internet interaction

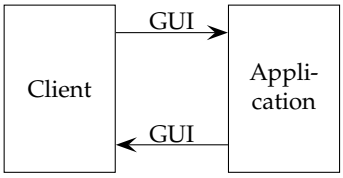


Figure 3.2: Window-based interaction

receiving the request, the application calculates a response which is sent back to the requesting client. As can be seen in Figure 3.1, when testing an Internet application we have to take into account five entities, viz. clients, communication protocols, third parties, Web servers and the application itself.

Clients The clients we focus on are so-called *thin clients*. This means that they have reduced or no possibility to do calculations. They make use of a centralised resource to operate. In the Wikipedia Encyclopedia[1], a thin client is defined as "a computer (client) in client-server architecture networks which has little or no application logic, so it has to depend primarily on the central server for processing activities".

In the context of Internet applications, thin clients are usually Web browsers. In general, more than one client can simultaneously access an Internet application. Unlike stand-alone applications, while the application stays alive, clients can fail, i.e. they can "disappear": a browser can simply be closed without notifying the application.

Dependency on third parties Since interaction takes place via the Internet, communication depends on third parties. First of all, packages transmitted go via routers which control the Internet traffic. It is not known which route on the World Wide Web is taken to get from the client to the server and back. Apart from transmitting the requests and responses, there are more dependencies, like DNS servers for translating domain names into IP addresses, trusted third parties for verifying certificates and e-mail servers for both the sending and receiving of e-mail messages.

[1]See `http://en.wikipedia.org/wiki/Thin_client`

Stand-alone applications usually do not depend on any of these parties.

Communication via the Internet Most of the communication with Internet applications we focus on is based on the Hypertext Transfer Protocol (HTTP). This protocol is request-response based. A Web server is waiting for requests from clients. As soon as a request comes in, the request is processed by an application running on the server. It produces a response which is sent back. Since the communication takes place via the Internet, delay times are unknown and communication can fail. Furthermore, messages can overtake other messages.

Web servers A Web server is a piece of hardware connected to the Internet. In contrast to stand-alone machines running a stand-alone application, a client might try to access a Web server which is down or overtaxed, causing the interaction to fail.

Internet applications The Internet application itself is running on a Web server. The applications we focus on, are based on request/response interaction with multiple clients. Since more than one client can interact with the application simultaneously, it is important to send back a response to the right client, and therefore there might be a notion of who is communicating with the application. By keeping track of the interacting parties, requests and corresponding responses can be grouped into so-called *sessions*.

Main differences between Internet-based and window-based applications are the failing of clients and Web servers, the failing of communication and overtaking of messages between clients and the application and the dependency on third parties. Furthermore, Internet applications are request/response based where window-based applications interact with the clients using a (graphical) user interface. Most Internet applications focus on parallel communication with more than one client. Since multiple clients can share a common state space, testing Internet applications is fundamentally different from testing window-based applications. Window-based applications are mostly based on single-user interaction. Finally, we mention that we discuss security aspects which rise when dealing with Internet-based applications in Section 3.5. These security aspects also differ for Internet-based and window-based applications.

More differences between Web-based and window-based systems can be found in e.g. [RFPG96].

3.4 Example Applications

In this section some relevant applications are discussed. By inspecting these applications, we define concepts that are relevant to the language we develop. *Message sequence charts (MSCs)* are used for illustrating the applications [MB01]. MSC is a graphical language for the description of both interactions between and local actions of system components. See [IT00] for more information on the MSC language. The MSCs are only used for illustrating the protocols, not for giving a complete formal specification. In [BBM01a] an extension to MSCs is presented which is dedicated to specifying Internet applications. However, we decided not to use this extension since it asks for explaining lots of notation which goes beyond the goal of this section, namely getting a feeling of how the example applications work.

The selected applications are the drawing of *Sinterklaaslootjes*, an on-line meeting scheduler, voting via the Internet and an Internet auction.

3.4.1 Sinterklaaslootjes

Near *Sinterklaas*, which takes place yearly on the sixth of December, it is a Dutch tradition that people make a surprise packet for someone else. Participants of a surprise evening come together and they all draw a ticket with the name of one other participant on it. They have to make a surprise packet for the participant they have drawn. It is not allowed to make a packet for oneself, so if someone draws the ticket with his own name on it, all tickets must be recollected and the drawing must be done over again. Apart from the fact that people have to meet for drawing tickets, often one has to backtrack on the drawing procedure. By using the Internet everyone can draw his ticket at his own desk at a suitable time. In Figure 3.3 an example, specified in MSC (actually HMSC), of a simple protocol which achieves the desired functionality is given.

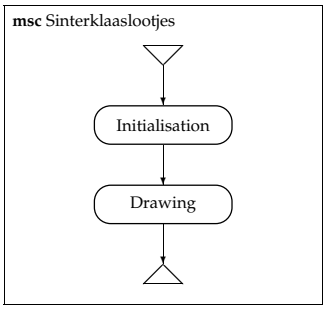


Figure 3.3: The drawing protocol

Figure 3.3 shows that first an initialisation takes place, which is followed by a drawing phase. Both processes, the initialisation and drawing, are sketched in MSCs.

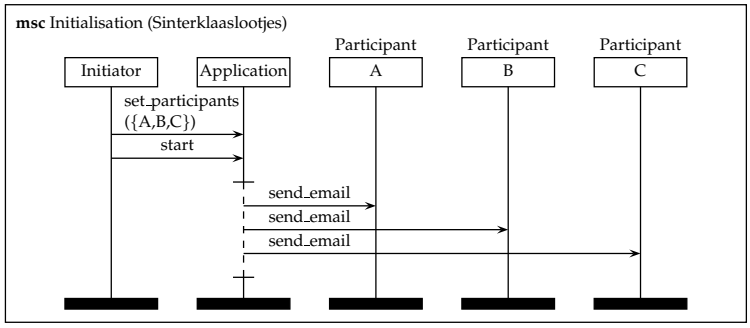


Figure 3.4: The initialisation phase

In the initialisation phase, specified in Figure 3.4, a person who is called the initiator starts the application by entering the list of participants which in this example contains three participants, A, B and C. After providing all participants, the initiator confirms that the drawing can start. The application sends an e-mail to all participants (in arbitrary order) to inform them of their participation. They are asked to visit the Internet site, i.e. the application, to draw a ticket.

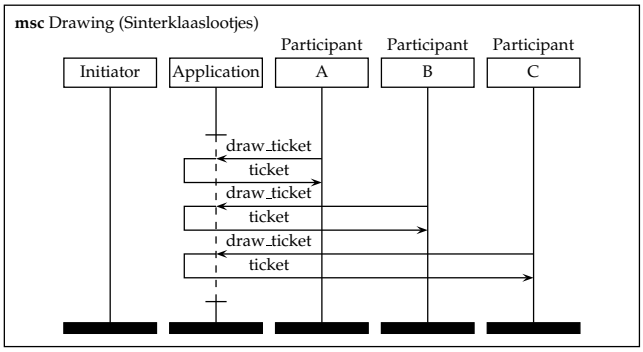


Figure 3.5: The drawing phase

Next, the drawing phase, given in Figure 3.5 starts. In arbitrary order, all participants draw a ticket containing the name of one of the other participants.

3.4.2 Meeting Scheduler

The meeting scheduler specified in this document is a simplified version of the meeting scheduler presented in [MRW01].

Scheduling a meeting often is a very time-consuming activity. As a scheduler, you have to make several phone calls and type several e-mails to collect the proper data to come to a date. Via the Internet you can automate these activities which is sketched in Figure 3.6.

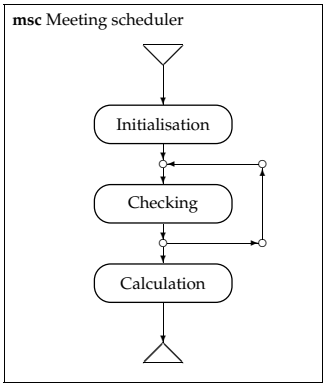


Figure 3.6: The Meeting scheduler protocol

This example consists of an initialisation phase in which an initiator provides the application with the proper data. Next, several check phases take place, in which invitees for the meeting can provide the application with their wishes. Finally, a calculation takes place in which a suitable date, if available, is selected.

An example of the initialisation phase is given in Figure 3.7. The initiator provides the application with a set of possible dates on which the meeting might take place. Apart from that, he tells the application whom to invite for the meeting. He also has to set a deadline before which all invitees have to tell the application which dates do not suit them. After having provided all this information, the initiator starts the scheduler. All invitees receive an e-mail in which they are asked to check the dates for convenience.

If an invitee is not available on one of the possible dates, he can provide the application with this inconvenience during the checking phase, as sketched in Figure 3.8. As long as the deadline is not reached, the inconvenient dates can be updated by all invitees.

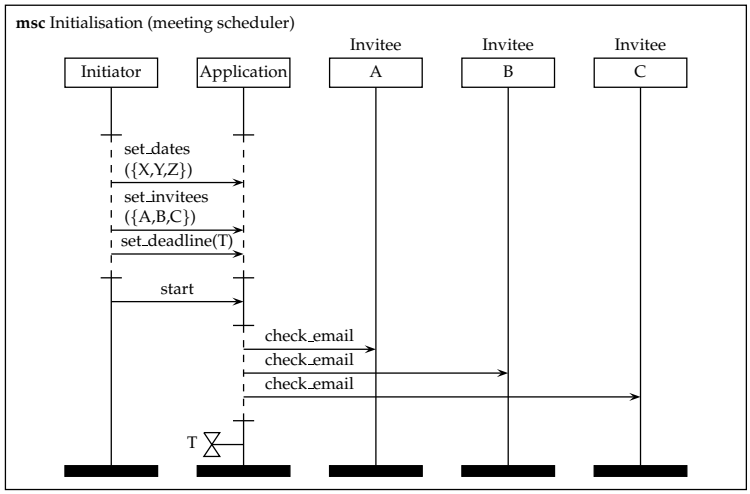


Figure 3.7: The initialisation phase

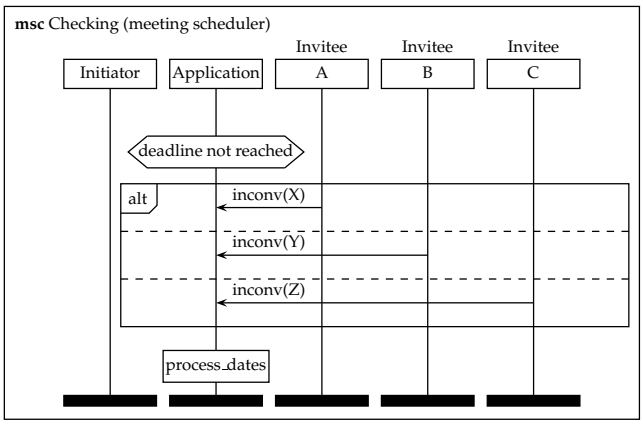


Figure 3.8: The checking phase

After the passing of the deadline, the application calculates whether a date can be found on which all invitees are available, as given in Figure 3.9. If a date is found, the initiator together with the invitees receive a confirmation, in which they are invited for the meeting and provided with the final date. If no date is available, the meeting is cancelled. The initiator and all invitees receive an e-mail containing the cancellation.

Note that the calculation phase given here does not have interactions with any of the

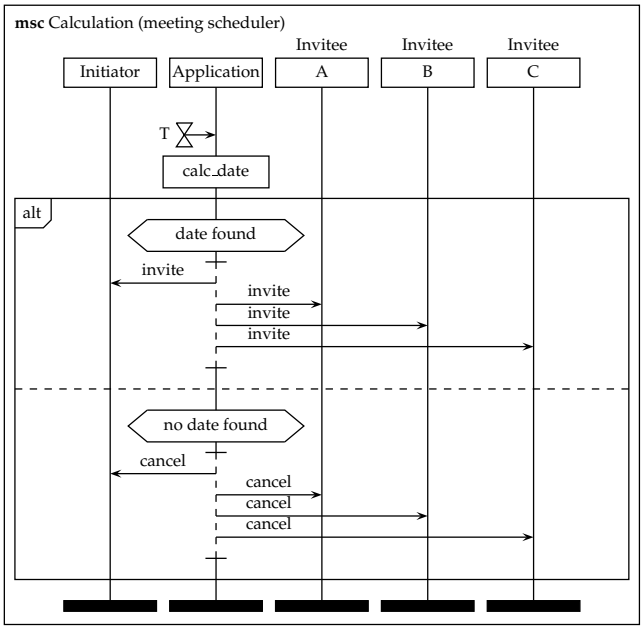


Figure 3.9: The calculation phase

users: only e-mails are sent to the initiator and invitees. Of course, it is also possible to specify that e.g. the initiator is allowed to pick a date, depending on availability of the most important invitees. After selecting this date some people are invited and others, those who cannot join the meeting, receive a message containing only the selected date. In that case the application can help the initiator, e.g., by ordering the dates on availability.

3.4.3 Internet Vote

Another example of an Internet application for distributed consensus is voting via the Web, as sketched in Figure 3.10. Similar to the meeting scheduler, the voting starts with an initialisation phase and ends with a calculation. In between these phases, one or more votes can take place.

In Figure 3.11 the initialisation is given. To start a vote, an initiator has to enter the candidates, voters and the final deadline. After providing the application with all this information, the initiator starts the actual vote. All voters receive an e-mail in which they are notified with the vote and invited to bring out their votes.

Next, all voters are allowed to vote at most once and they must do this before the

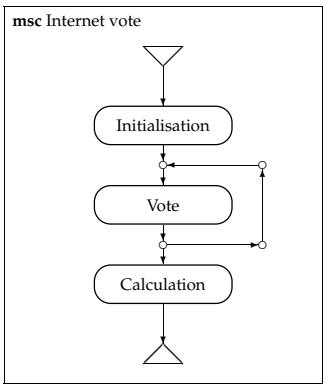


Figure 3.10: The Internet Vote protocol

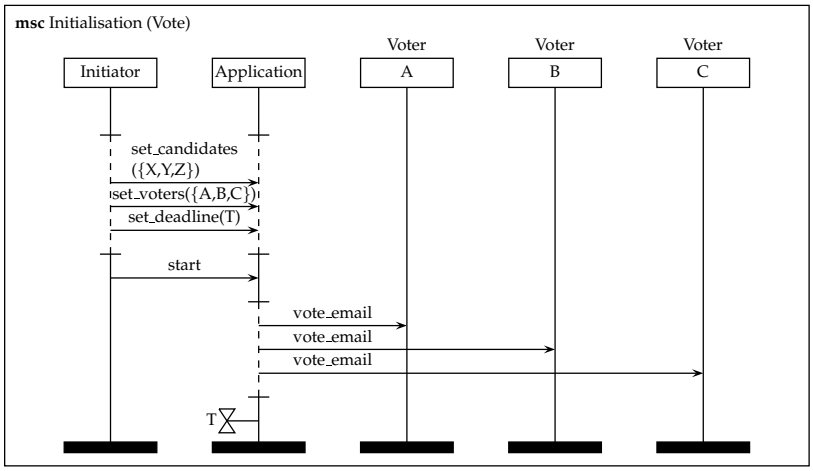


Figure 3.11: The initialisation phase

deadline, provided by the initiator, is reached. The voting process is sketched in Figure 3.12. All votes are processed by the application.

At the moment that the deadline is reached, the voters can no longer vote. The final calculation takes place and the results are sent to the initiator. Figure 3.13 describes this final phase.

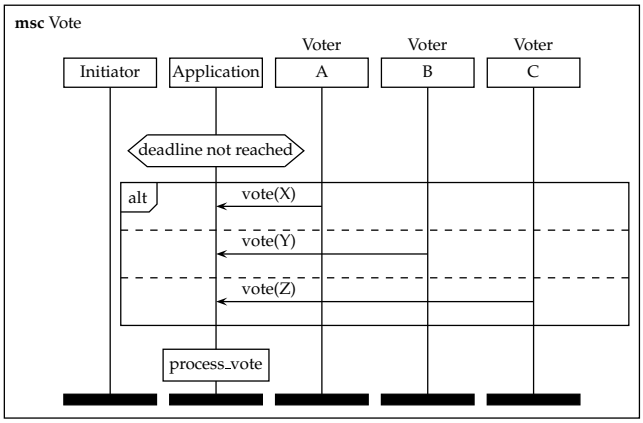


Figure 3.12: The vote phase

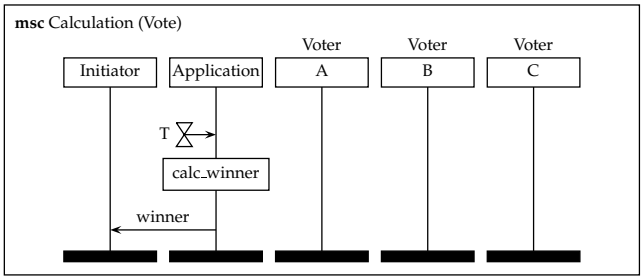


Figure 3.13: The calculation phase

3.4.4 Internet Auction

Another interesting protocol which takes part in the domain of distributed consensus applications is an Internet auction. This protocol is sketched in Figure 3.14. Similar to the example applications given before, this application starts with an initialisation phase. As in the case of the meeting scheduler and the Internet vote, the application ends with a calculation phase. In between these phases, the bidding takes place.

Anyone can start his own auction for his own products. Before someone can bid for an article, the auction has to be initialised by the initiator, which is drawn in Figure 3.15. He adds the article to the auction and sets a deadline before which the last bid has to be made. After providing the application with all relevant information, the auction is started.

People all over the world can bid for any kind of article which is added to the auction.

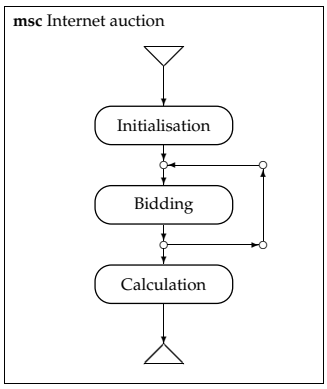


Figure 3.14: The Internet Auction protocol

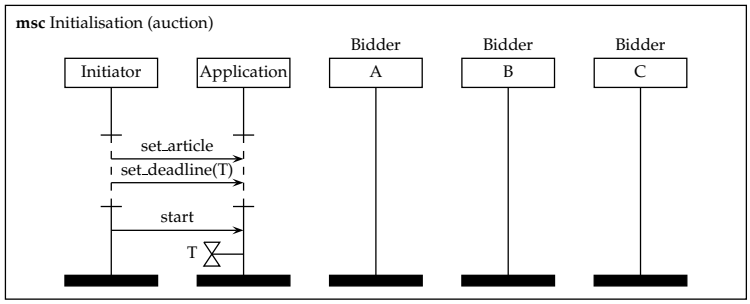


Figure 3.15: The initialisation phase

In the example given here, three bidders (A, B and C) can make a bid for the article as can be seen in Figure 3.16.

When the deadline is reached, the application calculates the highest bid and sends e-mails to the initiator and the highest bidder. This final phase is sketched in Figure 3.17).

Now that we have an idea of the type of applications we aim at specifying, we have a look at security issues that rise when implementing any of these example applications for the Internet.

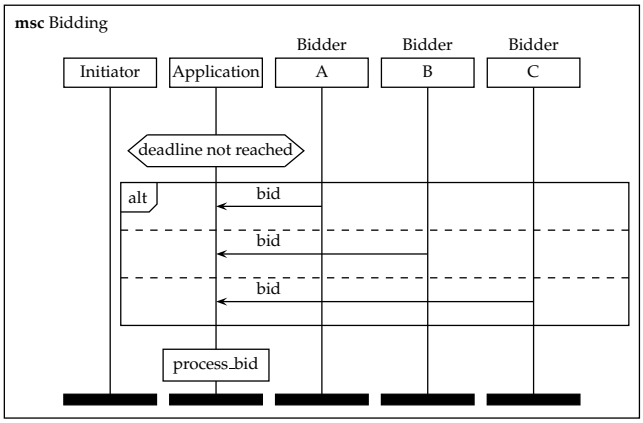


Figure 3.16: The bidding phase

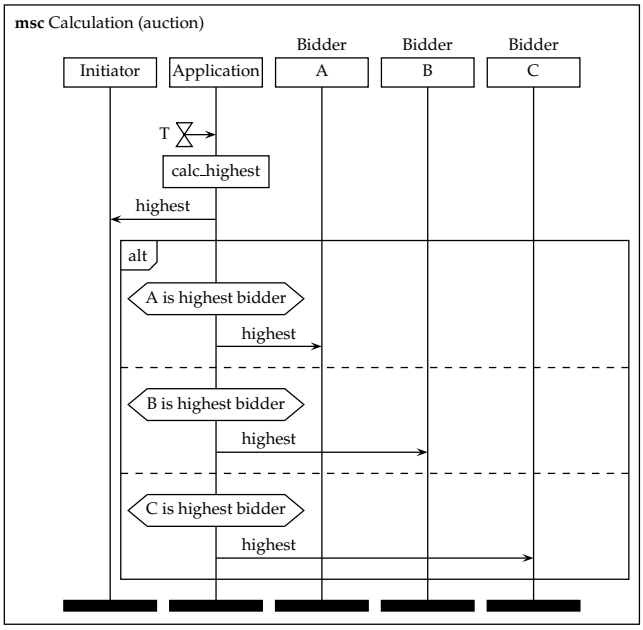


Figure 3.17: The calculation phase

3.5 Security Considerations

Using the Internet as a medium for communication has important advantages. Users can access applications at almost any time at any place in the world. Of course, being

widely available also introduces threats to the applications and its users. In this section we have a closer look at the security aspects that arise when dealing with Internet applications in general, and the applications we focus on in more detail. We do this by making use of the STRIDE model [HL02]. STRIDE is an acronym for *s*poofing identity, *t*ampering with data, *r*epudiation, *i*nformation disclosure, *d*enial of service and *e*levation of privilege. The STRIDE model sorts threats in these six categories.

Spoofing Identity Spoofing identity means that a client can pretend being someone else by using his or her user name and password to access the Internet application. Furthermore, applications can spoof identity of other applications, e.g. by showing a fake version of the other application's login screen. A client might fill in its authentication information, providing it in this way to the malicious application. Owners of this application can abuse the information provided by accessing the actual application with it.

In the problem domain, authentication of the client is an important concept. All interactions with the applications are based on who the client is, and what he is allowed to do. Authentication can be implemented in several ways, e.g. by asking the client to authenticate using personal information, like a user name and password. However, if a user shares this information with someone else, this other person can pretend being the intended user, and thus access the application without having the right identity. This careless handling of trusted users goes beyond the scope of the *DiCons* applications and is therefore not taken into account. Ways to prevent this spoofing of identities is providing trusted users with a tool to uniquely identify themselves. For example, this is done by online banking systems, providing their customers with card readers which generate unique codes. These codes have to be typed in for identification. We assume that *DiCons* applications have the ability to uniquely identify their users.

Tampering with Data Data tampering is modifying an application's data in an unintended way. The modification of data can take place in several ways. First of all, data, e.g. in a database, can be accessed directly and thus be modified directly. Apart from that, messages flowing between applications and a database, or over the Internet, between clients and an application can be altered. These modifications might cause unpredictable behaviour.

Although this is an important threat based on client hacking, we assume that the environment in which *DiCons* applications are executed is protecting the applications with respect to the tampering of data. This can for example be done by making use of Secure Socket Layers [FKK96] for communication. Furthermore, we assume that

the machine serving the application is protected in such a way that access to the application's data is only available to the application itself. So, e.g., no direct access to the applications' database should be provided. Finally, we assume that multiple instances of one application do not have a shared state space.

Repudiation If a user denies having performed an action, and other parties cannot prove this, we call this repudiation. These threats occur for example when a user buys an item in an Internet shop, or places a bid in an auction, after which he denies this. Apart from that, the execution of a prohibited operation by a user is also called repudiation if the user who executed the operation cannot be traced.

This also is an important threat, however, it mainly is a legal issue instead of a technical one. The tracing of attackers on this threat can be done by providing log files of all messages which are sent from and to the application, together with their sender and possibly some other information, like the IP address, i.e., the address where the sender actually is located on the Internet. This address can be spoofed as well, so a mechanism to protect the application with respect to address spoofing is required as well. When building a compiler for generating executable applications using formal specifications, this logging should be taken into account.

Apart from logging, important messages should be signed by customers such that it can be proved that the customer is the only person being able to send that message. This can e.g. be done by using digital signature techniques like Pretty Good Privacy [Gar95].

To really prevent repudiation attacks, complex encryption and signing techniques are required, which go beyond the scope of our research.

Information Disclosure If a user has access to information he is not allowed to see, we call this information disclosure. Examples are a user who receives personal information, like a creditcard number of another user, which he is not intended to see. Also a user's ability to read a file to which he should not have read access, or an intruding user who is interrupting messages he is not allowed to see, is called information disclosure.

As is the case with respect to data tampering, we assume that the server on which the application resides is protected in such a way that accessing the application's data in an unintended way is not possible. Also, we assume that messages are encrypted such that it is not possible that the information in an interrupted message becomes available to the malicious, interrupting, party. This can for example be done by using Secure Socket Layers [FKK96]. One of the main reasons for using formal methods for

testing and generation of Internet applications is to ensure that the information itself is only available to the intended users, so this is part of the *DiCons* specifications. By making use of a so-called access control model or authorisation control model, we prevent applications from sending information to unintended users.

Denial of Service A denial of service attack takes place when an application is not available to valid users. This can for example be caused by malicious users, overloading the systems capacity.

The denial of service attacks goes beyond the scope of *DiCons*. We assume that the environment in which the *DiCons* application resides is resistant to these attacks, or, at least, is protected as good as possible.

Elevation of Privilege Apart from information disclosure, it might be possible that a user gets privileges he is not intended to have. This is called elevation of privileges. These threats include situations in which an attacker gets privileges and becomes part in the trusted environment of the application.

Again, as we stated when discussing identity spoofing and the disclosure of information, this threat should be prevented by having a good authentication mechanism, in combination with an access control model. In this access control model, authenticated users should only get access to those parts of the applications for which they are authorised.

As a result of inspecting the applications we aim at specifying, we draw some conclusions with respect to security aspects of *DiCons* applications. First of all, we conclude that an authentication mechanism is an important concept in the problem domain. Such a mechanism makes it possible to (uniquely) identify users accessing the application. Assuming that users keep the authentication information, like names and passwords, to themselves, identity spoofing is prevented by securing messages sent from and to the application. The secured messages can be signed by a key which is maintained by a so-called trusted third party, such that "man in the middle" attacks can be prevented. We assume that the securing of messages is handled by the environment, e.g. by using Secure Socket Layers (SSL) [FKK96] for the sending of HTTP messages from and to interacting clients and by using Pretty Good Privacy (PGP) [Gar95] for encrypting e-mails. In Section 4.4.1 we have a closer look at clients in general, and how we identify them when interaction takes place.

Apart from client identification, an access control model is needed to prevent (possibly malicious) users to execute parts of the application which they are not allowed to access. Furthermore, by controlling access we can manage the data which is available to users, such that disclosure of information is prevented. In Section 4.4.2 we

have a more detailed look at access control and we introduce the model we make use of.

A logging mechanism must be available such that interactions with users can be traced. This logging, in combination with the signing of messages, makes it possible to prove that users have interacted with the application. Apart from the interaction itself, the time and (Internet) place of the interacting party should be included in the logs. This logging of interactions is left to the environment, e.g. to the service providing the Internet connection.

For the rest, we assume a secure environment which is as good as possible protected for denial of service attacks, which contains a mechanism for secure communication, and which cannot be accessed by malicious users in unintended ways. These environmental security constraints go beyond the scope of *DiCons* applications.

3.6 Access Control

In the previous section we concluded that access control plays an important role in our problem domain. Therefore, we discuss access control models in more detail. Using access control one can control permissions for accessing applications and application data in general. From [TS97, Tho97] we learn that there are two classes of access control, viz. the traditional *passive* subject-object approach and *active* security models. The main difference between the two is that in active models users' permissions can be activated and deactivated.

Examples of passive models are role-based access control (RBAC) models [SCFY96]. In these models, permissions are associated with roles of which users can be members. They can become active security models if, e.g., teams are introduced [Tho97]. On the other hand, one can look at access control from a task-oriented perspective using task-based authorisation controls (TBAC) [TS97], which are also examples of active models.

In the remainder of this section we discuss role based access control models, after which we have a look at task based authorisation control models. In Section 4.4.2 we explain which elements of the different models take part in the problem space.

3.6.1 Role Based Access Control

In role-based access control models as discussed in [SCFY96], a role is the basic building block for defining access and authorisation. RBAC models have, among other things, the following components: sets of users, roles and permissions; a many-to-many permission to role assignment relation; and a many-to-many user to role

assignment relation.

The set of users (or so-called *subjects*) contains all possible users who are allowed to access application data (*objects*) in some way. Users can have different permissions on objects. E.g., a reviewer of a paper has only read permissions where authors have both read and write permissions. Instead of assigning permissions to users, they are assigned to roles of which users can be a member. If a user is a member of a role, he or she has all permissions assigned to that role. A user can be a member of multiple roles. If so, he gets all permissions of the roles he is a member of.

A partial order (a hierarchy) on the set of roles can exist. Roles can be more powerful (*senior roles*) than other roles (*junior roles*), causing the senior role to inherit all permissions of the junior role.

Another important concept in RBAC models are constraints. Constraints can apply to all components of a RBAC model. To give an example, one can set the constraint that two roles are mutually exclusive, which means that a user cannot be a member of both roles simultaneously. In the reviewer/author example, a constraint should be that authors only have write permissions on papers they have written themselves.

In most UNIX-like systems there is a notion of *groups*. However, this is not the same as roles in a RBAC model. The main difference between roles and groups is that groups are typically treated as a collection of users and not as a collection of permissions [SCFY96]. In RBAC, a role is both a collection of users *and* a collection of permissions. According to Sandhu et al. [SCFY96] it should be approximately equally easy to determine role membership and role permissions. Apart from that, control of role membership and role permissions should be relatively centralised. In RBAC, users with a similar set of roles implicitly belong to one group. Furthermore, so called *administrative permissions* are introduced. These are permissions to modify users, roles, permissions and constraints.

Another component can be added to RBAC, viz. *duties* [Jon93]. They describe a responsible domain of a user, i.e. a set of tasks that have to be fulfilled under certain circumstances. The obeying of duties by users can be monitored, such that actions can be taken if users do not fulfil their tasks.

In RBAC, users, roles, permissions and constraints are predefined and, more or less, fixed. Only a user who has administrative permissions can change user to role and role to permission assignments. However this cannot be done automatically. Therefore, these models are called passive.

To make the RBAC model less passive, team-based access control (TMAC) is introduced [Tho97]. TMAC provides a way to apply RBAC in a collaborative environment. A *team* is defined as a collection of users with specific roles who try to accomplish a specific task. One objective for introducing TMAC is to provide just-in-time

permissions, i.e., there is a run-time permission activation mechanism at the level of individual users. This leads to automation, therefore reducing administrative overhead.

3.6.2 Task Based Authorisation Control

In task-based authorisation control (TBAC) models [TS97], sequences of operations, so-called tasks, need to be controlled. However, only a limited set of users is allowed to do specific tasks and, in most cases, within a limited time interval. TBAC enables granting and revoking of permissions to be automated and co-ordinated with the progression of the various tasks. In contrast to RBAC, this can be self-administering to a great extent. As in TBAC, in the workflow authorisation model (WAM) [AH96], authorisation is a more primitive concept. It represents that a subject has a privilege on an object for a certain time interval.

To start executing a specific task, an *authorisation step* takes place. After successfully ending this step, a set of permissions is activated. This set is dynamic and can change if, e.g., a deadline is passed.

TBAC understands the notion of "usage" associated with permissions. This means that an active permission does not imply an unlimited number of accesses with that permission. Rather, authorisations have strict usage, validity and expiration characteristics that may be tracked at run-time.

The main difference between TMAC and TBAC is that using TBAC leads to a workflow that is defined more easily. In TBAC there is no notion of roles. Users execute an authorisation step and, depending on the task's state, a set of permissions is activated. So instead of using roles, during execution of a task, different permissions are assigned to users depending on the task's progression. On the other hand, if users have the same permissions independent of the task's progression (e.g. reviewing of a paper) RBAC suits better.

In Section 4.4.2 we discuss which parts of the models introduced in this section are concepts in the problem space.

4

Concepts

In this chapter we have a more detailed look at the concepts. In Section 4.1 we introduce groups of concepts that can be determined by looking at the problem domain identified in Chapter 3. In the rest of this chapter the groups of concepts introduced are discussed in more detail. As explained in Section 2.2.2, we classify the concepts into three categories, viz. irrelevant concepts, variable concepts and fixed concepts. To recall, irrelevant concepts are those concepts that do not play any part in the solution to the problem. Statically variable concepts are those concepts that vary depending on instances of the problem and therefore influence the actual syntax of *DiCons*. The dynamically variable concepts vary within an instance of the problem domain and appear in the state space of the Internet applications specified in *DiCons*. Fixed concepts are identical to all instances of the problem.

We do not give formal definitions of the concepts here. In Part II of this thesis we formalise the concepts discussed in this chapter. However, we do refer to the chapters that handle formalisation of the concepts.

4.1 Introduction

We determine the concepts related to Internet applications for distributed consensus. We do this by looking at all elements that occur in the execution of the intended applications. In this chapter we give only informal definitions. In Part II, we formalise the concepts introduced here. In this section we introduce groups of concepts that are discussed in more detail in the remainder of this chapter.

All examples given in Section 3.4 are specifications of applications in general. One instance of the application is concerned with one execution of the protocol. So, e.g.,

one instance of the voting protocol can be the voting for a player of the year where another instances is used for voting for the chairman of a board. Instances can be identified by the data dedicated to the actual consensus problem with which the instance is concerned. The data, i.e. the variables and their valuations, is called the instance's state space.

A first group of concepts is concerned with (progression of) the Internet application itself. We call this group process-related concepts, which is discussed in Section 4.2.

All applications we focus on are based on asynchronous interaction with the users who strive to reach consensus. As can be seen by looking at the figures in Section 3.4, most interactions are initialised by the users. This is a result of the use of the HTTP protocol, as explained in Section 3.3. Furthermore, the application can actively send messages to users using e-mails for communication. So the only presentations are Web forms for input (receiving data) and output (sending data), and e-mails for output. The selected applications consist of multiple sequentially and/or concurrently executed sessions, which themselves consist of sequentially executed interactions. In the initial session, the initiator has to start the application by setting its parameters. The next sessions are the interaction sessions. In these sessions, the users can interact with the application until a specific goal is reached or until a deadline is passed. At that moment the final calculation starts. All concepts related to user interaction are discussed in Section 4.3.

As can be seen in the example applications, all specifications contain a special user who is called the initiator. This user is the user who provides the application with the data dedicated to the actual consensus problem. Examples of this data are given in Section 3.4. Note that the initiator might not be known by the application when he starts interacting with it. Therefore there must be a notion of anonymous users, who are able to identify themselves using some sort of identification mechanism. Another concept that can be determined is the group of users who try to reach consensus. In the examples given in Chapter 3, all members of this group play a similar role in the consensus protocol, e.g. voter, bidder or invitee. This group can be dynamic and initialised by the initiator, or it can be a collection of known users. So there is a need for grouping users who play a similar role. The concepts related to users are discussed in Section 4.4.

A calculation such as counting the votes or finding the highest bid can be done. Since calculations can be of any form, functions and local actions (of the applications) are completely different. For a drawing the application should prevent that a participant draws twice or that he draws himself. The application for a meeting scheduler has to process inconvenient dates and it has to check if a proper date exists. Internal actions for a voting application are checking if a voter has not voted before, processing votes and calculating the winner. An auction application has to process all bids and finally calculate the highest bid. In Section 4.5 we discuss concepts related to all kinds of

calculations.

Applications can have a deadline before which interactions with users must take place. After reaching the deadline, the possible final calculation takes place. This deadline could also be introduced in the drawing protocol. For example, it might be the day before *Sinterklaas*. As a result, there must be a notion of time. Apart from deadlines, time can be used for calculations, e.g. for calculating the average time a user reacts on an invitation. All time-related concepts are discussed in Section 4.6.

4.2 Processes

We aim at specifying Internet applications for distributed consensus. Note that we want to specify the applications, not their environment. We assume that the application is installed and running on a machine connected to the Internet. All interactionw with the application take place via the Internet, using the HTTP protocol. This means that the client sends requests to the application, on which the application calculates and returns a response. So the interaction behaviour of the application is event-driven. However, since the application actually is a sequentially executable process, we want to specify the application as a sequential process. The semantics of this process is an event-driven Web application.

Message reception from users not being the initiator only takes place in the interaction sessions. You are allowed to vote only once, for one candidate. And you are allowed to draw only one ticket. You can, however, bid several times for an article and check several times for inconvenient dates until a deadline is reached. So there must be a mechanism for determining the interactions which are allowed and those which are forbidden, i.e., there must be a mechanism to handle progression of the process. As a result, we determine the concept of *(progression of) process execution*. We classify the process execution as fixed since it is identical for all problems. A formal method for giving specifications of processes is process algebra. In a process algebra, a process is represented by an algebraic term, constructed from (atomic) actions using operators. The semantics of such a term is defined by axioms and/or operational rules. Our goal is to identify the right actions and operators to be able to easily write down correct specifications of Internet applications for distributed consensus. Execution is implicitly defined in the operational semantics of the process algebraic term that specifies the application's behaviour. In Chapter 5 we introduce the basic process algebra, which serves as a basis for the language we develop. Of course, since we develop a process algebra, actions and operators are also introduced in the rest of the formalisation, i.e., in Chapters 6 to 9.

An *instance* of an application is exactly one execution of the application, i.e., an instance of an application can be seen as a sequence of states in the state space. The

concept of instances are dynamically variable concepts which vary within a problem instance as defined in [MWW04]. A *problem instance* of the problem domain is determined by a set of characteristic concepts. So in our case, an application can be seen as a problem instance.

The progression a process makes depends on the specification together with the state the application is in. Actions can be executed depending on the state the application is in. This state is an element of the *state space*. The state space contains all possible valuations of all variables in the state of an application, together with a program counter, indicating "where the application is" at a certain moment. The state space is a dynamically variable concept, which depends on the application's specification. Where the valuation of the variables is a dynamically variable concept within a problem instance, declaration of variables are statically variable concepts. As a result, we need a (syntactical) mechanism to be able to define state space elements, i.e., we need syntax for introducing variables and their valuations. This is formalised in Chapter 6. The program counter does not need to be introduced explicitly, since progression is a fixed concept for all applications.

4.2.1 Transactions

As we have seen before, subsequent user interactions can be grouped into sessions. The applications we specify contain sessions for, e.g., voting, visiting auctions but also making payments via the Internet. Most of these sessions (or processes) are based on the concept of transactions. A transaction can be seen as a "set" of (inter)actions which occur "as a group" [EGLT76], i.e., they either all succeed or, when something goes wrong during execution, the state before starting the transaction is restored. This means that if during a transaction something goes wrong, the visitor does not have to fear that his vote is counted without him wanting to, that his correct bid for an object is ignored or that he pays an amount of money which never reaches the receiver's account. If the transaction is not finished correctly, it is completely undone. Apart from Internet applications, nearly all database systems implement transactional access: after doing table updates, one has to commit the updates to make them available to other users. As long as the updates are not committed, they can be undone (rolled back). So transactions play an important role when specifying the behaviour of interactive systems.

Gray and Reuter [GR93] define a *transaction* as a "set" of (inter)actions which occur "as a group", meaning that they either all succeed, or none of them succeeds. If all actions within a transaction are completed, the global state can be changed by an atomic and synchronised *commit* statement. During execution of this commit statement no other processes can access data updated by that statement.

To get a feeling for the use of transactions, have a look at the following example.

Suppose someone is going to the polls, e.g. to vote for a president. Then this voting process consists of several steps. First of all, the voter has to identify himself by showing his passport. If the identification succeeds, he gets a ballot containing the names of the candidates. The voter enters the polling booth, fills in the ballot and leaves the polling booth. Up till now, the voter has not voted yet. By destroying the ballot any time in the process, the voting is rolled back. However, if the voter puts his ballot in the ballot box he "commits" his vote. In this example, the commit action is an atomic action: putting the ballot in the ballot box. It might be possible that this action cannot be atomic, e.g. when doing a payment from one bank account to the other. The first account should be reduced and the second one increased. Suppose that something goes wrong while increasing the second account after the first one has been reduced. Then the entire transaction should be undone in such a way that the owner of the first account gets his money back and the second account contains exactly the amount of money as before starting the transaction. So:

> *Transactions help in coupling related actions that either should all succeed or none of them should succeed.*

Another example where transactions can be convenient is when two or more parallel processes access shared data. Suppose that two persons try to book the final seat for a flight. They both log in to the airline's Web site and fill in and submit the booking form. Then by submitting the form, they both update shared data, viz. the set of available seats. When not using transactions, the flight can get overbooked or one of the two passengers does not get registered although he receives a confirmation. So:

> *Transactions help in maintaining data integrity when parallel processes access shared data.*

As shown in the examples, the sharing of data can lead to unwanted or unexpected behaviour since updating and reading of the data can interleave. To prevent applications from having unexpected behaviour, its parallel components should meet the so-called ACID properties [Gra81, HR83]. ACID is an acronym for *atomicity*, *consistency*, *isolation* and *durability*.

Atomicity A transaction's changes to the state are atomic: either all happen or none happen.

Consistency A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation Even though transactions execute concurrently, it appears to each transaction, that other transactions take place either before or after it. So isolation

means that a program under transaction protection must behave exactly as it would do in single-user mode.

Durability Once a transaction completes successfully (commits), its changes to the state survive failure.

Processes that meet all four characteristics are called *transactions*. Transactions, and therefore atomic actions, are the basic building blocks for constructing applications. In Chapter 8 we formalise the concept of transactions.

4.3 User Interaction

As mentioned, user interaction is based on the Hypertext Transfer Protocol (HTTP) [FGM+99], which is the leading protocol for accessing the World Wide Web. Documents that are transfered via the Internet are most often written in the Hypertext Markup Language (HTML) [RLHJ99]. A short introduction to HTTP and HTML can be found in Section 3.2. In Section 4.3.1 we give an introduction on the level of abstraction we make use of when giving specifications of the interaction behaviour. The observations made in Section 4.3.1 lead to a set of interaction primitives which is introduced in Section 4.3.2.

4.3.1 Level of Abstraction

The basic problem when defining the interaction primitives is to determine the right level of abstraction. Since we are modelling communication using the HTTP protocol, it is possible to use the HTTP based interaction primitives, i.e., HTTP requests and HTTP responses as communication primitives. However, this does not correspond to the level of abstraction we would like to make use of. In order to get a feeling of the level of abstraction which is optimally suitable, consider Figure 4.1. In this drawing we sketch in MSC [IT00] a typical scenario of an Internet application which is called the *Meeting Scheduler*. This is an application which assists in scheduling a meeting by keeping track of all suitable dates and sending appropriate requests and convocations to the intended participants of the meeting, as also introduced in Section 3.4.

The example shows that we have two roles, viz. initiator and participant. In this scenario, there is only one user with role initiator, while there are three users with role participant. The MSC shows that the initiator starts the system by providing it with meeting information. Next, the system sends an invitation to the participants who reply by stating which dates suit them. After collecting this information, the system informs the initiator about the options for scheduling the meeting and awaits

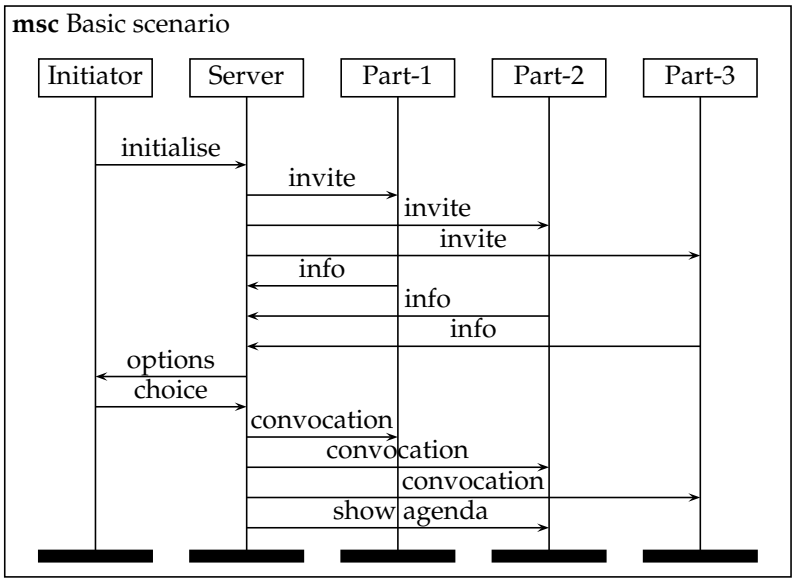


Figure 4.1: An MSC scenario of an Internet application

the choice made by the initiator. Finally, the system informs the participants about the date and offers the users to have a look at the agenda. Only participant 2 is interested in the agenda.

This example nicely shows at which level of detail one wants to specify such an application. The arrows in the diagram represent the basic interaction primitives. First, look at the *invite* messages. Since the participants do not know that they are invited for a meeting, the initiative of this interaction is at the server side. The way in which a server can actively inform a client is e.g. by sending an e-mail. This interaction only contains information transmitted from the server to the user. The messages *options* and *convocation* can also be implemented as e-mails.

The last message, *show agenda* contains information sent by the server to the user, on request of the user. This is simply the request and transmission of a non-interactive Web page.

Finally, we look at the first message, *initialise*. The initiator has to supply the system with various kinds of information, such as a list of proposed dates and a list of proposed participants. Sending information to the central application takes place in a so-called *session*. Messages *info* and *choice* can also best be implemented as sessions.

4.3.2 The Interaction Primitives

We make use of interaction primitives which are at the level of abstraction explained in the previous section. The naming of the primitives is based on the viewpoint of the application (running on the server). The primitives can be distributed over two classes, namely pushes and pulls. Reasoning from the application's point of view, a *push* sends data to an interacting users and a *pull* collects data from a user.

First of all, messages sent by the server (e.g. e-mails) are constructed using *active push* primitives. This primitive is called an active push since from the viewpoint of the server, it actively pushes data to a user. We use a left-arrow ($\leftarrow$) to textually specify this interaction primitive. Note that in the specifications, the user is placed on the left-hand side of the arrow, so the arrow is pointing to the user, indicating a push.

As explained, since we make use of standard Internet technology, i.e. e-mail and Web browsers, we are committed to use the Hypertext Transfer Protocol for most of the interactions between users and the central application. HTTP is based on a request-response way of interaction: a user has to request information from the central system, e.g. by filling in a URL in a Web browser. The central system then responds by sending a Web page or a Web form.

If the response on a user's request contains a non-interactive Web page, the interaction is a standard page request which normally takes place when surfing on the Internet. We call this kind of interactions *reactive pushes*. It is reactive since the server reacts on a user's request and it is a push since the server pushes data to the user. For this primitive we make use of the right-left-arrow ($\rightleftarrows$). This arrow shows that the user (which again is on the left-hand side) initiates the interaction after which the central system (on the right-hand side) responds.

On the other hand, if the response to the user's request contains a Web form, this form can be filled in and submitted by the user. This submission itself is another request, extended with some parameters containing the data filled in in the form. The central system has to send a response on this new request, possibly containing another Web form. In this way, we get a sequence of successive form submissions. This is called a *session*. Finally, after submission of the last form (which again is a request), the central system responds by sending a Web page *not* containing a Web form. This ends the session as no submission can take place anymore.

So a *session* is a sequence of form submissions, which is started by a user accessing the central system via a URL request and ended by the central system sending a plain Web page. In order to be able to clearly specify Internet applications from the viewpoint of the central system, we couple the sending of a form to its submission. This leads to splitting up the sequence of HTTP request/response primitives, resulting in three kinds of interaction primitives. In Figure 4.2 an overview of the HTTP

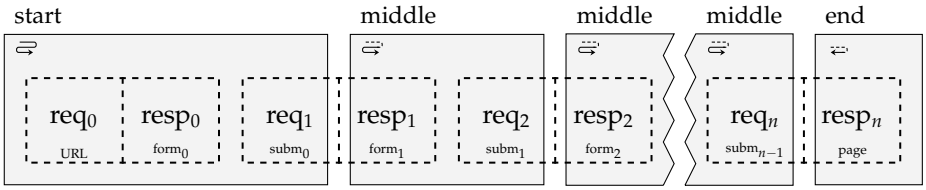


Figure 4.2: An overview of the session primitives

request/response sequence together with the interaction primitives is given.

The first interaction, a *reactive pull*, starts a session. This primitive specifies the URL request initialising the sessions, followed by the sending and submission of the first Web form. This interaction is represented using the right-left-right-arrow (⇄). The user starts the interaction, so it is called reactive. Apart from the data sent to the user there is also a flow of data from the user to the central system. Therefore we call this interaction a pull: the system pulls data from the user via a Web form.

Subsequently, zero or more mid-session interactions can take place, which specify the sending and submission of subsequent Web forms. These interactions are called *session-oriented pulls* and are represented by a session-left-right arrow (⇄). The dashed part represents the preceding request (i.e. the preceding form submission). Again, this is called a pull since there is a flow of data from the user to the central system. We call this interaction session-oriented because it is a session-based reaction on the submission of an earlier sent Web form.

Finally, for the ending of a session we make use of the *session-oriented push* primitive which specifies the sending of the final Web page. For this primitive we use the session-left-arrow (⇠) in which the dashed part again represents the preceding form submission. Again, we call this interaction session-oriented. It is called a push since the Web page sent to the user does not contain a Web form.

Looking at the five arrows one can easily see which arrows can be combined into sessions and which cannot. An arrow having a dashed tail can be connected to an arrow's head pointing in the tail's direction. Note that all correct interactions and sessions of interactions start with a solid tail and end with an arrow pointing to the left (i.e. to the user).

In Chapter 7 we formalise sessions and interaction primitives.

4.4 Users

When looking at the users interacting with Internet applications, we come to two important concepts, viz. client identification and access control. These two general concepts are discussed in this section.

4.4.1 Client Identification

Plenty of mechanisms and tools are available for clients to identify themselves. The main property of all mechanisms and tools is the communication of data which serves for this identification. This data has to be unique information with respect to clients, e.g., a string of characters or a card number and accompanying personal identification number (PIN). If the client is a human, this information can also consist of e.g. a (digital copy of a) fingerprint or iris scan, but also a smart-card serving as a digital passport or simply a password that is only known to the person to be identified. Note that the information used for identification of clients should be available to the client and verifiable by the application that tries to identify the client.

Apart from identification of known clients, many Internet applications can be accessed by anonymous users. To give an example, it is almost always possible to look at bids on products in an Internet auction without having to identify oneself. As soon as you want to place a bid, identification is required. This identification can be done by identifying as a client which is already known by the application, as described above, or by registering as a *new* client, after which identification information is sent to the requesting client. In the latter case, the client can subsequently identify himself by using the identification information sent to him.

To conclude, two possible identification methods should be available and are therefore included in the *DiCons* specification language:

- Identification of clients that are known to the application using unique identification information;
- Identification of clients that are not yet known to the application, by using a mechanism for distributing identification keys, so-called registration.

Of course, for most of the everyday Internet applications, like an auction or bulletin board, a simple but secure identification protocol should be provided. Since providing comprehensive identification protocols goes beyond the scope of this thesis, we abstract from the actual identification step.

However, to give an example of how such an identification step might look like, we discuss an identification protocol which is based on name/password combinations.

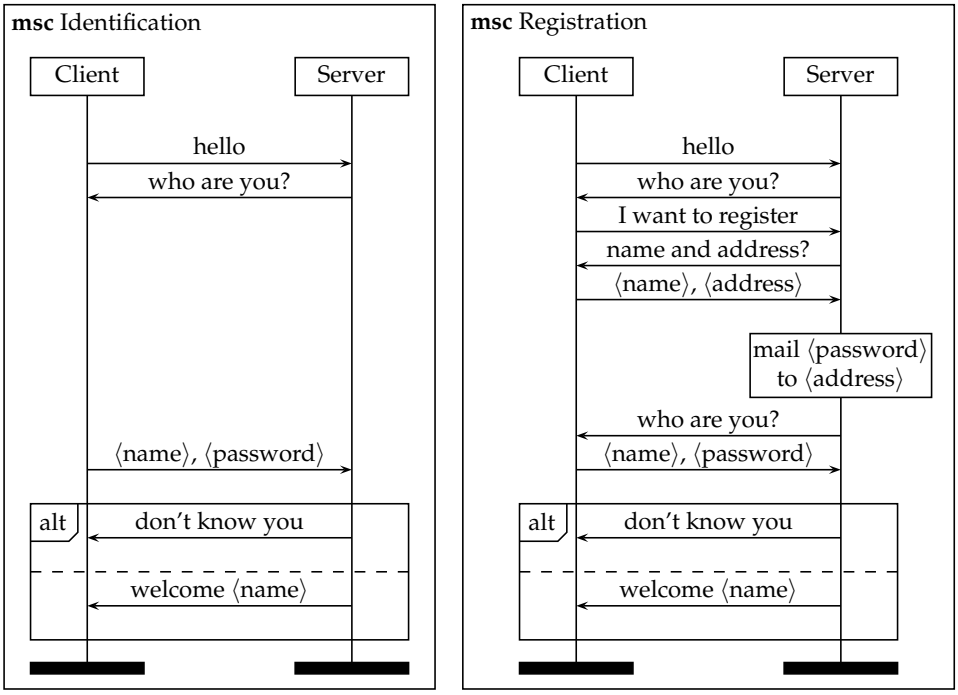


Figure 4.3: MSC scenarios for identifying clients

For identification of "anonymous clients" a mechanism is included which provides passwords by identifying the user using his e-mail address. This mechanism is called registration. By putting restrictions on these addresses we can assure that only a restricted set of persons can request for identification information. To give an example, forcing the e-mail addresses to end with `@tue.nl` causes only employees of the Technische Universiteit Eindhoven to be able to request for passwords.

In Figure 4.3, examples of the two identification protocols are given using Message Sequence Charts [IT00]. These protocols are formalised in Section 7.3. In the left-hand MSC a client known by the application identifies himself using his name and password as identification information. In the right-hand MSC, the registration protocol is depicted.

The protocol described in the left-hand MSC in Figure 4.3 is just one out of many examples of identification mechanisms. In this case, a name and password is communicated by the client, which is verified by the application running on the server. The application provides access if the client has the right permissions to access the application.

In the right-hand MSC in Figure 4.3, an example of a registration protocol is depicted. An unknown user provides the application with his name and e-mail address, after which a password is sent to him. This password can be used in combination with his name for identification as explained above.

In the applications we focus on, registration is only useful in combination with identification. As can be seen by comparing the two MSCs in Figure 4.3, registration is an extension of identification. Even stronger: for users who are known to the application, registration is replaced by identification.

Client identification is a statically variable concept with respect to *DiCons* specifications. By choosing a proper syntactical representation of clients, the identification step becomes implicit. A formalisation of client identification is given in Section 7.3.3.

4.4.2 Access Control

In Section 3.6 we gave an overview of several access control models. In this section we informally discuss the access control model we make use of in *DiCons*, based on the user interaction as specified in the Section 4.3.

In *DiCons*, we do not have a notion of roles as defined in RBAC models as discussed in Section 3.6.1. As explained earlier, roles are defined as collections of permissions. *DiCons* roles, on the other hand, can be compared with groups, i.e. collections of users as defined in [SCFY96]. Furthermore, in *DiCons*, no hierarchy on or relation between roles needs to exist. However, users can be members of multiple groups.

The *DiCons* access control model is an active model. As in TMAC, we have a notion of just-in-time permissions. Permissions change depending on the application's progression, possibly even after execution of each (inter)action in a *DiCons* specification.

In *DiCons* we have a construct to add an identification step to a session (see Section 4.4.1), which can be compared with the authorisation step available in TBAC [TS97]. In *DiCons*, authorisation depends on groups, so permissions depend on groups. In TBAC permissions depend on authorisation on a user level.

In comparison with TMAC [Tho97], we can consider execution of a *DiCons* application as a task. In that case, all users concerned with one execution can be seen as a group.

In *DiCons* there is no notion of constraints as contained in RBAC models (see Section 3.6.1). However, constraints can be explicitly included in an application's specification. These constraints can be verified using the operational semantics given in Chapter 9. To give an example, the constraint on paper's authors to only have write

permissions on papers they have written can be added to a *DiCons* specification.

Finally, we do not include administrative permissions explicitly. All users can have administrative permissions by allowing them to control group membership. In most applications only the initiator has administrative permissions. If "anonymous identification" takes place, all users implicitly have administrative permissions.

More detailed information on the *DiCons* access control model and a formalisation of it can be found in Section 7.3.

4.5 Types and Calculations

Our goal is to specify Internet applications, focussing on the communication with clients. Many calculations of different types can take place in between these communications, from determining the winner of a voting to complex mathematical calculations. Apart from a mechanism for specifying the calculations, this also asks for many types that should be available. This extensive set of types and operators on them goes beyond the scope of this thesis. We do not add types and operators for such calculations to *DiCons*. We assume the availability of a large set of types and operators, which we are free to use. Of course, if they become too specific with respect to the application specified, they should be explained and non-trivial postconditions should be given. In Chapter 6 more information in how we model types and calculations is given.

4.6 Time

As concluded in the introduction of this chapter, there must be a notion of time. On the one hand, since we have to deal with deadlines, on the other hand, since calculations with time and dates are useful. An example of a deadline is the time before which bidder should bid on a product, or voters should vote for a candidate. Calculations with time can be of use when, e.g., a date in the future should be provided by a user. This can be the case when being asked for possible dates for a meeting to be scheduled. Calculations, like comparing two time stamps, should be possible.

The way in which progression in time, which is a dynamically variable concept, is modelled is discussed in Section 6.4. Expression of deadlines, a statically variable concept, is discussed in Section 5.7.3, where the conditional disrupt operator is introduced.

4.7 Overview

To complete this chapter we summarise the concepts defined in the problem space in Table 4.1.

name	type	formalisation
process execution	fixed	Chapters 5 to 9
instance of an application	dynamically variable	Chapters 5 to 9
state space	dynamically variable	Chapter 6
declaration of variables	statically variable	Chapter 6
valuation of variables	dynamically variable	Chapter 6
users	dynamically variable	Chapter 7
groups	dynamically variable	Chapter 7
client identification	statically variable	Section 7.3.3
access control	statically variable	Section 7.3
communication primitives	statically variable	Chapter 7
session	statically variable	Chapter 7
transactions	statically variable	Chapter 8
types	statically variable	Chapter 6
calculations	statically variable	Chapter 6
time	dynamically variable	Section 6.4
deadlines	statically variable	Section 5.7.3

Table 4.1: Relevant concepts in the problem space of Internet applications for distributed consensus.

II

Modelling Internet Applications

5

Process Algebra

In this chapter we give a short introduction to the formalism that serves as a basis for our model, viz. Basic Process Algebra (BPA). A process algebra is a means of specifying algorithms and protocols, or processes, in a formal, unambiguous, way. The formalism of process algebra is introduced by Bergstra and Klop [BK82, BK84b]. A comprehensive overview of process algebra is given by Baeten and Weijland in [BW90]. In this thesis we make use of the so-called ACP-style for giving process algebraic specifications [BK84b, BW90]. As mentioned in the introduction, other process algebras exist, like the Calculus of Communicating Systems (CCS) [Mil80], Communication Sequential Processes (CSP) [Hoa78] and the π-calculus [MPW92].

Apart from BPA, we introduce the notions of unsuccessful and successful termination, together with their algebraic representations and we introduce some conditional operators which we add to our language.

5.1 Alphabet

Each process term, i.e., each algebraic representation of a process, is constructed from (atomic) actions using predefined operators. So, when defining a process algebra, there is the need for an alphabet. An alphabet, denoted by $\mathbb{A}$, is a set of symbols representing the (atomic) actions that the processes can execute. These actions serve as the basic building blocks when constructing process terms.

To give an example, when defining the well-known vending machine for getting coffee or tea, the alphabet could look like

$$\mathbb{A} = \{\text{insert-coin}, \text{get-coffee}, \text{get-tea}\} \ .$$

So the atomic actions are inserting a coin, getting coffee or getting tea. Most often, these actions are abbreviated, so instead of the alphabet given above, we would for example get

$$\mathbb{A} = \{i, c, t\} \ .$$

5.2 Operators

Having defined an alphabet, there is a need for operators to construct process terms from the actions. In Basic Process Algebra, there are two operators available, the alternative composition operator, or choice, and the sequential composition operator.

The alternative composition operator, denoted as $+$, is used for constructing a choice between processes. Given processes x and y, $x + y$ is the process that executes either x or y.

The sequential composition operator, denoted as $\cdot$, is used for defining processes in which (sub)processes should take place sequentially. Given processes x and y, $x \cdot y$ is the process that first executes x, and after completion of x continues with executing y.

By looking at the vending machine again, using the operators given above, we can define the process that first a coin should be inserted after which a choice for either coffee or tea can be made:

$$\text{insert-coin} \cdot (\text{get-coffee} + \text{get-tea})$$

As can be seen in the example, we make use of parentheses to group (sub)processes for readability. Another example is the process that first the choice should be made for coffee or tea, after which a coin is inserted and the chosen drink is produced. If, e.g., the vending machine has two slots for inserting a coin, the process can be modelled as follows:

$$(\text{insert-coin} \cdot \text{get-coffee}) + (\text{insert-coin} \cdot \text{get-tea})$$

The moment of choice makes this process different from the process given above.

5.3 Axioms

Each process algebra is based on a set of axioms, where every axiom consists of an equality between two process terms. This set of of axioms leads to an algebra. Basic Process Algebra (BPA) is axiomatised by 5 axioms, named A1–5. The axioms are given in Table 5.1.

$x + y$	$=$	$y + x$	A1
$(x + y) + z$	$=$	$x + (y + z)$	A2
$x + x$	$=$	x	A3
$(x + y) \cdot z$	$=$	$x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z$	$=$	$x \cdot (y \cdot z)$	A5

Table 5.1: Axioms of BPA.

As can be seen, the equations in Table 5.1 contain variables (x, y, z, ...) which are universally quantified. This means that the equations are assumed to be valid for every instance of the variables. An instance of a variable is an atomic action, i.e. a constant in the alphabet, or it is a process constructed using the operators and constants of the algebra.

The first three axioms concern the alternative composition operator. Axiom A1 expresses the commutativity of alternative composition: choosing between x and y is the same as choosing between y and x. Associativity of alternative composition is expressed by Axiom A2: when choosing between three alternatives, the grouping of the alternatives is irrelevant. Axiom A3 expresses the idempotency of alternative composition: a choice between identical alternatives is a choice for the process concerned. Axiom A4 concerns both the alternative and sequential composition and expresses the right distributivity of sequential composition over alternative composition: choosing between x and y, followed by z is the same as choosing between x followed by z and y followed by z. Finally, Axiom A5 expresses the associativity of the sequential composition: when first doing x, followed by y, followed by z, the grouping of the three processes is irrelevant.

Note that we did not include an axiom for left distributivity of sequential composition over alternative composition:

$$x \cdot (y + z) \neq (x \cdot y) + (x \cdot z)$$

This expresses the fact that the moment of choice in both processes is different, as explained in the vending machine example in Section 5.2. This difference is often called a difference in branching structure.

Also note that when giving Axiom A4, we assume the existence of an operator precedence scheme: sequential composition binds stronger than the alternative composition. In this thesis we adhere to the standard scheme which distributes the operators over four categories, from strongest to weakest binding:

1. all (possibly parameterised) unary operators;
2. the sequential composition operator $\cdot$;

3. all (possibly parameterised) binary operators, except $+$ and $\cdot$;
4. the alternative composition operator $+$.

Using this scheme, we can drop many parentheses. Furthermore, since associativity of both the alternative and the sequential composition operators exist (Axioms A2 and A5), we can use notations $x + y + z$ and $x \cdot y \cdot z$ as well.

By making use of the axioms of a process algebra, in this case Axioms A1–A5, processes can be rewritten into other processes using equational logic. We call two processes x and y *derivably equal* in a given process algebra P if they can be rewritten into each other using equational logic:

$$P \vdash x = y$$

5.4 Deadlock

Apart from the alphabet and operators given in the previous sections, we introduce the notion of deadlock. If a deadlock state is reached, no actions can be performed but the process does not terminate. We make use of constant symbol δ to model deadlock. BPA extended with deadlock behaviour is denoted by BPA_δ. We assume that $\delta \notin \mathbb{A}$. Apart from Axioms A1–5, as given in Table 5.1, BPA_δ consists of Axioms A6 and A7 as given in Table 5.2.

$x + \delta$	$=$	x	A6
$\delta \cdot x$	$=$	δ	A7

Table 5.2: Axioms for deadlock.

Axiom A6 states that no deadlock occurs as long as an alternative is available. Axiom A7 says that after a deadlock occurs, no actions can follow.

5.5 The Empty Process

As a counterpart of deadlock, in many cases it is useful to have a special constant to express successful termination. This process is the empty process, denoted as ε, which expresses the process that can only terminate successfully. BPA_δ extended with the empty process is denoted by $\mathrm{BPA}_{\delta\varepsilon}$. Apart from Axioms A1–7, the algebra is extended with two more axioms for the empty process, A8 and A9, given in Table 5.3. Similarly to the case for δ, we have $\varepsilon \notin \mathbb{A}$.

$x \cdot \varepsilon$	$=$	x	A8
$\varepsilon \cdot x$	$=$	x	A9

Table 5.3: Axioms for the empty process.

5.6 Semantics

To add semantics to a process algebra, we make use of a term-deduction system semantics, also called *Structural Operational Semantics*, as introduced by Plotkin [Plo81]. We use notation $x \xrightarrow{a} x'$ to denote that process x can do an a-step, or a-transition, to process x'. Apart from the transition, we introduce a termination predicate on processes, $x \downarrow$, which indicates that process x has an option to terminate successfully.

For a process algebra P, the semantics of P are given by term-deduction system $T(P)$, induced by the a set of deduction rules which capture the operational behaviour. The deduction rules for $T(\mathrm{BPA}_{\delta\varepsilon})$ are given in Table 5.4.

$$\frac{}{\varepsilon \downarrow}\,{}_{1} \qquad \frac{}{a \xrightarrow{a} \varepsilon}\,{}_{2} \qquad \frac{x \downarrow, \quad y \downarrow}{x \cdot y \downarrow}\,{}_{3} \qquad \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}\,{}_{4} \qquad \frac{x \downarrow, \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}\,{}_{5}$$

$$\frac{x \downarrow}{x + y \downarrow, \quad y + x \downarrow}\,{}_{6} \qquad \frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x', \quad y + x \xrightarrow{a} x'}\,{}_{7}$$

Table 5.4: Deduction rules for $T(\mathrm{BPA}_{\delta\varepsilon})$.

Rule 1 states that the empty process ε terminates. In rule 2 it is stated that a single action can be executed, which results in a successfully terminating process. If both processes x and y terminate successfully, than the sequential composition of x and y also terminates successfully (rule 3). Rule 4 states that if x can do an a-step to x', then $x \cdot y$ can do an a-step to $x' \cdot y$. In rule 5 it is stated that if x terminates successfully and y can do an a-step to y', that $x \cdot y$ can do this a-step to y' as well. Next, deduction rule 6 states that if x terminates successfully, then both $x + y$ and $y + x$ can terminate successfully. Finally, rule 7 states that if x can do an a-step to x', then $x + y$ and $y + x$ can do a similar a-step to x'. Note that no deduction rules involving deadlock process δ are available. When reaching a δ state, neither successful termination nor a transition can occur.

5.7 Additional Operators

We introduce three additional operators. The first two are contained in mostly all specification languages, the third one is dedicated to the type of applications we focus on. We start by introducing the conditional branching operator which specifies the if-statement. The conditional repetition operator specifies the while-statement. Finally, we introduce the conditional disrupt operator, which can be used for specifying deadlines, amongst other things.

5.7.1 Conditional Branching

We use the conditional branching operator ($\_ \triangleleft \_ \triangleright \_$), as defined in [HHJ+87]. This operator behaves like the *if-then-else-fi* operator in sequential programming:

$$x \triangleleft b \triangleright y \quad \equiv \quad \textit{if } b \textit{ then } x \textit{ else } y \textit{ fi}$$

Boolean expression b may be an arbitrary expression containing predicates on any part of the state space. This is explained in more detail in Chapter 6, where states are introduced. The axioms for conditional branching are given in Table 5.5.

$x \triangleleft b \triangleright y$	=	x	if b	CB1
$x \triangleleft b \triangleright y$	=	y	if $\neg b$	CB2

Table 5.5: Axioms for the conditional branching operator.

Axioms CB1 and CB2 are concerned with the cases that the boolean expression b evaluates to true or false, respectively. Evaluation of guards itself is explained in Chapter 6.

The SOS rules for conditional branching are given in Table 5.6. Note that also for the operational semantics, evaluation of guard b in the state is necessary.

5.7.2 Conditional Repetition

The conditional repetition can be compared with a *while-do-od* loop in traditional programming. We make use of the conditional repetition operator ($\_ \triangleright\!\triangleright \_$) to specify these repetitions:

$$b \triangleright\!\triangleright x \quad \equiv \quad \textit{while } b \textit{ do } x \textit{ od}$$

$$\frac{x\downarrow,\quad b}{x \lhd b \rhd y\downarrow} \qquad \frac{x \xrightarrow{a} x',\quad b}{x \lhd b \rhd y \xrightarrow{a} x'} \qquad \frac{y\downarrow,\quad \neg b}{x \lhd b \rhd y\downarrow} \qquad \frac{y \xrightarrow{a} y',\quad \neg b}{x \lhd b \rhd y \xrightarrow{a} y'}$$

Table 5.6: Deduction rules for the conditional branching operator.

The while loop repeats process x until test b proves false. Again, b may be an arbitrary boolean expression, which is explained in detail in Chapter 6. The axioms for conditional branching are given in Table 5.7.

$$\begin{array}{lcll} b \rhd\!\rhd x & = & x \cdot (b \rhd\!\rhd x) & \text{if } b \qquad \text{CR1} \\ b \rhd\!\rhd x & = & \varepsilon & \text{if } \neg b \qquad \text{CR2} \end{array}$$

Table 5.7: Axioms for the conditional repetition operator.

Axiom CR1 is concerned with the case that the guard holds, i.e. that expression b evaluates to true. In that case, the guarded process is started. When it finishes, the entire process is started again. Axiom CR2 specifies that when the guard evaluates to false, the process equals the empty process.

The SOS rules for conditional branching are given in Table 5.8.

$$\frac{x \xrightarrow{a} x',\quad b}{b \rhd\!\rhd x \xrightarrow{a} x' \cdot (b \rhd\!\rhd x)} \qquad \frac{\neg b}{b \rhd\!\rhd x\downarrow}$$

Table 5.8: Deduction rules for the conditional repetition operator.

Note that when b holds and x terminates, process $b \rhd\!\rhd x$ does *not* terminate. This can be compared with process

$$\textit{while}\ \text{true}\ \textit{do}\ \text{skip}\ \textit{od}$$

which does not terminate but repeatedly, possibly infinitely often, executes the skip.

5.7.3 Conditional Disrupt

We introduce the $b \rhd\!\!- x$ statement to specify conditional disrupts. This means that process x is normally executed until b becomes true. At that moment the statement terminates, independent of the (inter)actions that are taking place at that moment. If x terminates before b becomes true the statement terminates too. By placing a time check in b we can specify time-out interrupts. However, b may be an arbitrary boolean expression. The axioms for the conditional disrupt operator are given in Table 5.9.

$b \rhd\!\!- x$	$=$	ε	if b	CD1
$b \rhd\!\!- \varepsilon$	$=$	ε	if $\neg b$	CD2
$b \rhd\!\!- \delta$	$=$	δ	if $\neg b$	CD3
$b \rhd\!\!- a \cdot x$	$=$	$a \cdot (b \rhd\!\!- x)$	if $\neg b$	CD4
$b \rhd\!\!- x + y$	$=$	$(b \rhd\!\!- x) + (b \rhd\!\!- y)$	if $\neg b$	CD5

Table 5.9: Axioms for the conditional disrupt operator.

Axiom CD1 states that if a disrupt takes place, the process terminates successfully. In Axiom CD2 it is stated that if the conditionally disrupted process terminates successfully, than the process can not be disrupted. Axiom CD3 states that deadlock cannot be disrupted. Axiom CD4 states that as long as no disrupt takes place, the process can continue executing its actions. Finally, conditional disrupts distribute over the alternative composition as stated in Axiom CD5.

The SOS rules for the conditional disrupt operator are given in Table 5.10.

$$\frac{x \downarrow}{b \rhd\!\!- x \downarrow} \qquad \frac{b}{b \rhd\!\!- x \downarrow} \qquad \frac{x \xrightarrow{a} x', \quad \neg b}{b \rhd\!\!- x \xrightarrow{a} b \rhd\!\!- x'}$$

Table 5.10: Deduction rules for the conditional disrupt operator.

6

Modelling States and Time

In this chapter we focus on how to model states using process algebra. We introduce states, state spaces, and state manipulation. We give the basic concepts that are needed for modelling states. Furthermore, we introduce the so-called scope-operator, an algebraic operator for handling states and changes to the state. We give the syntax for this extension to BPA and its operational semantics.

6.1 Introduction

In Chapter 5 we gave a short introduction to Basic Process Algebra. In this algebra, we have operators for sequential and alternative composition. As our goal is to model Internet applications using a process algebra, we need some more constructs. One important construct is adding the modelling of states. During execution of an (Internet) application, the application can reach several states. In general, the state is the valuation of all declared variables together with a set of program counters, indicating the part of the application to execute next.

Depending on the state of an application, the next action to execute can be determined. This determination depends on the progression the program has made, however, it might also depend on valuations of variables. This dependence on valuations is especially the case when using for example *if-then-else-fi* (Section 5.7.1) or *while-do-od* (Section 5.7.2) constructions. When giving a process algebraic specification, the progression of the program is implicitly specified in the operational rules in contrast to the valuations of variables. Therefore, there is a need for a mechanism for declaring variables and for keeping track of the valuation of declared variables. As mentioned, the operational semantics of an application depends on this valuation, so deduction rules should be extended with the valuation of all declared variables.

This is done by extending the processes with a state space containing this valuation, as is explained in Section 6.3.1.

Furthermore, since the calling of functions and procedures is needed, declaration of local variables should also be possible. Of course, when adding an operational meaning to a specification, problems like name clashes and undeclared variable references should be taken into account.

6.2 States

Before introducing the operator for declaring variables (the so-call scope operator), we give an introduction to states. As mentioned before, the states we are interested in, consist of the valuations of variables. To tackle the problem of name clashing when nesting variable declarations, we make use of so-called *valuation stacks* for defining states, which are based on the notion of state stacks as defined in [BK02]. A valuation stack contains a pile of valuations of variables. When declaring a variable, it is put on top of the stack, not taking into account whether a variable with the same name is declared before. When assigning a value to a variable with a given name, the top most variable with that name is used for the assignment. In this way name clashes are prevented. The scope operator (which is used for declaring variables) makes use of the stack structure of states. In a later stage, in Chapter 8 on page 103, we also introduce valuation sets since we need to do calculations using state changes.

The types of variables are fixed in a state. Many types are needed when specifying real-life Internet applications. Next to well-known types like natural numbers, text strings and dates, more complex abstract data type like products, votes and sets of *Sinterklaaslootjes* can be useful. We allow the use of these abstract data types in specifications. Therefore we introduce the universe of types $\mathbb{T}$. Each type can be seen as a set of constants. Apart from the types, we define the universe of identifiers by $\mathbb{ID}$. The identifiers are used for identifying variables. Then, a valuation can be considered the coupling of a constant of a certain type to an identifier.

$\mathbb{T}$ **The universe of types**
This set contains all possible types.

$\mathbb{ID}$ **The universe of identifiers**
This set contains all possible identifiers.

Definition 6.2.1 *(Valuation) Let T be a type, $T \in \mathbb{T}$, i be an identifier identifying a variable of type T, $i \in \mathbb{ID}$, and c be a constant of type T. Then, a* valuation *v is a triple $\langle i, c, T \rangle$,*

denoted as

$$v \quad =_{\text{def}} \quad i \mapsto c : T \ .$$

If a variable is declared but no valuation is specified, i.e., no value is assigned to it yet, we use notation $i \mapsto \bot : T$.

Definition 6.2.2 *(Uninitialised valuation) Let T be a type, $T \in \mathbb{T}$ and i be an identifier identifying a variable of type T, $i \in \mathbb{ID}$. An* uninitialised valuation *v is then defined by*

$$v \quad =_{\text{def}} \quad i \mapsto \bot : T \ .$$

Uninitialised valuation $i \mapsto \bot : T$ is interpreted as the existence of a value c of type T such that $i \mapsto c : T$. However, this value c is not known to the application.

Using this definition of $\bot$, we assume that $\bot$ is an element of all types, i.e., we only consider types of which $\bot$ is an element.

Definition 6.2.3 *(Valuation stack) Let v be a valuation. A valuation stack σ is then defined by*

$$\begin{array}{rcll} \sigma & =_{\text{def}} & \lambda & \textit{the empty valuation stack} \\ & \mid & v :: \sigma & \textit{a non-empty valuation stack} \ . \end{array}$$

We abbreviate valuation stacks with a single valuation v, that is valuation stack $v :: \lambda$, to v. In the remainder of this thesis, when making use of the term *state* we mean a valuation stack representing the state. The set of all possible states is denoted by $\mathbb{S}$.

For determining the value of a variable in a state, we make use of an evaluation function, which is a (partial) function on identifiers of variables.

Definition 6.2.4 *(Evaluation) Let σ be a state, v be a valuation, i be an identifier and c be a constant. The evaluation function $\sigma(i)$ is a partial function: it is not defined if i does not occur in σ. Since i does not occur in λ, evaluation $\lambda(i)$ is not defined. Evaluation $(v :: \sigma)(i)$ is defined by the following equation:*

$$(v :: \sigma)(i) \quad = \quad \begin{cases} c & \textit{if } \exists T \ v = i \mapsto c : T \\ \sigma(i) & \textit{otherwise} \end{cases}$$

We extend the evaluation function to lists of variables: $\sigma(\vec{\imath})$.

As can be concluded, we draw an explicit distinction between variables being defined in the state and evaluating to $\bot$ and variables not being defined in the state at

all. Evaluation in the first case can result in a run-time error where errors caused by evaluation in the second case can be determined at compile time.

We also have a function for determining the type of a variable, which is a partial function on identifiers of variables as well.

Definition 6.2.5 *(Type determination) Let σ be a state, v be a valuation, i be an identifier and T be a type. The type determination function* $\text{type}(i, \sigma)$ *is a partial function. It is not defined if i does not occur in σ. Since i does not occur in λ, type determination* $\text{type}(i, \lambda)$ *is not defined. Type determination* $\text{type}(i, v :: \sigma)$ *is defined by the following equation:*

$$\text{type}(i, v :: \sigma) \quad = \quad \begin{cases} T & \textit{if } \exists c\ v = i \mapsto c : T \\ \text{type}(i, \sigma) & \textit{otherwise} \end{cases}$$

We extend the type determination function to lists of variables: $\text{type}(\vec{\imath}, \sigma)$.

Apart from determining the value and type of a variable, we want to be able to substitute valuations in a state stack. A substitution can change the mapping of a variable to a mapping of the same variable to another value. Substitution $\sigma[c/i]$ replaces the uppermost occurrence of a valuation of a variable identified by i in state stack σ (i.e. the occurrence with the smallest scope) with a new valuation where this variable is mapped to value c.

Definition 6.2.6 *(Substitution) Let σ be a state stack, v be a valuation, T be a type, i be an identifier and c be a constant of type T. The substitution of c for i is then defined by*

$$\begin{aligned} \lambda[c/i] \quad &= \quad \lambda \\ (v :: \sigma)[c/i] \quad &= \quad \begin{cases} (i \mapsto c : T) :: \sigma & \textit{if } \exists c'\ v = i \mapsto c' : T \\ v :: (\sigma[c/i]) & \textit{otherwise} \end{cases} \end{aligned}$$

Likewise, we define substitution of lists of variables, $\sigma[\vec{c}/\vec{\imath}]$, where all i's in $\vec{\imath}$ occur exactly once.

6.3 Scope Operator

In the former section we introduced a mechanism for handling states where scopes of variables might differ. In order to use this mechanism in combination with a process algebra, we introduce the *scope operator*, denoted as $[\_|\_]$. The left argument is a valuation, the right argument is the process which defines the scope of the valuation.

In the valuation, apart from a constant, we also allow the mapping of a variable i to an expression e of type T: $i \mapsto e : T$. In this case, the expression first is evaluated in the current state, and its value is used for calculations.

To give an example,

$$[\, n \mapsto 0 : \mathbb{N} \,|\, (n := n + 1) \cdot (n := n \times 2) \,]$$

specifies that we have a process algebra with an alphabet containing actions for adding and multiplying natural numbers. Using the scope operator, we can declare a natural number n. Since we also allow expressions, we could for example also specify

$$[\, m \mapsto 0 : \mathbb{N} \,|\, [\, n \mapsto 0 + m : \mathbb{N} \,|\, (n := n + 1) \cdot (n := n \times 2) \,]\,] \,.$$

Within the scope of this n, the value of n can be used for calculations and changed by actions. Note that $(n := n + 1)$ and $(n := n \times 2)$ are elements of the alphabet. Their semantics is a substitution as defined in Definition 6.2.6.

By making use of the scope operator, the state of the application can be extended with valuations of variables. This, of course, can have an effect on the actions performed within the scope of this valuation. Therefore, we make use of an action function

$$\text{action} : \mathbb{A} \times \mathbb{S} \to \mathbb{A}$$

where $\text{action}(a, \sigma)$ is evaluation of action a in state σ. The action function makes use of the evaluation function as defined in Definition 6.2.4: it evaluates all variables in the action to their values, if possible. We use abbreviation $a(\sigma)$ for $\text{action}(a, \sigma)$. So, e.g.,

$$\begin{array}{ll} & \text{action}(n := n + 1, n \mapsto 0 : \mathbb{N}) \\ = & \{ \text{ abbreviation } \} \\ & (n := n + 1)(n \mapsto 0 : \mathbb{N}) \\ = & \{ \text{ evaluation } \} \\ & n := 0 + 1 \end{array}$$

On the other hand, execution of an action might affect valuations in the state of the application, since assignments to variables in the scope can occur. Therefore, we introduce an effect function:

$$\text{effect} : \mathbb{A} \times \mathbb{S} \to \mathbb{S}$$

where $\text{effect}(a, \sigma)$ is the effect of action a on state σ. The effect function uses the substitution as defined in Definition 6.2.6. So, e.g.,

$$\begin{array}{ll} & \text{effect}(n := n + 1, n \mapsto 0 : \mathbb{N}) \\ = & \{ \text{ substitution } \} \\ & n \mapsto 1 : \mathbb{N} \end{array}$$

$$\frac{\langle x, (i \mapsto e(\sigma) : T) :: \sigma \rangle \downarrow}{\langle [\, i \mapsto e : T \mid x \,], \sigma \rangle \downarrow} \qquad \frac{\langle x, (i \mapsto e(\sigma) : T) :: \sigma \rangle \xrightarrow{a} \langle x', (i \mapsto c : T) :: \sigma' \rangle}{\langle [\, i \mapsto e : T \mid x \,], \sigma \rangle \xrightarrow{a} \langle [\, i \mapsto c : T \mid x' \,], \sigma' \rangle}$$

Table 6.1: Deduction rules for the scope operator.

By adding the state to the deduction rules using tuple notation $\langle \_, \_ \rangle$, this results in the following deduction rule:

$$\frac{}{\langle a, \sigma \rangle \xrightarrow{\text{action}(a,\sigma)} \langle \varepsilon, \text{effect}(a, \sigma) \rangle}$$

Since we allow any kind of types, we cannot put restrictions on operators to be used. Though in contrast to types, operators do influence process behaviour. First of all, they are important when evaluating guards as introduced in the conditional operators in Section 5.7. This can be seen by looking at the deduction rules for the conditional operators in Table 6.4 on page 65. This behaviour can be modelled without extending *DiCons*, since evaluations are contained in the premises of the deduction rules.

The second occurrence of variables, types and operators is when specifying local actions, procedures and functions. These actions can be self-explaining, for example when using the assignment and the multiplication operators: $n := n \times 2$. However, it might also be possible that functions show complex behaviour, e.g., when determining the highest bid in an auction using the highest function: $b := \text{highest}(B)$. In the latter case, the effect of the action on the state should be provided:

$$\text{effect}(b := \text{highest}(B), \sigma) \quad = \quad \sigma' \text{ where } b(\sigma') \in B(\sigma') \ \wedge \ \forall c{\in}B(\sigma')\ c \leq b(\sigma')\ .$$

We only give the changes of the state. For all variables for which no statement in the effect is provided, it is assumed that the action will not affect their valuations.

6.3.1 Semantics

The deduction rules for the scope operator can be found in Table 6.1. As can be seen, deduction rules are extended with a state and describe transitions between tuples of processes and states. For process x and state σ, such a tuple is denoted by $\langle x, \sigma \rangle$. Both results of applying the action and effect functions to the action and current state can be found in the deduction rules.

In Chapter 5 we also gave axiomatic semantics of the operators. However, since we

focus on the operational semantics in the remainder of this thesis, we do not give a complete axiomatisation of the operators introduced.

The deduction rules given in Chapter 5 are also extended with a state component. In the following section, we extend them with a time component as well. The extended deduction rules can be found in Table 6.4 on page 65.

6.4 Adding a Time Component

When specifying Internet applications it is useful that a time component is available. Time is measured on a discrete scale.

$\mathfrak{T}$ **Time**
Each time stamp is a constant of type $\mathfrak{T}$, measured on a discrete scale.

In the transition system, time can always proceed without changing the state of a system. Process behaviour is not affected directly by passage of time, thus each expression allows passage of time without change. However, boolean expression can contain the (dynamic) now constant, $now \in \mathfrak{T}$, and therefore evaluate differently due to passage of time. By use of the conditional operators, which are defined in Section 5.7, this in turn affects process behaviour.

As a result, the deduction rules are also extended with a time constant t, $t \in \mathfrak{T}$, and the effect and action functions get this time component as an extra parameter.

So,

$$\text{action} : \mathbb{A} \times \mathbb{S} \times \mathfrak{T} \rightarrow \mathbb{A}$$

where $\text{action}(a, \sigma, t)$ is evaluation of action a in state σ at time t. The action function makes use of the evaluation function as defined in Definition 6.2.4: it evaluates all variables in the action to their values, if possible. We use abbreviation $a(\sigma, t)$ for $\text{action}(a, \sigma, t)$.

The effect function is also extended with a time component:

$$\text{effect} : \mathbb{A} \times \mathbb{S} \times \mathfrak{T} \rightarrow \mathbb{S}$$

Function $\text{effect}(a, \sigma, t)$ returns the effect of execution of action a in state σ at time t.

6.4.1 Semantics

The deduction rule modelling a time step can be found in Table 6.2. As can be seen, deduction rules are extended with a time stamp and describe transitions between

tuples of processes, states and time stamps. A time step only increases time. The other components remain the same. However, as mentioned above, evaluation of expressions can give different results if time passes, and so a change in time can influence process behaviour.

$$\frac{}{\langle x, \sigma, t\rangle \xmapsto{tick} \langle x, \sigma, t+1\rangle}$$

Table 6.2: Deduction rule for the time step.

Deduction rules for the scope operator in combination with time can be found in Table 6.3. The only difference with the deduction rules given in Table 6.1 is the addition of the time component to the tuples.

$$\frac{\langle x, (i \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \downarrow}{\langle [\, i \mapsto e : T \mid x \,], \sigma, t\rangle \downarrow}$$

$$\frac{\langle x, (i \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \xrightarrow{a} \langle x', (i \mapsto c : T) :: \sigma', t\rangle}{\langle [\, i \mapsto e : T \mid x \,], \sigma, t\rangle \xrightarrow{a} \langle [\, i \mapsto c : T \mid x' \,], \sigma', t\rangle}$$

Table 6.3: Deduction rules for the scope operator with time.

Also the deduction rules for the empty process, atomic actions, the sequential and alternative composition and the additional operators introduced in Section 5.7 are extended with a state and time component. The extended rules can be found in Table 6.4. Again, the only difference with the deduction rules given before is the addition of state and time components. The use of the action and effect functions are shown in the second deduction rule.

$$\frac{}{\langle \varepsilon, \sigma, t\rangle \downarrow} \qquad \frac{}{\langle a, \sigma, t\rangle \xrightarrow{\text{action}(a,\sigma,t)} \langle \varepsilon, \text{effect}(a, \sigma, t), t\rangle}$$

$$\frac{\langle x, \sigma, t\rangle \downarrow, \quad \langle y, \sigma, t\rangle \downarrow}{\langle x \cdot y, \sigma, t\rangle \downarrow} \qquad \frac{\langle x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle}{\langle x \cdot y, \sigma, t\rangle \xrightarrow{a} \langle x' \cdot y, \sigma', t\rangle}$$

$$\frac{\langle x, \sigma, t\rangle \downarrow, \quad \langle y, \sigma, t\rangle \xrightarrow{a} \langle y', \sigma', t\rangle}{\langle x \cdot y, \sigma, t\rangle \xrightarrow{a} \langle y', \sigma', t\rangle} \qquad \frac{\langle x, \sigma, t\rangle \downarrow}{\langle x + y, \sigma, t\rangle \downarrow, \quad \langle y + x, \sigma, t\rangle \downarrow}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle}{\langle x + y, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle, \quad \langle y + x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle}$$

$$\frac{\langle x, \sigma, t\rangle \downarrow, \quad b(\sigma, t)}{\langle x \triangleleft b \triangleright y, \sigma, t\rangle \downarrow} \qquad \frac{\langle x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle, \quad b(\sigma, t)}{\langle x \triangleleft b \triangleright y, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle}$$

$$\frac{\langle y, \sigma, t\rangle \downarrow, \quad \neg b(\sigma, t)}{\langle x \triangleleft b \triangleright y, \sigma, t\rangle \downarrow} \qquad \frac{\langle y, \sigma, t\rangle \xrightarrow{a} \langle y', \sigma', t\rangle, \quad \neg b(\sigma, t)}{\langle x \triangleleft b \triangleright y, \sigma, t\rangle \xrightarrow{a} \langle y', \sigma', t\rangle}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle, \quad b(\sigma, t)}{\langle b \triangleright\!\!\triangleright x, \sigma, t\rangle \xrightarrow{a} \langle x' \cdot (b \triangleright\!\!\triangleright x), \sigma', t\rangle} \qquad \frac{\neg b(\sigma, t)}{\langle b \triangleright\!\!\triangleright x, \sigma, t\rangle \downarrow}$$

$$\frac{\langle x, \sigma, t\rangle \downarrow}{\langle b \triangleright\!\!- x, \sigma, t\rangle \downarrow} \qquad \frac{b(\sigma, t)}{\langle b \triangleright\!\!- x, \sigma, t\rangle \downarrow} \qquad \frac{\langle x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle, \quad \neg b(\sigma, t)}{\langle b \triangleright\!\!- x, \sigma, t\rangle \xrightarrow{a} \langle b \triangleright\!\!- x', \sigma', t\rangle}$$

Table 6.4: Deduction rules for the empty process, atomic actions, the sequential and alternative operators and the additional operators.

7

Modelling Internet Communication

In this chapter we formalise the communication primitives we make use of for modelling communication via the Internet. We summarise the elements of the alphabet we use for defining the interaction primitives in Section 7.1 and we give the operational semantics of the interaction primitives in Section 7.2.

7.1 Alphabet

In Section 4.3 we introduced the interaction primitives we make use of for specifying user interaction via the Internet. We will shortly summarise the interaction primitives below.

- $\leftarrow$ **Active server push**
  An *active push* takes place if the server sends a message to a client (arrow to the left) which is not directly the result of a request from that client. So the server initiates the interaction which is indicated by the arrow's tail "pointing" to the right.

- $\Leftarrow$ **Reactive server push**
  A *reactive push* takes place if the server sends a Web page (arrow to the left), not containing a Web form, to a client which is the result of a normal request from that client (client initiates, so tail to the left), i.e. not generated by filling out a previously received Web form.

⇄ **Reactive server pull (start session)**
This interaction takes place if a client sends a request to the server (upper horizontal line, tail to the left) on which the server responds by sending a Web form (middle line). This form is filled in and submitted by the client (lower line, arrow to the right). A reactive pull starts a session with one particular client.

⇄ **Session-oriented server pull (within session)**
In response to a prior form submission (upper, dashed line), the server sends a Web form to the client (middle line). Subsequently, the client submits the filled in form (lower line, arrow to the right). This interaction is repeated in the middle of a session.

⇄ **Session-oriented server push (end session)**
The server sends a non-interactive Web page to the client (lower line) in response to a prior form submission by the client (upper, dashed line). This interaction ends a session since no Web form can be filled in anymore.

For specifying the interaction primitives in process algebra, we add two atomic actions to the alphabet, viz. one for an HTTP request and one for an HTTP response. Next, we define the five interaction primitives of which four are having substructure. This substructure is specified in the deduction rules for the actions, which is explained in Section 7.2.

Before being able to give an alphabet, we first need to introduce some sets:

$\mathbb{U}$ **The universe of users**
This set contains all possible users. Each user interacting with an application is represented by a constant in $\mathbb{U}$.

$\mathbb{M}$ **The universe of messages**
All messages are uniquely identified by an element in $\mathbb{M}$ and can therefore be represented by their symbolic name m in $\mathbb{M}$.

$\mathbb{P}_i$ **The universe of input parameters**
A message can be extended with parameters. For request messages these parameters are variables that are assigned by a user by filling in a Web form, i.e., input parameters denoted by $i, i_1, i_2, \ldots$

$\mathbb{P}_o$ **The universe of output parameters**
For response messages and e-mail messages, the parameters are values of expressions to be included in the actual message that needs to be sent to the requesting user, i.e., output parameters denoted by $o, o_1, o_2, \ldots$

The alphabet that is used for specifying interactions, $\mathbb{C}$, contains the following sets of atomic actions:

- **Request actions**

$$\{\text{req}.u.\vec{\imath} \mid u \in \mathbb{U}, i_1, \ldots, i_n \in \mathbb{P}_i\}$$

  These actions represent URL requests or form submissions by a user u, extended with input parameters $\vec{\imath}$.

- **Response actions**

$$\{\text{resp}.u.m.\vec{o} \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o\}$$

  These actions represent the sending of a Web page or Web form m to user u with output parameters $\vec{o}$.

Apart from these two sets of atomic actions, the alphabet also contains actions representing the interaction primitives as defined in Section 4.3.2. They are defined by five sets:

- **Active server push actions**

$$\{u \leftarrow m(\vec{o}) \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o\}$$

  These actions represent the sending of an e-mail message m to user u with output parameters $\vec{o}$. The output parameters are expressions which are presented to the user. The active server push actions are also atomic actions.

- **Reactive server push actions**

$$\{u \rightleftharpoons m(\vec{o}) \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o\}$$

  These actions represent the sending of a plain Web page m to user u with output parameters $\vec{o}$ as a response to a URL request by user u. The output parameters are expressions which are presented to the user.

- **Reactive server pull actions**

$$\{u \Rightarrow m(\vec{o};\vec{\imath}) \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o, i_1, \ldots, i_n \in \mathbb{P}_i\}$$

  These actions represent the sending and submission of a Web form m with output parameters $\vec{o}$ and input parameters $\vec{\imath}$ to user u as a response to a URL request by user u. The output parameters $\vec{o}$ are expressions which are presented to the user. The input parameters $\vec{\imath}$ are variables which the user has to assign by filling in the form.

- **Session-oriented server pull actions**

$$\{u \overset{\vec{o}}{\leftrightarrows} m(\vec{o};\vec{i}) \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o, i_1, \ldots, i_n \in \mathbb{P}_i\}$$

  These actions represent the sending and submission of a Web form m with output parameters $\vec{o}$ and input parameters $\vec{i}$ to user u as a response to a prior submission of a Web form by user u. Again, the output parameters $\vec{o}$ are expressions which are presented to the user and the input parameters $\vec{i}$ are variables which the user has to assign.

- **Session-oriented server push actions**

$$\{u \overset{\vec{o}}{\leftarrow} m(\vec{o}) \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o\}$$

  These actions represent the sending of a plain Web page m to user u with output parameters $\vec{o}$ as a response to a prior submission of a Web form by user u. The output parameters are expressions which are presented to the user.

7.2 Semantics

The operational semantics of the interaction primitives depends on their HTTP request/response behaviour. The labels in the labelled transition system we use for modelling the operational behaviour are similar to the atomic actions for the request and response actions as defined in Section 7.1. Apart from request and response labels, we introduce a set of labels for representing the mail sending actions. The request label is extended by a valuation vector which contains the values assigned to the input parameters of the request action. So, we make use of the following set of action labels:

- **Mail sending actions**

$$\{\text{mail}.c.m.\vec{o} \mid c \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o\}$$

  These actions represent the sending of an e-mail message m to user c with output parameters $\vec{o}$. The output parameters are evaluated in the current state at the time of sending the e-mail.

- **Request actions**

$$\{\text{req}.c.\vec{i}.\vec{d} \mid c \in \mathbb{U}, i_1, \ldots, i_n \in \mathbb{P}_i, \vec{d} \in \text{type}(\vec{i})\}$$

  These actions represent URL requests or form submissions by a user c, extended with input parameters $\vec{i}$ and values $\vec{d}$ filled in by the user, which are assigned to the input parameters. Note that the length of $\vec{i}$ of course must be equal to the length of $\vec{d}$. Also note that the type function used depends on the state the process is.

- **Response actions**

$$\{\text{resp}.c.m.\vec{o} \mid c \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_k \in \mathbb{P}_o\}$$

These actions represent the sending of a Web page or Web form m to user c with output parameters $\vec{o}$. The output parameters are evaluated in the current state at the time of sending the response.

The deduction rules for both the atomic and composed actions for the interaction primitives can be found in Table 7.1.

$$\frac{u(\sigma,t) = c, \quad \vec{d} \in \text{type}(\vec{\imath},\sigma)}{\langle \text{req}.u.\vec{\imath},\sigma,t\rangle \xrightarrow{\text{req}.c.\vec{\imath}.\vec{d}} \langle \varepsilon, \sigma[\vec{d}/\vec{\imath}], t\rangle} \qquad \frac{u(\sigma,t) = c, \quad \vec{o}(\sigma,t) = \vec{d}}{\langle \text{resp}.u.m.\vec{o},\sigma,t\rangle \xrightarrow{\text{resp}.c.m.\vec{d}} \langle \varepsilon,\sigma,t\rangle}$$

$$\frac{u(\sigma,t) = c, \quad c \neq \bot, \quad \vec{o}(\sigma,t) = \vec{d}}{\langle u \leftarrow m(\vec{o}),\sigma,t\rangle \xrightarrow{\text{mail}.c.m.\vec{d}} \langle \varepsilon,\sigma,t\rangle} \qquad \frac{u(\sigma,t) = c}{\langle u \rightleftarrows m(\vec{o}),\sigma,t\rangle \xrightarrow{\text{req}.c.\varepsilon.\varepsilon} \langle \text{resp}.u.m.\vec{o},\sigma,t\rangle}$$

$$\frac{u(\sigma,t) = c}{\langle u \rightleftarrows m(\vec{o};\vec{\imath}),\sigma,t\rangle \xrightarrow{\text{req}.c.\varepsilon.\varepsilon} \langle \text{resp}.u.m.\vec{o} \cdot \text{req}.u.\vec{\imath},\sigma,t\rangle}$$

$$\frac{u(\sigma,t) = c, \quad \vec{o}(\sigma,t) = \vec{d}}{\langle u \rightleftarrows m(\vec{o};\vec{\imath}),\sigma,t\rangle \xrightarrow{\text{resp}.c.m.\vec{d}} \langle \text{req}.u.\vec{\imath},\sigma,t\rangle} \qquad \frac{u(\sigma,t) = c, \quad \vec{o}(\sigma,t) = \vec{d}}{\langle u \rightleftarrows m(\vec{o}),\sigma,t\rangle \xrightarrow{\text{resp}.c.m.\vec{d}} \langle \varepsilon,\sigma,t\rangle}$$

Table 7.1: Deduction rules for the interaction primitives.

As can be seen, the deduction rules are straightforward, looking at the HTTP structure of the interaction primitives as defined in Section 4.3.2 on page 40. The atomic actions can do a step labelled by the atomic action, and then successfully terminate. The composed actions can do a request or response step after which the rest of the HTTP interaction modelled by the primitive should take place. If a request is received in which input parameters are available, the valuations in the state are changed depending on the values included in the request. The message $m \in M$ is a constant and need therefore not be evaluated in the current state. Users and output parameters are variable and thus depend on the state the application is in. At the moment of the sending of responses and emails, the output parameters are evaluated and the resulting values are included in the corresponding messages.

Note that we do not allow mails to be sent to undefined users. This is because the application does not know where to send the mail to. However, unknown users can

interact with the application by not asking them to identify before starting the first interaction.

In the deduction rules given in Table 7.1 we do not take into account that multiple interactions with the application take place in parallel. Therefore, the coupling of requests to responses and vice versa is straight forward. However, when adding parallelism this coupling should be made explicit, which results in an adaptation of the deduction rules. How this is done is explained below, in Section 7.3.1.

7.3 Access Control

As mentioned in the domain space identification in Chapter 4 there is a need for an access control model . Using such a model makes it possible to manage permissions users have. In *DiCons*, permissions are implicitly included in the specification of the Internet applications: the set of permissions of a user is exactly the set of interactions he is allowed to execute.

We stated that users are collected in groups. As mentioned before, in *DiCons*, users are constants in the universe of users $\mathbb{U}$. So a group is a subset of this universe and an element of the universe of groups:

$\mathbb{G}$ **The universe of groups**
This set contains all possible groups:

$$\mathbb{G} = \mathcal{P}(\mathbb{U})$$

There exist three access mechanisms for starting a session with an Internet application:

1. The client accesses the application anonymously. No identification step takes place and all users in $\mathbb{U}$ are allowed to interact with the application.
2. The client accesses the application by executing an identification step, identifying himself as a user known to the application.
3. The client accesses the application and makes himself known to the application by registering. A registration step takes place and the user is added to a group of registered users. It should be able to restrict the set of users that are allowed to register.

In the remainder of this section we give a formalisation of these three access control mechanisms. Note that (possibly grouped) clients can access applications simulta-

$$\frac{\langle x,\sigma,t\rangle\downarrow,\quad \langle y,\sigma,t\rangle\downarrow}{\langle x\parallel y,\sigma,t\rangle\downarrow}\qquad \frac{\langle x,\sigma,t\rangle \xrightarrow[k]{a} \langle x',\sigma',t\rangle}{\langle x\parallel y,\sigma,t\rangle \xrightarrow[k0]{a} \langle x'\parallel y,\sigma',t\rangle,\quad \langle y\parallel x,\sigma,t\rangle \xrightarrow[k1]{a} \langle y\parallel x',\sigma',t\rangle}$$

Table 7.2: Deduction rules for the merge operator.

neously. Therefore, before formalising the access control mechanisms, we first introduce both the *merge* and the *replication operator*, which are used for the parallel composition of processes.

7.3.1 Parallel Composition

In order to put processes in parallel, we have the merge operator, or parallel composition operator, $\parallel$ at our disposal. Putting processes x and y in parallel, denoted by $x \parallel y$, means that the execution of x and y takes place concurrently. Operationally, this means that if one of the processes can execute an action, the parallel composition can also execute this action. The operational semantics are given in Table 7.2. The labels below the transition arrows (k, $k0$ and $k1$) are explained below.

When multiple sessions with one and the same user take place in parallel, it must be possible to determine the session from which local actions and interactions with that user are executed. This cannot be done by looking at the user himself. Therefore we introduce *session labels*. As can be seen by looking at the deduction rules, we use a sequence of bits for this labelling of transitions, with a 0 denoting the left component, and a 1 the right component. In essence, this is the same labelling as used by Degano and Priami [DP92] and others, often called *locations*.

This labelling could also be added to the tuple of processes, states and time stamps. However, session labels are only concerned with the actual branching structure, not with the state. Even stronger, labelling can differ for the same process when using one of the generalised parallel composition operators given below. Therefore we decide to have these labels as a parameter of the transition, putting them under the transition arrow.

In the model of *DiCons*, which is given in Chapter 9, session labels are added to all deduction rules such that all transitions become session aware.

Definition 7.3.1 *(Session label) Let $d \in \{0,1\}$. A* session label *k is defined by*

$$\begin{array}{rcll} k & =_{\text{def}} & \lambda & \textit{the empty session label} \\ & \mid & kd & \textit{a non-empty session label .} \end{array}$$

$$\frac{\langle [\, u \mapsto \bot : \mathbb{U} \mid x \,], \sigma, t \rangle \downarrow}{\langle ?_u x, \sigma, t \rangle \downarrow} \qquad \frac{\langle [\, u \mapsto \bot : \mathbb{U} \mid x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto \bot : \mathbb{U} \mid x' \,], \sigma', t \rangle}{\langle ?_u x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto \bot : \mathbb{U} \mid x' \,] \parallel ?_u x, \sigma', t \rangle}$$

Table 7.3: Deduction rules for the anonymous replication operator.

The set of all possible session labels is denoted by $\mathbb{K}$. In addition to the binary merge operator, we introduce some generalised operators for parallel composition in the remainder of this section. These operators are of use when (multiple) users interact with the application in parallel.

7.3.2 Anonymous Interaction

As explained in Section 4.4.1 on page 42, anonymous interaction means that any user is allowed to interact with the application without having to identify himself. Lack of an identification step makes it possible for any user (i.e., all $u \in \mathbb{U}$) to start an interaction. Apart from simple reactive push interactions (sending Web pages without Web forms), when formally specifying anonymous interactions, there is a need for sessions: although the application does not know who it is communicating with, there still can be sequences of related interactions.

From the application's point of view, it makes no sense to let one anonymous user interact with it while another one is denied access. Even stronger, one user can interact with the application several times, since it does not have to be known to the application that it is interacting with the same user. These interactions with several users can take place simultaneously. Therefore, in addition to the binary merge operator, we introduce a generalised parallel composition operator: the so-called *anonymous replication* operator ($?$). Process $?_u x$ expresses that all users u in the universe of users $\mathbb{U}$ can execute (inter)actions in process x between user u and the central application in parallel and more than once. The user is anonymous and thus is not identifiable by the application. The u can occur in x and is bound to $\bot$ as soon as the first action of x is executed. As a result, sessions can take place, however, no active push interaction can occur, i.e., no e-mail can be sent. The operational semantics of the anonymous replication operator is given in Table 7.3.

We make use of the anonymous replication operator to specify the anonymous interaction behaviour. Since any of the users $u \in \mathbb{U}$ can start a session with an application, the domain of the replication operator is the entire universe of users:

$$?_u\, x$$

$$\frac{\forall_{c \in G(\sigma,t)} \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle !_{u \in G} x, \sigma, t \rangle \downarrow}$$

$$\frac{c \in G(\sigma, t) \in \mathbb{G}, \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle !_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, !_{u \in G} x, \sigma', t \rangle}$$

Table 7.4: Deduction rules for the replication or bang operator.

7.3.3 Identification

Apart from anonymous interaction, we also explained in Section 4.4.1 on page 42 that clients can identify themselves. Furthermore, we mentioned that since many identification protocols are available, we abstract from the actual identification protocol. Therefore, we introduce the so-called *replication* or *bang* operator (!). Process $!_{u \in G} x$ expresses that all users u in group G can execute (inter)actions in process x between user u and the central application in parallel and more than once. Again, the u can occur in x and is bound as soon as the first action of x is executed. This implicitly models an identification mechanism as introduced in Section 4.4.1. We use the bang operator on a group of users G ($G \in \mathbb{G}$), where G contains those users that are allowed access:

$$!_{u \in G}\, x$$

In this case, users in group G are allowed to execute process x more than once.

The operational semantics of the bang operator is given in Table 7.4.

In several cases, like the *Sinterklaaslootjes* and the voting example, users are allowed to execute process x only once: one can only draw one ticket and is allowed to vote only once. Therefore, we introduce the *generalised merge* or *generalised parallel composition* operator:

$$\|_{u \in G}\, x$$

Process $\|_{u \in G} x$ specifies that all users u ($u \in G$) execute (inter)actions in process x between user u and the central application in parallel but only once. So, in contrast to the replication operation, after execution of x for all users u in group G, the generalised merge operation terminates. See Table 7.5 for the operational semantics of the generalised merge operator.

Note that identification takes place *before* the first action of x is executed. The identification step itself is not contained in the operational semantics.

$$\frac{\forall_{c \in G(\sigma,t)} \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle \|_{u \in G} x, \sigma, t \rangle \downarrow}$$

$$\frac{c \in G(\sigma,t) \in \mathbb{G}, \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, \|_{u \in G \setminus \{c\}} x, \sigma', t \rangle}$$

Table 7.5: Deduction rules for the generalised merge operator.

7.3.4 Registration

In Section 4.4.1 on page 42 we also introduced registration. For specifying registration, we introduce extensions for both the bang and the generalised merge operator.

Using the *extended bang* or *extended replication* operator, process $!^{H}_{u \in G}\, x$ expresses that all users in group G are allowed to execute process x more than once. Group H contains the set of users that is known to the application. We call this group H the *registered group*. If the user is not known to the application (yet), i.e. $u \notin H$, registration takes place and the user is added to the group of users H. If the user is already contained in group H, identification takes place. This semantics is implicitly expressed in Table 7.6. Depending on user c being in group H in the last two deduction rules, either registration or identification takes place.

Expression $!^{H}_{u \in G}\, x$ expresses that any user in group G can register and subsequently execute x. If H initially contains users, these users can log in without first having to register. Note that from the user's point of view a different identification step takes place depending on whether the user is contained in H: If so, identification as shown in the left-hand MSC in Figure 4.3 on page 43 takes place. If not, he registers after which he logs in, as depicted in the right-hand MSC in Figure 4.3. This distinction is not expressed in the operational semantics. For expressing that any user can register, we can use this operator using the universe of users as domain:

$$[\, G \mapsto \mathbb{U} : \mathbb{G} \,|\, !^{H}_{u \in G}\, x \,]$$

Apart from the extended bang operator, we introduce the *extended generalised merge* or *extended generalised parallel composition* operator. Process $\|^{H}_{u \in G}\, x$ shows a similar behaviour, except that all users in G are allowed to execute process x once. Again, group H contains the users that are already known by the application. The semantics of the extended generalised merge operator can also be found in Table 7.6.

$$\frac{\forall_{c \in G(\sigma,t)} \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle \mathbf{!}^{H}_{u \in G} x, \sigma, t \rangle \downarrow} \qquad \frac{\forall_{c \in G(\sigma,t)} \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle \|^{H}_{u \in G} x, \sigma, t \rangle \downarrow}$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \mathbf{!}^{H}_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, \mathbf{!}^{H}_{u \in G} x, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle}$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|^{H}_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, \|^{H}_{u \in G \setminus \{c\}} x, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle}$$

Table 7.6: Deduction rules for the extended generalised parallel composition operators.

Note that since the valuation of group H is affected by both processes $\mathbf{!}^{H}_{u \in G}$ and $\|^{H}_{u \in G}$, this H must be declared in the scope of both processes.

8

Modelling Transactional Behaviour

Our goal is to give formal specifications of Internet applications for distributed consensus, which are interactive systems. Therefore we need a formalism to specify transactional processes, as introduced in Section 4.2.1. Since the language we make use of for specifying Internet applications is based on process algebra, we have to model the transactional behaviour of processes in process algebra too. Both transactions and process algebra are widely used, however, as far as we know, no formalism *dedicated to* expressing transactions using process algebra has been developed before. However, in Section 8.7 we do mention some formalisms that can serve as a basis for this purpose.

In this chapter we join the concepts of transactions and process algebra which leads to a nice formalism for specifying transactional behaviour. Although we prove that transactional behaviour can already be specified in a standard process algebra extended with linear recursion, introducing specific transactional operators cause specifications to be shorter and thus more legible and manageable. In this chapter we present these operators by giving both an axiomatic and operational semantics. The concepts introduced here are used later, in Chapter 9, for modelling Internet applications.

We abstract from parts of the concept which must be added to make the formalism useful for specifying real-life processes. However, we summarise what has to be done to put the formalism into practice, e.g. by combining transactions with the formalism for specifying states as introduced in Chapter 6. In our opinion, these extensions do not influence the complexity of the formalism, however, a lot of (syn-

tactical) extensions have to be added.

In Section 4.2.1 we informally introduced transactions. In Section 8.1, transactions are explained in more detail. Apart from that, we give an example by which the use of transactional processing is clarified. Next, in Section 8.2, we adapt the concept of transactions in such a way that we are able to specify transactional behaviour using process algebra. We introduce both axiomatic and operational semantics for the process algebra with transactions (PAtrans), which can be read as a stand-alone algebraic formalism. In Section 8.3 we give two more examples of the use of the transactional operators and we do some calculations using the axioms introduced in Section 8.2. We prove some properties of the process algebra in Section 8.4. We discuss what has to be done to make the process algebra useful for modelling real-life applications. In Section 8.5 we combine transactions with states. Degrees of isolation are discussed in Section 8.6. Finally, we discuss related work in Section 8.7.

Parts of this chapter are presented in [Bee02].

8.1 Introduction to Transactions

In Section 4.2.1 we shortly introduced transactions. As mentioned there, Gray and Reuter [GR93] define a transaction as a "set" of (inter)actions which occur "as a group". By means of some examples, we concluded that transactions help in coupling related actions that either should all succeed or none of them should succeed. Apart from that, we concluded that transactions help in maintaining data integrity when parallel processes access shared data.

We introduced the so-called ACID properties, where ACID is an acronym for *atomicity*, *consistency*, *isolation* and *durability*. Processes that meet all four characteristics are called *transactions*. Transactions, and therefore atomic actions, are the basic building blocks for constructing applications. In the first part of this chapter (Sections 8.2 to 8.4) we construct a process algebra, PAtrans, that only focusses on the isolation property of transactions. Since we do not take state changes themselves into account in PAtrans, we do not discuss atomicity here. However, we do specify the rolling back and submitting of transactions, which can also be used for modelling the consistency property. Durability goes beyond the scope of this thesis, since we do not model failure of the system in which the transaction takes place. We assume that the environment in which applications are executed preserves durability.

Transactions can be nested, so transactions might contain subtransactions. This nesting can be useful when having subprocesses that behave like transactions, like the payment from one bank account to the other from within a larger transaction.

In general, a transaction consists of subtransactions, read and write actions, ended

by a commit action. If during a transaction something goes wrong, a rollback takes places, undoing all data changes, and the transaction can start over again. If all actions succeed, the commit statement causes the data changes to be durable.

During execution of parallel transactions, a transaction can lock other transactions by accessing shared data. That is, transactions can cause other transactions to come in a state in which they are not allowed to execute specific actions. This locking mechanism prevents accessing so-called dirty data (i.e. data that has been changed, but not committed yet) by using *read locks*. Furthermore, by using *write locks* it prevents having lost updates, i.e. changed data is updated by another transaction before it had been committed. Unlocking takes place while committing or rolling back a transaction.

In this chapter we focus on transactions that are not allowed to update data that is updated by another running transaction, so-called first degree isolated transactions. (See Section 8.6 for more information on degrees of isolation.) So we only take write locks into account. Read locks can be added to the formal definition for transactions in a similar way as write locks. So by leaving out read actions and read locks we do not reduce the complexity in a major way. In PAtrans we focus on specifying transactions, so we also leave out the explicit changes to the state space. We concentrate on the transactional behaviour of processes, which does not take into account *how* the data changes but *whether* the data changes. The actual changes of the state are introduced in Section 8.5.

To give an idea on how transactions are used, we give a small example which nicely shows the behaviour of the transactional operator in defining processes. In this section, we only give some informal definitions of the operators. They are formalised in later sections.

Have a look at the two processes given in Figure 8.1. The assignments to variable a are atomic actions. By using the $\cdot$ operator we compose these actions into sequentially executable processes, as explained in Section 5.2. E.g., $a := 0 \cdot a := a + 2$ specifies that first a becomes 0 after which a is increased by 2.

Both processes consist of two subprocesses which are placed in parallel using the merge operator ($\|$). Repeatedly, a nondeterministic choice to execute an action from either the left-hand or the right-hand subprocess is taken. As can be seen, each subprocess sequentially executes two assignments to variable a. In the right-hand process, we make use of $\langle\langle$ and $\rangle\rangle$ brackets to embrace the subprocesses, which turns them into transactions. Since both transactional processes $\langle\langle a := 0 \cdot a := a + 2\rangle\rangle$ and $\langle\langle a := 1 \cdot a := a \times 2\rangle\rangle$ access shared variable a simultaneously, write access to variable a in one of the transactions locks the other transactions until a rollback ($\mathcal{R}$) or commit ($\mathcal{C}$) takes place. See Figure 8.1 for the intended process graphs of both processes. We only make use of gray arrows for the rollback transitions for readability.

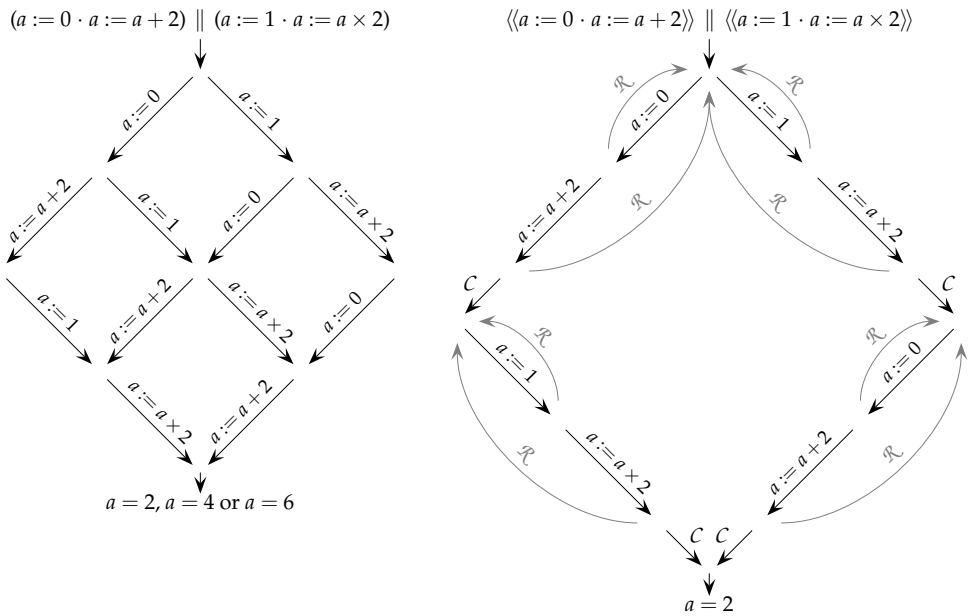


Figure 8.1: An example of the use of the transactional operator

In the left-hand process, normal interleaving of the two subprocesses on either side of the merge operator is allowed. This leads to 6 ($\frac{4!}{2!2!}$) possible traces, resulting in three possible outcomes: a equals 2, 4 or 6. Note that the nodes of the graph do *not* represent states. We only use the graph for showing the interleaving behaviour of the process. Representing unique states by nodes would result in a much larger graph, resulting in three terminating states for the different outcomes.

By turning both subprocesses into transactions, write access to a in one of the subprocesses leads to locking the other subprocess (see the right graph in Figure 8.1). If a subprocess is not finished, a rollback ($\mathcal{R}$) can take place, resulting in a transition to the state before starting the subprocess. If both actions in a subprocess are executed, the transaction can commit ($\mathcal{C}$), which leads to unlocking the other subprocess.

It can be easily seen that although we have an infinite number of possible executions (viz. rollbacks can take place any number of times), if the process finishes then a equals 2. In the end of this chapter we explain how transactional behaviour can be combined with states as presented in Chapter 6, such that valuations of a in a transactional environment can be modelled.

The example given in this section nicely shows the expressiveness of the transactional operator for defining transactional behaviour. In the remainder of this chapter we give an extension to the Basic Process Algebra which enables us to formally spec-

ify transactional behaviour.

8.2 A Process-Algebraic Approach

As mentioned in Chapter 5, our starting point is an algebraic axiomatisation $\mathrm{BPA}_{\delta\varepsilon}$ (Basic Process Algebra with deadlock and the empty process). The signature of $\mathrm{BPA}_{\delta\varepsilon}$ consists of action alphabet $\mathbb{A}$, alternative composition operator $+$, sequential composition operator $\cdot$ and constants δ and ε. The axioms for $\mathrm{BPA}_{\delta\varepsilon}$, A1–9, are given in Tables 5.1, 5.2 and 5.3, their operational semantics in Table 5.4.

To be able to formally specify the transactional behaviour as discussed in the former section, we introduce some new operators and extensions to the alphabet in this section. The algebra is named PAtrans.

8.2.1 Transactional Operator

To group actions into transactions, we need a transactional composition operator. As mentioned in Section 8.1, we turn a process into a transaction by making use of the *transactional operator*, i.e., by embracing it using $\langle\langle$ and $\rangle\rangle$ brackets.

We focus on transactional behaviour of the processes, so we leave out the actual data changes. Furthermore, we abstract from read access to shared variables. Instead of writing $a := a + 2$ or $a := 1$, we simply write a. At this moment, we are not interested in the exact value of a, but in the fact that the valuation of a might change. This makes the algebra much more readable without losing expressiveness. In a later stage, in Section 8.5, we show how the algebra can be extended such that evaluation of variables is possible.

As a result we can look at an action a as being a write action on a shared variable which is uniquely identifiable by a. The right process in Figure 8.1, for example, would be modelled by $\langle\langle a \cdot a\rangle\rangle \parallel \langle\langle a \cdot a\rangle\rangle$.

If a transaction executes action a, $a \in \mathbb{A}$, all transactions running in parallel with this transaction should be locked with respect to write access to shared variable a. During execution of a transaction, something can go wrong, e.g. a connection gets lost or a time-out takes place. If this happens, the entire transaction has to be rolled back, unlocking all other transactions that were locked by actions executed in this transaction. After this rollback, the transaction can start over again. If no rollback takes place, the transaction can commit, causing other transactions to get unlocked as well. For specifying this mechanism, we make use of an auxiliary operator $\langle\langle \_, \_, \_\rangle\rangle$. The first parameter is used for storing the actual transactional process. In case of a rollback, we make use of this parameter to restart the process. The second parameter is

used for storing the set of executed actions, i.e. the shared variables that are updated. This set is used for the unlocking of other transactions (and resetting the variable's values). A set is sufficient since all variables have only one value before being updated by a transaction. Finally, the last parameter contains that part of the process that needs to be executed before the transaction can commit. So $\langle\langle x, A, y\rangle\rangle$ represents *"transactional process x, which has already executed the set of actions A and still has to execute process y before a commit statement can take place"*.

To model transactional behaviour, first of all, we extend the alphabet. The idea is to turn actions into lockable actions when they are executed from within a transaction. This is done by adding a so-called lock counter to them. If a parallel running transaction has executed a similar action before, their lock counters are increased such that the actions get locked. Execution of a rollback or commit action unlocks the actions by decreasing the lock counters. For example, if not taking into account rollback actions, process $\langle\langle a \cdot b\rangle\rangle$ equals process $a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}}$.

To specify the locking and unlocking behaviour, we introduce both locking and unlocking operators. Execution of a lockable action in parallel to another process results in the application of the locking operator to the parallel running process. This operator causes the lock counters of similar actions to be increased. On the other hand, execution of an unlocking action introduces the unlocking operator, which is also applied to the parallel running process. This operator decreased the lock counters of actions locked by the transaction that is committed or rolled back.

We give an example (modulo rollback actions) of the execution of locking and unlocking actions and the application of the locking and unlocking operators to get an idea of the intended behaviour:

$$
\begin{aligned}
\langle\langle a \cdot b\rangle\rangle \parallel \langle\langle a \cdot b\rangle\rangle &\overset{(1)}{=} a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}} \parallel a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}} \\
&\overset{(2)}{=} a_0 \cdot (b_0 \cdot \mathcal{C}_{\{a,b\}} \parallel [a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}}]_a) \\
&\overset{(3)}{=} a_0 \cdot (b_0 \cdot \mathcal{C}_{\{a,b\}} \parallel a_1 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}}) \\
&\overset{(4)}{=} a_0 \cdot b_0 \cdot (\mathcal{C}_{\{a,b\}} \parallel [a_1 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}}]_b) \\
&\overset{(5)}{=} a_0 \cdot b_0 \cdot (\mathcal{C}_{\{a,b\}} \parallel a_1 \cdot b_1 \cdot \mathcal{C}_{\{a,b\}}) \\
&\overset{(6)}{=} a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}} \cdot (\varepsilon \parallel \lfloor a_1 \cdot b_1 \cdot \mathcal{C}_{\{a,b\}} \rfloor_{\{a,b\}}) \\
&\overset{(7)}{=} a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}} \cdot (\varepsilon \parallel a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}}) \\
&\overset{(8)}{=} a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}} \cdot a_0 \cdot b_0 \cdot \mathcal{C}_{\{a,b\}}
\end{aligned}
$$

In step 1, we introduce the lockable and unlocking actions. Next, in step 2, lockable action a_0 is executed, applying the locking operator $[\_]_a$ to the parallel running process. This increases the lock counters of similar lockable actions (step 3), in this case action a_0 becomes action a_1. As a result a_1 is not allowed to be executed, so

only the left-hand process can continue. Step 4 introduces the locking operator of action b which is applied to the process in step 5: b_0 becomes b_1. Next, in step 6, the transaction commits by executing the $\mathcal{C}_{\{a,b\}}$ action, unlocking the parallel running process with respect to actions a and b using unlocking operator $\lfloor \_ \rfloor_{\{a,b\}}$. This operator decreases the counters in step 7, allowing the other transaction to continue by executing action a_0.

Note that at this level of abstraction no distinction is drawn between the parallel executed transactions. However, if e.g. in the left-hand process a represents $a := a + 1$ and in the right-hand side process a represents $a := a \times 2$, the process in step 2 should be doubled using the alternative composition operator.

Alphabet

Process $\langle\langle x, A, y \rangle\rangle$ is a transactional process with body x, which still has to execute process y before it can commit. Set A contains the variables that are locked by the transaction.

As a result, transaction $\langle\langle x, \emptyset, x \rangle\rangle$ equals transaction $\langle\langle x \rangle\rangle$, which has not executed any of its actions yet. If a transaction $\langle\langle x, A, y \rangle\rangle$ has already executed an action, i.e. if $A \neq \emptyset$, it can roll back, using rollback action $\mathcal{R}_A$. This causes all parallel running transactions that are locked with respect to variables in A to get unlocked. Next, $\langle\langle x \rangle\rangle$ can start over again. If a transaction commits, action $\mathcal{C}_A$ is executed which also unlocks other transactions that run in parallel to the committed transaction. Since we abstract from the data changes, $\mathcal{R}_A$ and $\mathcal{C}_A$ behave equally. Therefore, we make use of $\mathcal{U}$ (Unlocking action) to represent either $\mathcal{C}$ or $\mathcal{R}$.

Definition 8.2.1 *Let A be a set of actions, $A \subseteq \mathbb{A}$ and $\mathcal{U}$ be an unlocking action, $\mathcal{U} \in \{\mathcal{C}, \mathcal{R}\}$. Then $\mathcal{U}_A$ is the atomic unlocking action that unlocks all actions in A that are locked in parallel running transactions, once. Set A is called the* unlocking set.

We introduce a new action alphabet, $\mathbb{UL}$, containing unlocking actions:

$$\mathbb{UL} \quad =_{\text{def}} \quad \{\mathcal{U}_A \mid \mathcal{U} \in \{\mathcal{C}, \mathcal{R}\}, A \subseteq \mathbb{A}\} \ .$$

As stated in Definition 8.2.1, execution of an unlocking action unlocks actions once. If more than two transactions run in parallel, actions can get locked more than once: each execution of a lockable action locks parallel executed actions. Therefore, we provide a mechanism to extend actions with a lock counter indicating how many times the action is locked.

Definition 8.2.2 *Let a be an action and n be a natural number. Then a_n is a lockable action. Action a_n represents action a which is locked n times. In a_n, n is called a* lock counter.

$\langle\langle x \rangle\rangle$	$=$	$\langle\langle x, \emptyset, x \rangle\rangle$		TR1
$\langle\langle x, \emptyset, \delta \rangle\rangle$	$=$	δ		TR2
$\langle\langle x, A, \delta \rangle\rangle$	$=$	$\mathcal{R}_A \cdot \langle\langle x \rangle\rangle$	if $A \neq \emptyset$	TR3
$\langle\langle x, \emptyset, \varepsilon \rangle\rangle$	$=$	$\mathcal{C}_\emptyset$		TR4
$\langle\langle x, A, \varepsilon \rangle\rangle$	$=$	$\mathcal{C}_A + \mathcal{R}_A \cdot \langle\langle x \rangle\rangle$	if $A \neq \emptyset$	TR5
$\langle\langle x, \emptyset, \mathcal{U}_B y \rangle\rangle$	$=$	$\mathcal{U}_\emptyset \cdot \langle\langle x, \emptyset, y \rangle\rangle$		TR6
$\langle\langle x, A, \mathcal{U}_B y \rangle\rangle$	$=$	$\mathcal{U}_\emptyset \cdot \langle\langle x, A, y \rangle\rangle + \mathcal{R}_A \cdot \langle\langle x \rangle\rangle$	if $A \neq \emptyset$	TR7
$\langle\langle x, \emptyset, a_n y \rangle\rangle$	$=$	$a_n \cdot \langle\langle x, \{a\}, y \rangle\rangle$		TR8
$\langle\langle x, A, a_n y \rangle\rangle$	$=$	$a_n \cdot \langle\langle x, A \cup \{a\}, y \rangle\rangle + \mathcal{R}_A \cdot \langle\langle x \rangle\rangle$	if $a \notin A \wedge A \neq \emptyset$	TR9
$\langle\langle x, A, a_n y \rangle\rangle$	$=$	$a \cdot \langle\langle x, A, y \rangle\rangle + \mathcal{R}_A \cdot \langle\langle x \rangle\rangle$	if $a \in A$	TR10
$\langle\langle x, \emptyset, a y \rangle\rangle$	$=$	$a_0 \cdot \langle\langle x, \{a\}, y \rangle\rangle$		TR11
$\langle\langle x, A, a y \rangle\rangle$	$=$	$a_0 \cdot \langle\langle x, A \cup \{a\}, y \rangle\rangle + \mathcal{R}_A \cdot \langle\langle x \rangle\rangle$	if $a \notin A \wedge A \neq \emptyset$	TR12
$\langle\langle x, A, a y \rangle\rangle$	$=$	$a \cdot \langle\langle x, A, y \rangle\rangle + \mathcal{R}_A \cdot \langle\langle x \rangle\rangle$	if $a \in A$	TR13
$\langle\langle x, A, y + z \rangle\rangle$	$=$	$\langle\langle x, A, y \rangle\rangle + \langle\langle x, A, z \rangle\rangle$		TR14

Table 8.1: Axioms for the transactional operator.

Again, we introduce a new action alphabet, $\mathbb{L}$, containing lockable actions:

$$\mathbb{L} \quad =_{\text{def}} \quad \{a_n \mid a \in \mathbb{A}, n \in \mathbb{N}\} \ .$$

Axioms

If transaction $\langle\langle x, A, y \rangle\rangle$ executes action a for the first time, i.e. $a \notin A$, then a becomes lockable by extended it with a lock counter having value 0 and A is extended with a. If the transaction executed an a before, i.e. $a \in A$, no locking counter is added since the transaction already has exclusive rights on action a.

If a transaction $\langle\langle x, A, y \rangle\rangle$ has already executed actions, i.e., if $A \neq \emptyset$, then the transaction can rollback. So all processes $\langle\langle x, A, y \rangle\rangle$ have a summand $\mathcal{R}_A \cdot \langle\langle x \rangle\rangle$ if A is not empty.

We now have all ingredients for giving the axiomatic semantics of the transactional operators, TR1–14 in Table 8.1.

Axiom TR2 states that if the transaction does not successfully terminate, the composed process does not successfully terminate. However, as stated in TR3, it can roll back if and only if actions have been executed before, i.e. if $A \neq \emptyset$. If the transaction successfully terminates, it can commit (TR4–5). Since $A \neq \emptyset$ in TR5, the rollback summand is also included.

TR6 and TR7 state that if the next action in a transaction is an unlock action ($\mathcal{C}_B$ or $\mathcal{R}_B$), then this action is the result of a nested subtransaction. This unlock action

should only unlock actions from other parallel executed subtransactions. It should *not* unlock actions outside the transactions. Therefore, B is replaced by $\emptyset$.

For the other actions, it holds that they stay lockable (TR8–9) or become lockable (TR11–12) if they are executed by the transaction.

In TR10 and TR13 it is stated that action a which already was executed by the transaction ($a \in A$) does not need to be locked (any longer). Reaching this action implies that the transaction has already executed a similar action and therefore has locked other parallel executed transactions with respect to this action: only the first occurrence of an action in a transaction gets locked. Note that $a \in A$ implies $A \neq \emptyset$ and thus rolling back is also possible.

As mentioned, all transactions might roll back if they have executed actions before, i.e., there exists a summand $\mathcal{R}_A \cdot \langle\langle x \rangle\rangle$ if the A in $\langle\langle x, A, y \rangle\rangle$ is not empty (TR3, TR5, TR7, TR9–10 and TR12–13).

Finally, Axiom TR14 states the distributivity of the auxiliary transactional operator over the alternative composition. Note that $\langle\langle x + y \rangle\rangle = \langle\langle x \rangle\rangle + \langle\langle y \rangle\rangle$ does *not* hold as a result of possible roll back actions.

Semantics

The deduction rules for the sequential composition operator and alternative composition operator as given in Table 5.4 on page 53 can also be applied to actions in the locking and unlocking alphabet.

The semantics of the transactional operator is given by the term deduction system induced by the deduction rules shown in Table 8.2. The variables are defined as in the axioms. We make a case distinction over the atomic actions that can be executed.

Rule 8 handles rollback actions: If a transaction has started, i.e., $A \neq \emptyset$, the transaction can roll back by executing a rollback action $\mathcal{R}_A$, after which the transaction can start over again.

If a process terminates successfully, the transaction can commit, after which it can terminate (rules 9 and 10). An unlocking action that comes from within a transaction may not influence actions outside the transaction (rules 11 and 14). Furthermore, normal actions and lockable actions can always be executed (rules 12, 13, 15–18). Depending on whether they have been executed by the transaction before, i.e. if a is in A, they become or stay lockable and they are added to the set of executed actions A.

$$\frac{A \neq \emptyset}{\langle\langle x, A, y\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle}\,{}^{8} \qquad \frac{x \downarrow}{\langle\langle x\rangle\rangle \xrightarrow{\mathcal{C}_\emptyset} \varepsilon}\,{}^{9} \qquad \frac{y \downarrow}{\langle\langle x, A, y\rangle\rangle \xrightarrow{\mathcal{C}_A} \varepsilon}\,{}^{10}$$

$$\frac{x \xrightarrow{\mathcal{U}_B} x'}{\langle\langle x\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, \emptyset, x'\rangle\rangle}\,{}^{11} \qquad \frac{x \xrightarrow{a_n} x'}{\langle\langle x\rangle\rangle \xrightarrow{a_n} \langle\langle x, \{a\}, x'\rangle\rangle}\,{}^{12}$$

$$\frac{x \xrightarrow{a} x'}{\langle\langle x\rangle\rangle \xrightarrow{a_0} \langle\langle x, \{a\}, x'\rangle\rangle}\,{}^{13} \qquad \frac{y \xrightarrow{\mathcal{U}_B} y'}{\langle\langle x, A, y\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, A, y'\rangle\rangle}\,{}^{14}$$

$$\frac{y \xrightarrow{a_n} y', \quad a \notin A}{\langle\langle x, A, y\rangle\rangle \xrightarrow{a_n} \langle\langle x, A \cup \{a\}, y'\rangle\rangle}\,{}^{15} \qquad \frac{y \xrightarrow{a_n} y', \quad a \in A}{\langle\langle x, A, y\rangle\rangle \xrightarrow{a} \langle\langle x, A, y'\rangle\rangle}\,{}^{16}$$

$$\frac{y \xrightarrow{a} y', \quad a \notin A}{\langle\langle x, A, y\rangle\rangle \xrightarrow{a_0} \langle\langle x, A \cup \{a\}, y'\rangle\rangle}\,{}^{17} \qquad \frac{y \xrightarrow{a} y', \quad a \in A}{\langle\langle x, A, y\rangle\rangle \xrightarrow{a} \langle\langle x, A, y'\rangle\rangle}\,{}^{18}$$

Table 8.2: Deduction rules for the transactional operators.

8.2.2 Locking and Unlocking Operators

As mentioned in Section 8.1, locking comes in when processes are put in parallel. Locking takes place by using lock counters. When putting locking and lockable actions in parallel, lock and unlock operators are needed for increasing and decreasing these lock counters. We introduce two operators, $[x]_a$ and $\lfloor x \rfloor_A$ on processes to lock and unlock processes, respectively. The symbol used for locking, $[x]_a$ can be seen as a closed box containing process x that is locked using key a. Unlocking a process opens the box, so therefore, we use $\lfloor x \rfloor_A$ notation.

The locking operator has two parameters, process x and action a. $[x]_a$ means that all lockable a actions in x get locked (once more). Unlocking operator $\lfloor x \rfloor_A$ also has two parameters, viz. process x and a set of actions A, meaning that all locked actions that occur in x which are elements of A get unlocked once.

Axioms

The axioms for both the locking and unlocking operators, L1–7 and UL1–7, are given in Table 8.3.

Both the locking and unlocking operator do only influence lockable actions. As can

$[\delta]_b$	=	δ		L1
$[\varepsilon]_b$	=	ε		L2
$[\mathcal{U}_B x]_b$	=	$\mathcal{U}_B \cdot [x]_b$		L3
$[a_n x]_b$	=	$a_{n+1} \cdot [x]_b$	if $a = b$	L4
$[a_n x]_b$	=	$a_n \cdot [x]_b$	if $a \neq b$	L5
$[ax]_b$	=	$a \cdot [x]_b$		L6
$[x + y]_b$	=	$[x]_b + [y]_b$		L7
$\lfloor \delta \rfloor_A$	=	δ		UL1
$\lfloor \varepsilon \rfloor_A$	=	ε		UL2
$\lfloor \mathcal{U}_B x \rfloor_A$	=	$\mathcal{U}_B \cdot \lfloor x \rfloor_A$		UL3
$\lfloor a_n x \rfloor_A$	=	$a_{n-1} \cdot \lfloor x \rfloor_A$	if $a \in A \wedge n > 0$	UL4
$\lfloor a_n x \rfloor_A$	=	$a_n \cdot \lfloor x \rfloor_A$	if $a \notin A \vee n = 0$	UL5
$\lfloor ax \rfloor_A$	=	$a \cdot \lfloor x \rfloor_A$		UL6
$\lfloor x + y \rfloor_A$	=	$\lfloor x \rfloor_A + \lfloor y \rfloor_A$		UL7

Table 8.3: Axioms for the locking and unlocking operators.

be seen, Axioms L1–3, L6, UL1–3 and UL6 state that for all non-lockable actions both the locking and unlocking operator behave like the identity. When locking, a lockable action gets its lock counter only increased if it equals the locking action (Axiom L4). When unlocking and the lockable action is in the set of unlocking actions, its lock counter gets decreased (Axiom UL4).

Semantics

The semantics of the locking and unlocking operators is given by the term deduction system induced by the deduction rules shown in Table 8.4.

Non-lockable actions are influenced by neither the locking (rules 19, 20 and 23) nor the unlocking operator (rules 24, 25 and 28). Depending on the parameters of the locking and unlocking operators, lock counters can be increased (rules 21 and 22) or decreased (rules 26 and 27), respectively.

8.2.3 Merge Operator

Up till now, we have not mentioned how the operators introduced so far co-operate to reach the expected transactional behaviour. We specify parallel composition using the merge ($\parallel$) and auxiliary left-merge ($\lfloor\!\lfloor$) operators based on the merge operators introduced in [BK82]. Since locking of transactions is only of interest when trans-

$$\frac{x\downarrow}{[x]_b\downarrow}\,{}_{19} \qquad \frac{x \xrightarrow{\mathcal{U}_A} x'}{[x]_b \xrightarrow{\mathcal{U}_A} [x']_b}\,{}_{20} \qquad \frac{x \xrightarrow{a_n} x', \quad a \neq b}{[x]_b \xrightarrow{a_n} [x']_b}\,{}_{21}$$

$$\frac{x \xrightarrow{a_n} x'}{[x]_a \xrightarrow{a_{n+1}} [x']_a}\,{}_{22} \qquad \frac{x \xrightarrow{a} x'}{[x]_b \xrightarrow{a} [x']_b}\,{}_{23} \qquad \frac{x\downarrow}{\lfloor x\rfloor_A\downarrow}\,{}_{24}$$

$$\frac{x \xrightarrow{\mathcal{U}_B} x'}{\lfloor x\rfloor_A \xrightarrow{\mathcal{U}_B} \lfloor x'\rfloor_A}\,{}_{25} \qquad \frac{x \xrightarrow{a_n} x', \quad (a \notin A \vee n = 0)}{\lfloor x\rfloor_A \xrightarrow{a_n} \lfloor x'\rfloor_A}\,{}_{26}$$

$$\frac{x \xrightarrow{a_n} x', \quad (a \in A \wedge n > 0)}{\lfloor x\rfloor_A \xrightarrow{a_{n-1}} \lfloor x'\rfloor_A}\,{}_{27} \qquad \frac{x \xrightarrow{a} x'}{\lfloor x\rfloor_A \xrightarrow{a} \lfloor x'\rfloor_A}\,{}_{28}$$

Table 8.4: Deduction rules for the locking and unlocking operators.

$x \parallel y$	$=$	$x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$		M1
$\delta \lfloor\!\lfloor x$	$=$	δ		M2
$\varepsilon \lfloor\!\lfloor \delta$	$=$	δ		M3
$\varepsilon \lfloor\!\lfloor \varepsilon$	$=$	ε		M4
$\varepsilon \lfloor\!\lfloor \mathbf{a}x$	$=$	δ		M5
$\varepsilon \lfloor\!\lfloor (x+y)$	$=$	$\varepsilon \lfloor\!\lfloor x + \varepsilon \lfloor\!\lfloor y$		M6
$\mathcal{U}_A x \lfloor\!\lfloor y$	$=$	$\mathcal{U}_A(x \parallel \lfloor y\rfloor_A)$		M7
$a_0 x \lfloor\!\lfloor y$	$=$	$a_0(x \parallel [y]_a)$		M8
$a_n x \lfloor\!\lfloor y$	$=$	δ	if $n > 0$	M9
$ax \lfloor\!\lfloor y$	$=$	$a(x \parallel y)$		M10
$(x+y) \lfloor\!\lfloor z$	$=$	$x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$		M11

$\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}, \quad a \in \mathbb{A}$

Table 8.5: Axioms for the merge operator.

actions run in parallel, accessing a shared data space, the parallel composition of transactions introduces the locking and unlocking operators.

Axioms

The axioms for the parallel composition, M1–11, are given in Table 8.5. Axioms M1–6 are similar to the axioms for the empty process as defined in [Vra97]. We use

$$\frac{x\downarrow,\quad y\downarrow}{x \parallel y\downarrow}\,{}^{29} \qquad \frac{x \xrightarrow{\mathcal{U}_A} x'}{x \parallel y \xrightarrow{\mathcal{U}_A} x' \parallel \lfloor y \rfloor_A,\quad y \parallel x \xrightarrow{\mathcal{U}_A} \lfloor y \rfloor_A \parallel x'}\,{}^{30}$$

$$\frac{x \xrightarrow{a_0} x'}{x \parallel y \xrightarrow{a_0} x' \parallel [y]_a,\quad y \parallel x \xrightarrow{a_0} [y]_a \parallel x'}\,{}^{31} \qquad \frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y,\quad y \parallel x \xrightarrow{a} y \parallel x'}\,{}^{32}$$

$$\frac{x\downarrow,\quad y\downarrow}{x \lfloor\!\lfloor y\downarrow}\,{}^{33} \qquad \frac{x \xrightarrow{\mathcal{U}_A} x'}{x \lfloor\!\lfloor y \xrightarrow{\mathcal{U}_A} x' \parallel \lfloor y \rfloor_A}\,{}^{34} \qquad \frac{x \xrightarrow{a_0} x'}{x \lfloor\!\lfloor y \xrightarrow{a_0} x' \parallel [y]_a}\,{}^{35} \qquad \frac{x \xrightarrow{a} x'}{x \lfloor\!\lfloor y \xrightarrow{a} x' \parallel y}\,{}^{36}$$

Table 8.6: Deduction rules for the merge operator.

notation a for an action in $\mathbb{A}$ and **a** for an action in $\mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$. If an unlocking action is executed in parallel with another process, the action is executed and the process running in parallel gets unlocked once (M7). Execution of a lockable action, which itself is not locked (its lock counter equals 0), locks the process running in parallel (M8). Execution of a locked action in parallel with another action is not allowed, so this process reaches a deadlock state (M9). All other actions do not influence the parallel running processes' behaviour (M10).

Semantics

The semantics of the merge operator is given by the term deduction system induced by the deduction rules shown in Table 8.6.

For defining the deduction rules for the parallel composition operators, again we distinguish between atomic action in $\mathbb{A}$, $\mathbb{L}$ and $\mathbb{UL}$. If both the left and right process terminate, the parallel composition of these processes terminates (rules 29 and 33). Execution of unlock actions cause the process running in parallel to get unlocked once (rules 30 and 34) where execution of lockable actions introduce the lock operator, causing the parallel running processes to get locked once more (rules 31 and 35). All other executions do not influence parallel running processes (rules 32 and 36).

8.2.4 Overview of PAtrans

An overview of PAtrans, the process algebra with transactions, can be found in Appendix A on page 219, where the alphabet, operators, axioms and deduction rules are summarised.

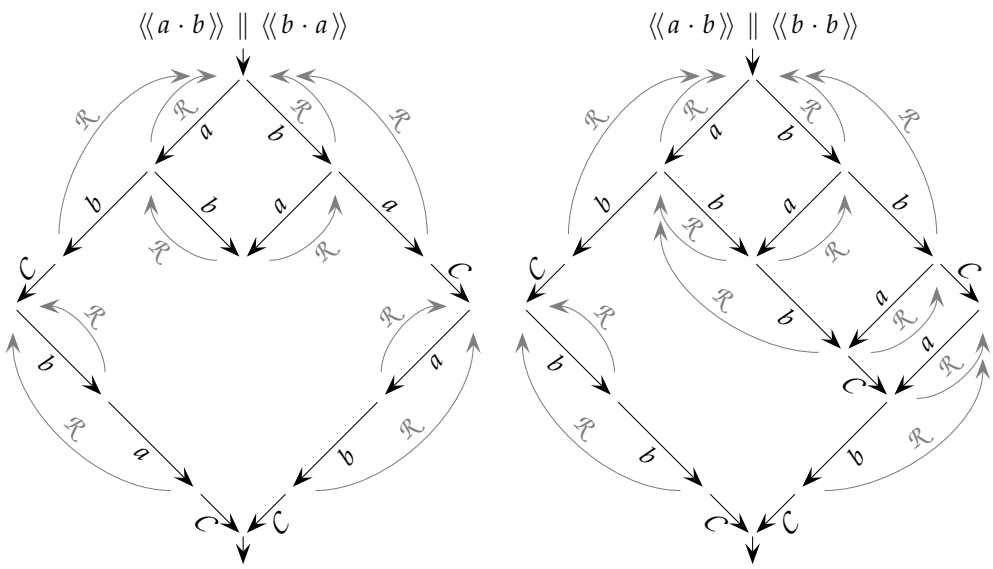


Figure 8.2: Examples of parallel running transactional processes with shared data.

8.3 Examples

In order to show the expressiveness of the operators introduced, two examples of parallel running transactional processes are given in Figure 8.2. In the left-hand process, the transactions can lock each other in such a way that the rolling back of at least one of the two subprocesses is unavoidable, viz. if the left-hand subprocess executes action a and the right-hand subprocess executes action b. In the example on the right-hand side, accessing shared data b in one of the subprocesses, i.e. executing action b, causes the other subprocess to get locked with respect to action b until a commit or rollback takes place.

In the remainder of this section we give an axiomatic calculation for the left-hand process in Figure 8.2. We calculate a recursive specification which no longer contains any of the newly introduced operators. We do the recursive calculation of $\langle\langle a \cdot b\rangle\rangle \parallel \langle\langle b \cdot a\rangle\rangle$ in several steps. First of all, we give a calculation of $\langle\langle a \cdot b\rangle\rangle$. This results in a recursive equation. Next, we calculate a recursive equation for process $\langle\langle a \cdot b\rangle\rangle \parallel \langle\langle b \cdot a\rangle\rangle$ using multiple variables. The result can be nicely mapped to the process graph given in Figure 8.2. We show this mapping of variables to nodes in Figure 8.4 together with an overview of the calculated recursive equations, so that the mapping can be verified. It can be concluded that the nodes (i.e. states) to which a roll back is possible need to be mapped to variables.

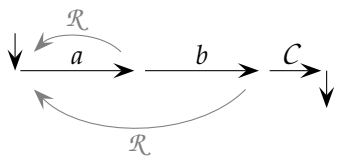


Figure 8.3: Process graph of $\langle\langle a \cdot b \rangle\rangle$.

Intuitively, process $\langle\langle a \cdot b \rangle\rangle$ is the process in which first a and then b is executed. After execution of one of the two actions, the process can roll back, restarting the process. After ending b, a commit action can take place which terminates the process. The intended process graph is given in Figure 8.3.

We give an axiomatic calculation of the process below.

Calculation 8.3.1

$$\begin{array}{ll} & \langle\langle a \cdot b \rangle\rangle \\ \stackrel{TR1}{=} & \langle\langle a \cdot b, \emptyset, a \cdot b \rangle\rangle \\ \stackrel{TR11}{=} & a_0 \cdot \langle\langle a \cdot b, \{a\}, b \rangle\rangle \\ \stackrel{TR12}{=} & a_0 \cdot (b_0 \cdot \langle\langle a \cdot b, \{a, b\}, \varepsilon \rangle\rangle + \mathcal{R}_{\{a\}} \cdot \langle\langle a \cdot b \rangle\rangle) \\ \stackrel{TR5}{=} & a_0 \cdot (b_0 \cdot (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle) + \mathcal{R}_{\{a\}} \cdot \langle\langle a \cdot b \rangle\rangle) \end{array}$$

As can be easily seen, abstracting from lock counters and unlocking sets leads to exactly the process as given in Figure 8.3. As a result from Calculation 8.3.1, we introduce two variables which are used in the calculation of process $\langle\langle a \cdot b \rangle\rangle \parallel \langle\langle b \cdot a \rangle\rangle$:

$$\begin{array}{lcl} Y & = & a_0 \cdot (b_0 \cdot (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot Y) + \mathcal{R}_{\{a\}} \cdot Y) \\ Z & = & b_0 \cdot (a_0 \cdot (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot Z) + \mathcal{R}_{\{b\}} \cdot Z) \end{array}$$

Both processes Y and Z are guarded recursive specifications which do not contain operators introduced in this chapter. Process $\langle\langle a \cdot b \rangle\rangle$ is a solution for the first equation and $\langle\langle b \cdot a \rangle\rangle$ is a solution for the second one.

We introduce a third variable,

$$X \quad = \quad \langle\langle\, a \cdot b \,\rangle\rangle \;\parallel\; \langle\langle\, b \cdot a \,\rangle\rangle \;.$$

In Calculation 8.3.2 we give a recursive calculation for X to come to a guarded recursive specification.

Calculation 8.3.2

$$
\begin{array}{ll}
 & X \\
\stackrel{\text{Def. } X}{=} & \langle\langle a \cdot b \rangle\rangle \parallel \langle\langle b \cdot a \rangle\rangle \\
\stackrel{M1}{=} & \langle\langle a \cdot b \rangle\rangle \lfloor\!\lfloor \langle\langle b \cdot a \rangle\rangle + \langle\langle b \cdot a \rangle\rangle \lfloor\!\lfloor \langle\langle a \cdot b \rangle\rangle \\
\stackrel{TR1,TR11}{=} & a_0 \cdot \langle\langle a \cdot b, \{a\}, b \rangle\rangle \lfloor\!\lfloor \langle\langle b \cdot a \rangle\rangle + b_0 \cdot \langle\langle b \cdot a, \{b\}, a \rangle\rangle \lfloor\!\lfloor \langle\langle a \cdot b \rangle\rangle \\
\stackrel{M8}{=} & a_0 \cdot (\langle\langle a \cdot b, \{a\}, b \rangle\rangle \parallel [\langle\langle b \cdot a \rangle\rangle]_a) + b_0 \cdot (\langle\langle b \cdot a, \{b\}, a \rangle\rangle \parallel [\langle\langle a \cdot b \rangle\rangle]_b) \\
\stackrel{\text{Def. } V,W}{=} & a_0 \cdot V + b_0 \cdot W
\end{array}
$$

We introduced two new variables in the final step of calculation 8.3.2:

$$
\begin{array}{rcl}
V & = & \langle\langle a \cdot b, \{a\}, b \rangle\rangle \parallel [\langle\langle b \cdot a \rangle\rangle]_a \\
W & = & \langle\langle b \cdot a, \{b\}, a \rangle\rangle \parallel [\langle\langle a \cdot b \rangle\rangle]_b
\end{array}
$$

We calculate guarded recursive specifications for V and W below. Like Y and Z, V and W are symmetric with respect to a and b.

Calculation 8.3.3

$$
\begin{array}{ll}
 & V \\
\stackrel{\text{Def. } V}{=} & \langle\langle a \cdot b, \{a\}, b \rangle\rangle \parallel [\langle\langle b \cdot a \rangle\rangle]_a \\
\stackrel{M1}{=} & \langle\langle a \cdot b, \{a\}, b \rangle\rangle \lfloor\!\lfloor [\langle\langle b \cdot a \rangle\rangle]_a + [\langle\langle b \cdot a \rangle\rangle]_a \lfloor\!\lfloor \langle\langle a \cdot b, \{a\}, b \rangle\rangle \\
\stackrel{\text{Calc. } 8.3.4}{=} & b_0 \cdot (\mathcal{C}_{\{a,b\}} \cdot Z + \mathcal{R}_{\{a,b\}} \cdot X) + [\langle\langle b \cdot a \rangle\rangle]_a \lfloor\!\lfloor \langle\langle a \cdot b, \{a\}, b \rangle\rangle \\
\stackrel{\text{Calc. } 8.3.5}{=} & b_0 \cdot (\mathcal{C}_{\{a,b\}} \cdot Z + \mathcal{R}_{\{a,b\}} \cdot X) + b_0 \cdot (\mathcal{R}_{\{b\}} \cdot V + \mathcal{R}_{\{a\}} \cdot W)
\end{array}
$$

The calculation for W is similar. As a result, we get five equations:

$$
\begin{array}{rcl}
X & = & a_0 \cdot V + b_0 \cdot W \\
V & = & b_0 \cdot (\mathcal{C}_{\{a,b\}} \cdot Z + \mathcal{R}_{\{a,b\}} \cdot X) + b_0 \cdot (\mathcal{R}_{\{b\}} \cdot V + \mathcal{R}_{\{a\}} \cdot W) \\
W & = & a_0 \cdot (\mathcal{C}_{\{a,b\}} \cdot Y + \mathcal{R}_{\{a,b\}} \cdot X) + a_0 \cdot (\mathcal{R}_{\{a\}} \cdot W + \mathcal{R}_{\{b\}} \cdot V) \\
Y & = & a_0 \cdot (b_0 \cdot (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot Y) + \mathcal{R}_{\{a\}} \cdot Y) \\
Z & = & b_0 \cdot (a_0 \cdot (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot Z) + \mathcal{R}_{\{b\}} \cdot Z)
\end{array}
$$

By labelling the nodes of the left-hand process in Figure 8.2, it can be easily seen that the calculation matches the process graph as expected. We show the labelling in Figure 8.4. As can be seen, exactly those nodes to which rollback transitions are possible occur as recursive equations.

In Calculation 8.3.3 we made use of two more calculations, Calculations 8.3.4 and 8.3.5. To complete this example we give these two calculations, which depend on two theorems.

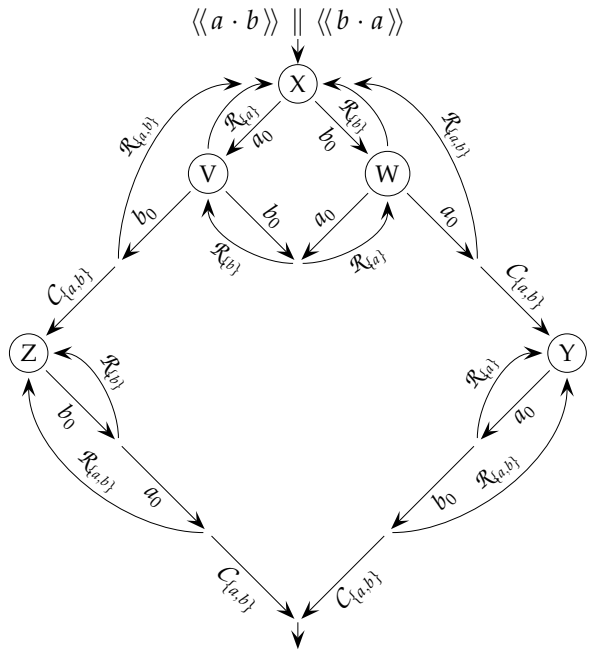


Figure 8.4: Calculated process graph of $\langle\langle a \cdot b \rangle\rangle \parallel \langle\langle b \cdot a \rangle\rangle$.

Calculation 8.3.4

$$
\begin{array}{ll}
 & \langle\langle a \cdot b, \{a\}, b \rangle\rangle \mathbin{\|\!\lfloor} [\langle\langle b \cdot a \rangle\rangle]_a \\
\overset{TR12}{=} & b_0 \cdot \langle\langle a \cdot b, \{a, b\}, \varepsilon \rangle\rangle \mathbin{\|\!\lfloor} [\langle\langle b \cdot a \rangle\rangle]_a \\
\overset{M8}{=} & b_0 \cdot (\langle\langle a \cdot b, \{a, b\}, \varepsilon \rangle\rangle \parallel [[\langle\langle b \cdot a \rangle\rangle]_a]_b) \\
\overset{TR5}{=} & b_0 \cdot ((\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle) \parallel [[\langle\langle b \cdot a \rangle\rangle]_a]_b) \\
\overset{M1}{=} & b_0 \cdot (\ (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle) \mathbin{\|\!\lfloor} [[\langle\langle b \cdot a \rangle\rangle]_a]_b + \\
 & \qquad [[\langle\langle b \cdot a \rangle\rangle]_a]_b \mathbin{\|\!\lfloor} (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle)\ ) \\
\overset{TR1,TR11,L5,L4}{=} & b_0 \cdot (\ (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle) \mathbin{\|\!\lfloor} [[\langle\langle b \cdot a \rangle\rangle]_a]_b + \\
 & \qquad b_1 \cdot [[\langle\langle b \cdot a, \{b\}, a \rangle\rangle]_a]_b \mathbin{\|\!\lfloor} (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle)\ ) \\
\overset{M9,A6}{=} & b_0 \cdot ((\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle) \mathbin{\|\!\lfloor} [[\langle\langle b \cdot a \rangle\rangle]_a]_b) \\
\overset{M11}{=} & b_0 \cdot (\mathcal{C}_{\{a,b\}} \mathbin{\|\!\lfloor} [[\langle\langle b \cdot a \rangle\rangle]_a]_b + \mathcal{R}_{\{a,b\}} \cdot \langle\langle a \cdot b \rangle\rangle \mathbin{\|\!\lfloor} [[\langle\langle b \cdot a \rangle\rangle]_a]_b) \\
\overset{M7}{=} & b_0 \cdot (\ \mathcal{C}_{\{a,b\}} \cdot (\varepsilon \parallel \lfloor [[\langle\langle b \cdot a \rangle\rangle]_a]_b \rfloor_{\{a,b\}}) + \\
 & \qquad \mathcal{R}_{\{a,b\}} \cdot (\langle\langle a \cdot b \rangle\rangle \parallel \lfloor [[\langle\langle b \cdot a \rangle\rangle]_a]_b \rfloor_{\{a,b\}})\ ) \\
\overset{Th.\ 8.3.7}{=} & b_0 \cdot (\mathcal{C}_{\{a,b\}} \cdot (\varepsilon \parallel \langle\langle b \cdot a \rangle\rangle) + \mathcal{R}_{\{a,b\}} \cdot (\langle\langle a \cdot b \rangle\rangle \parallel \langle\langle b \cdot a \rangle\rangle)) \\
\overset{Calc.\ 8.3.1, Th.\ 8.3.9}{=} & b_0 \cdot (\mathcal{C}_{\{a,b\}} \cdot \langle\langle b \cdot a \rangle\rangle + \mathcal{R}_{\{a,b\}} \cdot (\langle\langle a \cdot b \rangle\rangle \parallel \langle\langle b \cdot a \rangle\rangle)) \\
\overset{Def.\ Z,X}{=} & b_0 \cdot (\mathcal{C}_{\{a,b\}} \cdot Z + \mathcal{R}_{\{a,b\}} \cdot X)
\end{array}
$$

Calculation 8.3.5

$$
\begin{array}{ll}
 & [\langle\langle b \cdot a\rangle\rangle]_a \mathbin{\|\!\lfloor} \langle\langle a \cdot b, \{a\}, b\rangle\rangle \\
\overset{TR1,TR11}{=} & [b_0 \cdot \langle\langle b \cdot a, \{b\}, a\rangle\rangle]_a \mathbin{\|\!\lfloor} \langle\langle a \cdot b, \{a\}, b\rangle\rangle \\
\overset{L5,M8}{=} & b_0 \cdot ([\langle\langle b \cdot a, \{b\}, a\rangle\rangle]_a \mathbin{\|} [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b) \\
\overset{M11}{=} & b_0 \cdot (\ [\langle\langle b \cdot a, \{b\}, a\rangle\rangle]_a \mathbin{\|\!\lfloor} [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b + \\
 & \qquad [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b \mathbin{\|\!\lfloor} [\langle\langle b \cdot a, \{b\}, a\rangle\rangle]_a\ ) \\
\overset{Calc.\ 8.3.6}{=} & b_0 \cdot (\mathcal{R}_{\{b\}} \cdot ([\langle\langle b \cdot a\rangle\rangle]_a \mathbin{\|} \langle\langle a \cdot b, \{a\}, b\rangle\rangle) + \mathcal{R}_{\{a\}} \cdot ([\langle\langle a \cdot b\rangle\rangle]_b \mathbin{\|} \langle\langle b \cdot a, \{b\}, a\rangle\rangle)) \\
\overset{Def.\ V,W}{=} & b_0 \cdot (\mathcal{R}_{\{b\}} \cdot V + \mathcal{R}_{\{a\}} \cdot W)
\end{array}
$$

In its fourth step, Calculation 8.3.5 uses Calculation 8.3.6.

Calculation 8.3.6

$$
\begin{array}{ll}
 & [\langle\langle b \cdot a, \{b\}, a\rangle\rangle]_a \mathbin{\|\!\lfloor} [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b \\
\overset{TR12}{=} & [a_0 \cdot \langle\langle b \cdot a, \{a, b\}, \varepsilon\rangle\rangle + \mathcal{R}_{\{b\}} \cdot \langle\langle b \cdot a\rangle\rangle]_a \mathbin{\|\!\lfloor} [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b \\
\overset{TR5}{=} & [a_0 \cdot (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot \langle\langle b \cdot a\rangle\rangle) + \mathcal{R}_{\{b\}} \cdot \langle\langle b \cdot a\rangle\rangle]_a \mathbin{\|\!\lfloor} [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b \\
\overset{L7,L4,L3}{=} & (a_1 \cdot (\mathcal{C}_{\{a,b\}} + \mathcal{R}_{\{a,b\}} \cdot [\langle\langle b \cdot a\rangle\rangle]_a) + \mathcal{R}_{\{b\}} \cdot [\langle\langle b \cdot a\rangle\rangle]_a) \mathbin{\|\!\lfloor} [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b \\
\overset{M11,M9,A6}{=} & (\mathcal{R}_{\{b\}} \cdot [\langle\langle b \cdot a\rangle\rangle]_a) \mathbin{\|\!\lfloor} [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b \\
\overset{M7}{=} & \mathcal{R}_{\{b\}} \cdot ([\langle\langle b \cdot a\rangle\rangle]_a \mathbin{\|} \lfloor [\langle\langle a \cdot b, \{a\}, b\rangle\rangle]_b \rfloor_{\{b\}}) \\
\overset{Th.\ 8.3.7}{=} & \mathcal{R}_{\{b\}} \cdot ([\langle\langle b \cdot a\rangle\rangle]_a \mathbin{\|} \langle\langle a \cdot b, \{a\}, b\rangle\rangle)
\end{array}
$$

The final step in Calculation 8.3.6 makes use of Theorem 8.3.7, which states that locking an action once and immediately unlocking it results in the action before applying this locking and unlocking. Note that in Theorem 8.3.7 we use superscript notation for the indexing of elements to prevent conflicts with the subscript notation of lock counters.

Theorem 8.3.7 *Let $A \subseteq \mathbb{A}$ such that $|A| = M$ and $A = \{a^0, a^1, \ldots, a^{M-1}\}$. Then for all processes x,*

$$\text{PAtrans} \vdash \quad \lfloor [\ldots [x]_{a^0} \ldots]_{a^{M-1}} \rfloor_A \ = \ x\ .$$

Proof We proof this theorem using induction to the structure of x.

- $x = \delta$. Then, using L1 and UL1 it can be easily seen that $\lfloor [\ldots [x]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = \lfloor [\ldots [\delta]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = \delta = x$.
- $x = \varepsilon$. Similar to the previous case, using L2 and UL2.
- $x = \mathcal{U}_B \cdot y$ and $\lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = y$. Using L3, UL3 and induction, we get $\lfloor [\ldots [x]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = \lfloor [\ldots [\mathcal{U}_B \cdot y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = \mathcal{U}_B \cdot \lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = \mathcal{U}_B \cdot y = x$.

- $x = b_n \cdot y$ and $\lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = y$.
  - if $b \in A$, then there is exactly one a^m such that $b = a^m$. Then,

$$
\begin{array}{ll}
 & \lfloor [\ldots [x]_{a^0} \ldots]_{a^{M-1}} \rfloor_A \\
= & \{ \ x = b_n \cdot y \ \} \\
 & \lfloor [\ldots [b_n \cdot y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A \\
= & \{ \ \text{L5}, \forall 0 \leq j < m \ b \neq a^j \ \} \\
 & \lfloor [\ldots [b_n \cdot [\ldots [y]_{a^0} \ldots]_{a^{m-1}}]_{a^m} \ldots]_{a^{M-1}} \rfloor_A \\
= & \{ \ \text{L4}, b = a^m \ \} \\
 & \lfloor [\ldots [b_{n+1} \cdot [\ldots [y]_{a^0} \ldots]_{a^m}]_{a^{m+1}} \ldots]_{a^{M-1}} \rfloor_A \\
= & \{ \ \text{L5}, \forall m < j < M \ b \neq a^j \ \} \\
 & \lfloor b_{n+1} \cdot [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A \\
= & \{ \ \text{UL4}, b \in A \text{ and } n+1 > 0 \ \} \\
 & b_n \cdot \lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A \\
= & \{ \ \text{induction}, \lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = y \ \} \\
 & b_n \cdot y \\
= & \{ \ x = b_n \cdot y \ \} \\
 & x
\end{array}
$$

  - if $b \notin A$, then $\forall 0 \leq m < M \ b \neq a^m$ and as a result this case is similar to case $x = \mathcal{U}_B \cdot y$, using L6 and UL6 instead of L5 and UL5 instead of L3 and UL3.
- $x = b \cdot y$ and $\lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = y$. Similar to case $x = \mathcal{U}_B \cdot y$, using L6 and UL6 instead of L3 and UL3.
- $x = y + z$, $\lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = y$ and $\lfloor [\ldots [z]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = z$. Using L7, UL7 and induction, we get $\lfloor [\ldots [x]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = \lfloor [\ldots [y+z]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = \lfloor [\ldots [y]_{a^0} \ldots]_{a^{M-1}} \rfloor_A + \lfloor [\ldots [z]_{a^0} \ldots]_{a^{M-1}} \rfloor_A = y + z = x$.

□

In Theorem 8.3.9 we state that putting the empty process in parallel to another process allows us to drop the empty process in some cases. In contrast to most process algebras, the axiom $x = x \parallel \varepsilon$ does not hold for all PAtrans processes x, viz. it does *not* hold if (a subprocess of) x is locked. In that case axiom M9 can be applied resulting in a state of deadlock. Therefore we have a restricted theorem stating that if a process in which no actions are locked is put in parallel to the empty process, then this parallelism can be eliminated.

Definition 8.3.8 *Let x be a process in PAtrans. Then $p(x)$ is the predicate that no action in x is locked:*

$$\begin{array}{rcl} p(\delta) & \equiv & \text{true} \\ p(\varepsilon) & \equiv & \text{true} \\ p(\mathbf{a} \cdot x) & \equiv & \neg\exists a{\in}\mathbb{A}\exists n{>}0\ \mathbf{a} = a_n \quad \wedge \quad p(x) \\ p(x+y) & \equiv & p(x) \quad \wedge \quad p(y) \end{array}$$

Theorem 8.3.9 *Let x be a process in* PAtrans *such that no action in x is locked, i.e. $p(x)$ holds for p as defined in Definition 8.3.8. Then*

$$\varepsilon \parallel x \ = \ x \ .$$

Proof We prove this using induction to the structure of x:

- $x = \delta$. Then, using M1–3 and A6, $\varepsilon \parallel x = \varepsilon \parallel \delta = \varepsilon \mathbin{\lfloor\!\lfloor} \delta + \delta \mathbin{\lfloor\!\lfloor} \varepsilon = \delta + \delta = \delta = x$.
- $x = \varepsilon$. Then, using M1, A3 and M4, $\varepsilon \parallel x = \varepsilon \parallel \varepsilon = \varepsilon \mathbin{\lfloor\!\lfloor} \varepsilon + \varepsilon \mathbin{\lfloor\!\lfloor} \varepsilon = \varepsilon \mathbin{\lfloor\!\lfloor} \varepsilon = \varepsilon = x$.
- $x = \mathcal{U}_A \cdot y$ for $A \subseteq \mathbb{A}$ and since $p(y)$ holds, $\varepsilon \parallel y = y$. Then using M1, M5, M7, UL2 and induction, we get $\varepsilon \parallel x = \varepsilon \parallel (\mathcal{U}_A \cdot y) = \varepsilon \mathbin{\lfloor\!\lfloor} (\mathcal{U}_A \cdot y) + (\mathcal{U}_A \cdot y) \mathbin{\lfloor\!\lfloor} \varepsilon = \delta + \mathcal{U}_A \cdot (y \parallel \lfloor \varepsilon \rfloor_A) = \mathcal{U}_A \cdot (y \parallel \varepsilon) = \mathcal{U}_A \cdot y = x$.
- $x = a_n \cdot y$ for $a \in \mathbb{A}$ and since $p(x)$ holds and thus $p(y)$ holds, $n = 0$ and $\varepsilon \parallel y = y$. Then using M1, M5, M8, UL2, $n = 0$ and induction, we get $\varepsilon \parallel x = \varepsilon \parallel (a_n \cdot y) = \varepsilon \mathbin{\lfloor\!\lfloor} (a_0 \cdot y) + (a_0 \cdot y) \mathbin{\lfloor\!\lfloor} \varepsilon = \delta + a_0 \cdot (y \parallel [\varepsilon]_a) = a_0 \cdot (y \parallel \varepsilon) = a_0 \cdot y = a_n \cdot y = x$.
- $x = a \cdot y$ for $a \in \mathbb{A}$, and since $p(x)$ holds and thus $p(y)$ holds, $\varepsilon \parallel y = y$. Then using M1, M5, M10 and induction, we get $\varepsilon \parallel x = \varepsilon \parallel (a \cdot y) = \varepsilon \mathbin{\lfloor\!\lfloor} (a \cdot y) + (a \cdot y) \mathbin{\lfloor\!\lfloor} \varepsilon = \delta + a \cdot (y \parallel \varepsilon) = a \cdot y = x$.
- $x = y + z$ and since $p(x)$ holds and thus $p(y)$ and $p(z)$ hold, $\varepsilon \parallel y = y$ and $\varepsilon \parallel z = z$. Then, using M1, M6, M11 and induction, we get $\varepsilon \parallel x = \varepsilon \parallel (y + z) = \varepsilon \mathbin{\lfloor\!\lfloor} (y + z) + (y + z) \mathbin{\lfloor\!\lfloor} \varepsilon = \varepsilon \mathbin{\lfloor\!\lfloor} y + \varepsilon \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} \varepsilon + z \mathbin{\lfloor\!\lfloor} \varepsilon = \varepsilon \parallel y + \varepsilon \parallel z = y + z = x$.

□

As can be seen in Figure 8.4 and by looking at the recursive specification calculated in Calculations 8.3.1 to 8.3.6, process $\langle\langle a \cdot b \rangle\rangle \parallel \langle\langle b \cdot a \rangle\rangle$ can reach a state in which a rollback is unavoidable. In Calculation 8.3.5, detailed information is given. If no rollback is possible, this leads to a state of deadlock. In the next section we prove, amongst other things, that the transactional operator does not introduce deadlocks without having the possibility to roll back.

8.4 Properties of Process Algebra with Transactions

In this section we prove some properties of PAtrans. First of all, we prove that the transactional operator in combination with the merge operator does not introduce deadlock situations in which no rollback is possible. We prove this since Axioms M5 and M9 introduce possibly unwanted deadlocks.

Lemma 8.4.1 *After execution of an action in $\mathbb{A}$ or lockable action in $\mathbb{L}$ from within a transaction, there is always an alternative to rollback the transactions.*

Proof We need to prove that for processes x and y, action $a \in \mathbb{A}$, $n \in \mathbb{N}$ and $A \subseteq \mathbb{A}$, after execution of a in $\langle\langle x, A, ay\rangle\rangle$ or a_n in $\langle\langle x, A, a_n y\rangle\rangle$ an alternative to roll back is available.

By looking at axioms TR8–13, execution of an $a \in \mathbb{A}$ and an $a_n \in \mathbb{L}$ are treated similarly, except for the adding of a lock counter at the first occurrence of the execution of a non-lockable action (TR13) and the stripping off of the lock counter when executing a lockable that that has been executed before (TR10). As can be seen, execution of such an action leads to $\langle\langle x, A \cup \{a\}, y\rangle\rangle$. By looking at axioms TR3, TR5, TR7, TR9–10 and TR12–13, it can be seen that if $A \neq \emptyset$, which is the case since at least $a \in A$, we can always do a rollback $\mathcal{R}_A$ to $\langle\langle x\rangle\rangle$, independent of y. □

Theorem 8.4.2 (Absence of Deadlock) *When only making use of the main operators for alternative composition $+$, sequential composition $\cdot$, parallel composition $\|$ and transactions $\langle\langle\ \rangle\rangle$ for specifying processes, no deadlock is introduced.*

Proof By looking at the axioms, it can be seen that new deadlocks are only introduced in M5 and M9. All other deadlocks (δ) appear on both the left-hand and right-hand side of the equations. So $\varepsilon \lfloor\!\lfloor \mathbf{a}x = \delta$ and for $n > 0$, $a_n x \lfloor\!\lfloor y = \delta$. We prove that both processes $\varepsilon \lfloor\!\lfloor \mathbf{a}x$ and $a_n x \lfloor\!\lfloor y$ for $n > 0$ only occur next to an alternative for rolling back, and thus can be eliminated using axiom A6.

- $\varepsilon \lfloor\!\lfloor \mathbf{a}x$: Since we assumed that we only make use of the main operators, this process must be a subprocess of process $\varepsilon \| \mathbf{a}x$ and thus an alternative occurs, viz. $\mathbf{a}x \lfloor\!\lfloor \varepsilon$ (M1). We still have a deadlock situation if $\mathbf{a} = a_n$ for $n > 0$ (M9). We prove below that for $a_n x \lfloor\!\lfloor y$, an alternative exists to roll back, so choosing $y = \varepsilon$ we also prove it for this case.
- $a_n x \lfloor\!\lfloor y$ for $n > 0$: Again, since we only make use of the main operators, $a_n x \lfloor\!\lfloor y$ occurs as a subprocess of $a_n x \| y$. So alternative $y \lfloor\!\lfloor a_n x$ exists. Since $n > 0$, $a_n x$

must have been locked (L4). So a x' must exist such that $a_n x = [a_{n-1}x']_a$. By looking at the axioms, it can be seen that the locking operator is only introduced in axiom M8, so an action a_0 must exist such that $a_0(y \parallel [a_{n-1}x']_a)$. Since after execution of a_0, which is locking and therefore *must* come from within a transactions, a rollback can always take place (using Lemma 8.4.1), it must be the case that $a_0(y \parallel [a_{n-1}x']_a + R_A\langle\langle x''\rangle\rangle)$ for some $A \subseteq \mathbb{A}$ and process x''. So process $y \parallel [a_{n-1}x']_a$ and thus $y \parallel a_n x$ only occurs as alternative to a rollback.

□

Next, we give a soundness proof, i.e. we prove that the set of closed PAtrans terms modulo bisimulation equivalence, $T(\text{PAtrans})/\leftrightarrows$, is a model for PAtrans.

Definition 8.4.3 (Bisimulation for PAtrans) *Let T(PAtrans) be the term deduction system induced by deduction rules 1–36 as shown in Tables 5.4, 8.2, 8.4 and 8.6. Bisimulation for* PAtrans *is then defined as follows: a symmetric binary relation R on closed terms in* PAtrans *is a bisimulation if and only if the following* transfer conditions *hold for all closed* PAtrans *terms p and q:*

1. *if $R(p,q)$ and $T(\text{PAtrans}) \models p\downarrow$, then $T(\text{PAtrans}) \models q\downarrow$.*
2. *if $R(p,q)$ and $T(\text{PAtrans}) \models p \xrightarrow{\mathbf{a}} p'$, where $\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$, then there exists a process term q' such that $T(\text{PAtrans}) \models q \xrightarrow{\mathbf{a}} q'$ and $R(p',q')$.*

Two closed PAtrans *terms p and q are bisimular, notation $p \leftrightarrows q$, if there exists a bisimulation relation R such that $R(p,q)$.*

Using bisimulation, we can now construct a model for the axioms of PAtrans. In order to do this, we first need to know that bisimulation is a congruence with respect to all operators.

Lemma 8.4.4 (Bisimulation is a congruence) *Let T(PAtrans) be the term deduction system induced by the deduction rules 1–36 as shown in Tables 5.4, 8.2, 8.4 and 8.6. Then bisimulation equivalence is a congruence on the set of closed* PAtrans *terms.*

Proof It can be easily seen that the operational semantics given in Tables 5.4, 8.2, 8.4 and 8.6 is in *path format* [BV93]. Since it is proved that if a term deduction system is in path format, bisimulation is a congruence [BV95] (based on [BV93, Fok94]), this proves this lemma. □

Definition 8.4.5 (Bisimulation model for PAtrans) *The bisimulation model for the process algebra with transactions (*PAtrans*) is constructed by taking the equivalence classes of the set of all closed* PAtrans *terms with respect to bisimulation equivalence. As bisimulation is a congruence, the operators can be pointwise defined on the equivalence classes.*

Theorem 8.4.6 (Soundness of PAtrans) *The set of closed* PAtrans *terms modulo bisimulation equivalence, $T(\text{PAtrans})/\leftrightarrows$, is a model for* PAtrans.

Proof We prove this theorem by proving that each axiom is sound, i.e., by proving that for all closed instantiations of the axiom, both sides of the axiom correspond to the same element of the bisimulation model. This proof outline is taken from [Ver97, BV95]. The proof itself can be found in Section B.1 on page 223. □

We now turn to completeness. Since we deal with a (restricted) form of recursion, proving completeness requires a detailed analysis of the process terms (see e.g. [BFP01]). Apart from that, formally specifying transactional behaviour is concerned with parallelism, which makes the model even more complex. Since this proof of completeness goes beyond the scope of the research project, we leave it for future work.

By proving that all PAtrans terms can be eliminated to a term in a $\text{BPA}_{\delta\varepsilon}$ with guarded linear recursion ($\text{BPA}_{\delta\varepsilon}$rec), we prove that the expressiveness of the process theory has not increased. Since the same set of processes can be defined without using PAtrans operators, the transactional operators introduced can be considered a syntactic extension for simplifying the notation of transactional processes.

If we have a closer look at the bisimulation model for PAtrans, we see that all elements in this model are *regular processes*, i.e. processes having finitely many states and finitely many transitions. A regular process is represented by an equivalence class of finite transition systems modulo bisimulation. In [BK84a] it is shown that the set of all finite transition systems modulo bisimulation is a model for BPA with unique solutions for all linear recursive specifications.

Theorem 8.4.7 (Elimination to $\text{BPA}_{\delta\varepsilon}$rec) *For every closed* PAtrans *term t there exists a linear recursive specification E over* $\text{BPA}_{\delta\varepsilon}$ *such that t is a solution of E.*

Proof This theorem is proved by induction on the general structure of t. We sketch the proof for $t \equiv \langle\langle t_1, A, t_2\rangle\rangle$ for closed PAtrans terms t_1 and t_2 and $A \subseteq \mathbb{A}$, where t_2 is a subprocess of t_1 and A a set of actions from t_1. We give a set of linear recursive equations E containing variables of the form $X_{t_2}^{t_1,A}$, and prove by induction on the structure of t_2 that for all terms $\langle\langle t_1, A, t_2\rangle\rangle$ there exists a $\text{BPA}_{\delta\varepsilon}$rec term s such that

$T(\text{PAtrans}) \models s = \langle\langle t_1, A, t_2 \rangle\rangle$. Let E be defined as follows:

$$
\begin{array}{llll}
E = \{ & X_{\delta}^{t,\emptyset} = \delta, & X_{\delta}^{t,A} = \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
& X_{\varepsilon}^{t,\emptyset} = \mathcal{C}_{\emptyset}, & X_{\varepsilon}^{t,A} = \mathcal{C}_A + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
& X_{\mathcal{U}_B \cdot t_1}^{t,\emptyset} = \mathcal{U}_{\emptyset} \cdot X_{t_1}^{t,\emptyset}, & X_{\mathcal{U}_B \cdot t_1}^{t,A} = \mathcal{U}_{\emptyset} \cdot X_{t_1}^{t,A} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
& X_{c_n \cdot t_1}^{t,\emptyset} = c_n \cdot X_t^{t,\{c\}}, & X_{b_n \cdot t_1}^{t,A} = b_n \cdot X_{t_1}^{t,A \cup \{b\}} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
& X_{a_n \cdot t_1}^{t,A} = a \cdot X_{t_1}^{t,A} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, & X_{c \cdot t_1}^{t,\emptyset} = c_0 \cdot X_{t_1}^{t,\{c\}}, \\
& X_{b \cdot t_1}^{t,A} = b_0 \cdot X_{t_1}^{t,A \cup \{b\}} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, & X_{a \cdot t_1}^{t,A} = a \cdot X_{t_1}^{t,A} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
& X_{t_1+t_2}^{t,B} = X_{t_1}^{t,B} + X_{t_2}^{t,B} & \\
| & A \subseteq \mathbb{A},\ A \neq \emptyset,\ B \subseteq \mathbb{A},\ n \in \mathbb{N},\ n > 0,\ a \in A,\ b \in \mathbb{A} \setminus A,\ c \in \mathbb{A} \quad \}
\end{array}
$$

Since A is finite (all actions in A come from t_1) and t_2 is a subprocess of t_1, E is finite. As can be seen by comparing axioms TR2–14 with the recursive equations in E, $\langle\langle t_1, A, t_2 \rangle\rangle$ satisfies the equation for $X_{t_2}^{t_1,A}$ in E. The full proof can be found in Section B.2 on page 232. □

8.5 Combining Transactions and States

As mentioned in Section 8.1, we abstracted from read access to variables and the actual data changes. In this section we explain how we can combine transactions with the concept of states as introduced in Chapter 6 to come closer to real-life transactional behaviour.

8.5.1 Introduction

Up till now, we only took write actions into account. However, both read actions and internal actions might also be needed when giving real-life examples. Therefore, we need to draw a distinction between write actions and other (i.e. read or internal) actions. In former sections we assumed that execution of action a meant a write action to (shared) variable a. We can explicitly indicate whether an action is a write action to a variable. The set of actions A in $\langle\langle x, A, y \rangle\rangle$ is then redefined such that it contains only updated variables.

Although this is a major change in the notation and the way we deal with variables, this does not influence the axioms drastically. If we consider first degree isolated transactions, there are dependencies between write actions only. (Degrees of isolation are discussed in Section 8.6.) So both read actions and internal actions are not influenced by the locking operator and therefore only write actions should have their lock counter increased. We make this distinction between different kinds of actions by adapting the conditions in the axioms and operational rules.

Since only (more) conditions on the format of the actions are added, this extension does not influence the soundness nor the elimination to $\mathrm{BPA}_{\delta\varepsilon}\mathrm{rec}$ substantially with respect to the operators introduced in this chapter.

As introduced in Chapter 6, we have a formalism for specifying states and state changes. In this section we combine the formalism for specifying transactions with this formalism. Both formalisms are concerned with valuation changes of variables. Since we need a mechanism for the handling of state changes and doing calculations using state changes we introduce valuations sets, together with some operators on them.

Definition 8.5.1 *A* valuation set *V is a set containing valuations where valuations are as defined in Definition 6.2.1. All valuations in a valuation set have a unique identifier:*

$$\forall\, (i \mapsto c : T) \in V\ \forall\, (i' \mapsto c' : T') \in V \setminus \{i \mapsto c : T\} \quad i \neq i'$$

The set of all possible valuations is denoted by $\mathbb{V}$.

We make use of some operators on sets of valuations, which we introduce below. First of all, we use function vars for determining the variable identifiers occurring in a valuation set.

Definition 8.5.2 *Let $V \subseteq \mathbb{V}$ be a valuation set. The set of identifiers in V,* $\mathrm{vars}(V)$*, is then defined by*

$$\mathrm{vars}(V) \quad = \quad \{i \in \mathbb{D} \mid \exists T{\in}\mathbb{T}\, \exists c{\in}T\ (i \mapsto c : T) \in V\}$$

We also introduce some binary infix operators on valuation sets, to which we add a small bullet ($\bullet$). The first reason is for identifying them as being operators on valuation sets. Apart from this identification, the bullet makes the operators asymmetric, which is preferred since none of the operators is commutative.

Definition 8.5.3 *Let $V \subseteq \mathbb{V}$ and $W \subseteq \mathbb{V}$ be valuation sets. The* addition *of W to V, $V \bullet\!\!\cup W$ contains all valuations in V and those valuations in W that map variables not occurring in V:*

$$V \bullet\!\!\cup W \quad = \quad V \cup \{(i \mapsto c : T) \in W \mid i \notin \mathrm{vars}(V)\}$$

Definition 8.5.4 *Let $V \subseteq \mathbb{V}$ and $W \subseteq \mathbb{V}$ be valuation sets. The* identifier intersection *of these sets, $V \bullet\!\!\cap W$, is a valuation set containing all valuations in V for which the variables are also mapped in W:*

$$V \bullet\!\!\cap W \quad = \quad \{(i \mapsto c : T) \in V \mid i \in \mathrm{vars}(W)\}$$

Definition 8.5.5 *Let $V \subseteq \mathbb{V}$ and $W \subseteq \mathbb{V}$ be valuation sets. The set V without W, $V \setminus\!\!\bullet\, W$, is a valuation set containing those valuations from V where the variable is* not *mapped in W:*

$$V \setminus\!\!\bullet\, W \quad = \quad \{(i \mapsto c : T) \in V \mid i \notin \mathrm{vars}(W)\}$$

Apart from the operators on valuation sets, we introduce some additional operators on valuation stacks (see Definition 6.2.3). First of all, we make use of a function (vars) to determine the variable identifiers occurring in a valuation stack.

Definition 8.5.6 *Let σ be a valuation stack. The set of identifiers in σ,* $\mathrm{vars}(\sigma)$*, is then defined by*

$$\begin{array}{rcl} \mathrm{vars}(\lambda) & = & \emptyset \\ \mathrm{vars}((i \mapsto c : T) :: \sigma) & = & \{i\} \cup \mathrm{vars}(\sigma) \end{array}$$

In Definition 6.2.6 we introduced the substitution operator on valuation stacks. This function is extended to substitutions of valuation sets.

Definition 8.5.7 *Let σ be a valuation stack and V be a valuation set. Then, the substitutions of valuations in σ by valuations V, $\sigma[V]$, is defined by*

$$\begin{array}{rcl} \lambda[V] & = & \lambda \\ (i \mapsto c : T) :: \sigma[V] & = & \begin{cases} (i \mapsto c' : T) :: (\sigma[V \setminus\!\!\bullet\, \{i \mapsto c' : T\}]) & \textit{if } \exists c' \ (i \mapsto c' : T) \in V \\ (i \mapsto c : T) :: (\sigma[V]) & \textit{otherwise} \end{cases} \end{array}$$

Now that we have introduced valuation sets together with operators on them, we have a look at the deduction rules for the operators introduced in Section 8.2. The deduction rules nicely show all elements that should be thought of when replacing the current PAtrans actions with real actions and when adding a state and time component to them. In the remainder of this section we discuss the elements that should be taken into account when adapting the deduction rules in such a way that they become suitable for the combination with states and time.

8.5.2 Rollbacks and Commits

Since a rollback should be able to restore a state which existed before starting a transaction, we should not only keep track of the variables that are updated when executing actions from within transactions, but also of their valuations at the time of their first update within the transaction. This is done by replacing the set of actions A in $\langle\langle x, A, y\rangle\rangle$ with a valuation set V, so $\langle\langle x, A, y\rangle\rangle$ becomes $\langle\langle x, V, y\rangle\rangle$. This set V contains all valuations that are updated by the transaction. Apart from this extension, we

draw a distinction between rolling back a transaction and committing one. If a rollback takes place, the original values, stored in V, are restored and the variables are unlocked. If, on the other hand, a transaction commits, we only unlock the variables, keeping the new valuations as assigned from within the transaction. To achieve this, the A which is added to commit ($\mathcal{C}$) and rollback ($\mathcal{R}$) actions is also replaced by valuation set V.

By looking at the deduction rules for the operators introduced in this chapter, it can be seen that we already draw a distinction between rolling back a transaction (deduction rule 8 in Table 8.2) and committing one (deduction rules 9 and 10 in Table 8.2). So in rule 8 we should restore the original valuations where in rules 9 and 10 the new valuations should be kept. The effect of rolling back or committing a transaction which updated valuations V in state σ at time t, effect($\mathcal{U}_V, \sigma, t$), is therefore defined as follows:

$$\begin{aligned} \text{effect}(\mathcal{R}_V, \sigma, t) &=_{\text{def}} \sigma[V] \\ \text{effect}(\mathcal{C}_V, \sigma, t) &=_{\text{def}} \sigma \end{aligned}$$

As can be seen in deduction rule 8 (in Table 8.2 on page 88), we make use of $A \neq \emptyset$ to check whether an action had already been executed in the current transaction. Since read actions and internal actions do not affect any of the valuations in the state, the set of valuations V in $\langle\langle x, V, y\rangle\rangle$ can be empty, even if actions are executed. Therefore, we make the assumption that we only allow the use of non-auxiliary operators when giving specifications of processes. This assumption, together with the fact that the auxiliary transactional operator ($\langle\langle \_, \_, \_\rangle\rangle$) is only introduced after executing the initial action of the transaction (see deduction rules 9, 11, 12 and 13 in Table 8.2), allows us to drop the premise of this rule. As a result, we get the following deduction rule:

$$\frac{}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow{\mathcal{R}_V} \langle\langle\langle x\rangle\rangle, \sigma[V], t\rangle}$$

Since there are no premises, we call this deduction rule an axiom.

8.5.3 Executing Actions from within a Transaction

As mentioned, we replace the set A in $\langle\langle x, A, y\rangle\rangle$ with a set of valuations V. On page 83, we introduced abbreviation a for a write action to a shared variable a. Furthermore, on page 84 we said that $\langle\langle x, A, y\rangle\rangle$ represents *"transactional process x, which has already executed the set of actions A and still has to execute process y before a commit statement can take place"*. This set of actions A is actually the set of already updated

variables. This is exactly what we store in V: the original valuations which are updated by executed actions from within the transaction. So, when executing an action a, we add all valuations to V which are affected by action a and which are not already contained in V.

Determining the set of updated valuations is an easy task as long as the values of updated variables change. In that case, the set of valuations that are updated by execution of action a in state σ at time t is the difference between the states before and after execution of action a. However, it might be possible that even if valuations keep unchanged, valuations are updated. To give an example we revise the process given on the right-hand side in Figure 8.1 on page 82. We initialise the a with two different values:

$$[\, a \mapsto 0 : \mathbb{N} \,|\, \langle\!\langle a := 0 \cdot a := a + 2 \rangle\!\rangle \;\|\; \langle\!\langle a := 1 \cdot a := a \times 2 \rangle\!\rangle \,]$$

and

$$[\, a \mapsto 1 : \mathbb{N} \,|\, \langle\!\langle a := 0 \cdot a := a + 2 \rangle\!\rangle \;\|\; \langle\!\langle a := 1 \cdot a := a \times 2 \rangle\!\rangle \,] \,.$$

In the first case with initial value 0, the effect of $a := 0$ on state $a \mapsto 0 : \mathbb{N}$ is $a \mapsto 0 : \mathbb{N}$. So the difference is empty. However, locking should take place since otherwise we can get the unwanted sequence

$$a := 0 \;\cdot\; a := 1 \;\cdot\; a := a \times 2 \;\cdot\; a := a + 2$$

which results in state $a \mapsto 4 : \mathbb{N}$. The same holds when initialising with 1. In that case we can also get into a state where a maps to 4, viz. after execution of sequence

$$a := 1 \;\cdot\; a := 0 \;\cdot\; a := a + 2 \;\cdot\; a := a \times 2 \;\;.$$

Initialisation with any other value, including $\perp$, results in the intended process as shown in Figure 8.1.

To solve this problem, we define an updates predicate on variable identifiers and action, which states whether the action *might update* the valuation of the variable:

$$\begin{aligned} \text{updates} \quad &: \quad \mathbb{ID} \times \mathbb{A} \to \mathbb{B} \\ \text{updates}(i, a) \quad &= \quad \exists \sigma \in \mathbb{S}\; \exists t \in \mathfrak{T} \quad i(\sigma, t) \neq i(\text{effect}(i, \sigma, t), t) \end{aligned}$$

Having the updates functions, we can specify a function U for determining the valuations in a state that are (possibly) updated by an action:

$$\begin{aligned} \mathrm{U} \quad &: \quad \mathbb{A} \times \mathbb{S} \times \mathfrak{T} \to \mathcal{P}(\mathbb{V}) \\ \mathrm{U}(a, \lambda, t) \quad &= \quad \emptyset \\ \mathrm{U}(a, (i \mapsto c : T) :: \sigma, t) \quad &= \quad \begin{cases} \{i \mapsto c : T\} \,\dot{\cup}\, \mathrm{U}(a, \sigma, t) & \text{if updates}(i, a) \\ \mathrm{U}(a, \sigma, t) & \text{otherwise} \end{cases} \end{aligned}$$

So $\mathrm{U}(a, \sigma, t)$ returns the set of valuations in σ that are possibly updated by the action a at time t. We use notations U for the *updated* function since it is the first character of "updated" and its result is a valuation set which corresponds to formerly used valuation sets V and W.

The set of valuations that is updated and where no mapping to the variables is (already) available in V, can be expressed by

$$\mathrm{U}(a, \sigma, t) \,\bullet\!\!\setminus\, V \ .$$

In rules 15–18 in Table 8.2 on page 88, premises $a \in A$ and $a \notin A$ state whether variable a had or had not been updated by an action from within the transaction. We replace these premises by premises which express that either none or at least one valuation is updated by the action which was not updated by actions from within the transaction before. So the premises can be adapted as follows:

$$\begin{array}{lll} a \in A & \text{is replaced by} & \mathrm{U}(a, \sigma, t) \,\bullet\!\!\setminus\, V = \emptyset \\ a \notin A & \text{is replaced by} & \mathrm{U}(a, \sigma, t) \,\bullet\!\!\setminus\, V \neq \emptyset \end{array}$$

Furthermore, instead of adding a to A in rules 12, 13, 15 and 17 , we now have to add the updated valuations to valuation set V:

$$\begin{array}{rll} \{a\} & \text{is replaced by} & \mathrm{U}(a, \sigma, t) \\ A \cup \{a\} & \text{is replaced by} & V \,\bullet\!\!\cup\, \mathrm{U}(a, \sigma, t) \end{array}$$

8.5.4 Locking and Unlocking

We recall the considerations that led to the use of lock counters, as stated on page 86: "If transaction $\langle\langle x, A, y\rangle\rangle$ executes action a for the first time, i.e. $a \notin A$, then a is extended with a lock counter having value 0 and A is extended with a. If the transaction executed an a before, i.e. $a \in A$, no locking counter is added since the transaction already has exclusive rights on action a". As mentioned in the previous section, the check for $a \notin A$ is replaced by $\mathrm{U}(a, \sigma, t) \,\bullet\!\!\setminus\, V \neq \emptyset$, i.e., there is at least one valuation that is updated by action a, which was not yet updated by an action in the transaction. The lock counter contains the number of variables that are locking the action, i.e., the number of times the action is locked because it tries to update a variable that is locked. Apart from that, the locking operator uses the locking action which syntactically represents the updated variable. However, when adding states and time, we should not lock variables identified by the name of the action, but we lock the set of lockable actions which also try to update the newly updated set of valuations $\mathrm{U}(a, \sigma, t) \,\bullet\!\!\setminus\, V$. Therefore, apart from the lock counter, we extend locking actions with

this valuation set. To give an example, we present the adapted deduction rule for rule 17 in Table 8.2:

$$\frac{\langle y, \sigma, t\rangle \xrightarrow{a} \langle y', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) \,\bullet\!\!\setminus\, V \neq \emptyset}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow{a_{0,\mathrm{U}(a,\sigma,t) \,\bullet\!\setminus\, V}} \langle\langle\langle x, V \,\bullet\!\!\cup\, \mathrm{U}(a, \sigma, t), y'\rangle\rangle, \sigma', t\rangle}$$

Furthermore, since we draw a distinction between read and write actions by comparing states before and after execution of actions, we need to split deduction rule 13 in Table 8.2 into two rules. We only want write actions to be lockable, so rule 13 should only add a lock counter if the executed action changes the state, that is, if $\mathrm{U}(a, \sigma, t) \neq \emptyset$:

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) = \emptyset}{\langle\langle\langle x\rangle\rangle, \sigma, t\rangle \xrightarrow{a} \langle\langle\langle x, \emptyset, x'\rangle\rangle, \sigma', t\rangle} \qquad \frac{\langle x, \sigma, t\rangle \xrightarrow{a} \langle x', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) \neq \emptyset}{\langle\langle\langle x\rangle\rangle, \sigma, t\rangle \xrightarrow{a_{0,\mathrm{U}(a,\sigma,t)}} \langle\langle\langle x, \mathrm{U}(a, \sigma, t), x'\rangle\rangle, \sigma', t\rangle}$$

Now, we have all ingredients to adapt the deduction rules for the transactional operator given in Table 8.2.

We still need to adapt the rules for the locking and unlocking operators and for the parallel composition. These rules deal with the actual (un)locking of actions. We first have a look at the parallel composition, after which we discuss the locking and unlocking operators in more detail.

As explained in Section 8.2.3, the merge operator introduces the locking and unlocking operator if a locking or unlocking action is executed, respectively (see deduction rules 30, 31, 34 and 35 in Tables 8.6 on page 91). The unlocking actions unlock the variables contained in the valuation set with which it is parameterised (the V in $\mathcal{U}_V$). This set is also added to the unlocking operator, which is explained in more detail below. So we only need to replace the A's in the deduction rules with V's. The locking operator, however, should also be extended with a set of variables, i.e., the set of valuations that are updated by the executed action. Having a one-on-one relation on the actions and the locked variables (as in PAtrans), we could parameterise the locking operator with the action itself. However, since the one-on-one relation no longer holds when adding states and time, we have to extend the locking operator with the newly updated variables, as explained above. This exactly equals the set of valuations added to the locking actions, i.e., the V in $a_{n,V}$. To give an example, we present the adapted deduction rule for rule 31 in Table 8.6:

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{a_{0,V}} \langle x', \sigma', t\rangle}{\langle x \parallel y, \sigma, t\rangle \xrightarrow{a_{0,V}} \langle x' \parallel [y]_V, \sigma', t\rangle, \quad \langle y \parallel x, \sigma, t\rangle \xrightarrow{a_{0,V}} \langle [y]_V \parallel x', \sigma', t\rangle}$$

The locking and unlocking operators increase and decrease the lock counters of lockable actions, respectively. This changing of the lock counter depends on the set of

variables that is added as a parameter to the operators. In PAtrans, the locking operator increases the lock counter of all actions that are equal to the action which is attached to the locking operator. This means that those actions are locked that update the variables added to the locking operator. Since we now add a set of variables (which is actually a set of valuations), we need to check whether the lockable actions update any of the variables included in this set. As we have seen before, the set of variables updated by lockable action $a_{n,V}$ is V. Thus, the test whether the lock counter should be increased is determined by testing for emptiness of the intersection of this valuation set V and parameter W of the locking operator:

$$\begin{array}{lll} a = b & \text{is replaced by} & V \mathbin{\underset{\bullet}{\cap}} W \neq \emptyset \\ a \neq b & \text{is replaced by} & V \mathbin{\underset{\bullet}{\cap}} W = \emptyset \end{array}$$

So if at least one variable from W is updated by lockable action $a_{n,V}$, its lock counter should be increased. We increase the lock counter with the number of variables that cause the locking of the action, i.e. the number of variables that are both updated by the lockable action and the locking action: $|V \mathbin{\underset{\bullet}{\cap}} W|$. As a result, e.g. deduction rule 22 in Table 8.4 is adapted as follows:

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{a_{n,V}} \langle x', \sigma', t\rangle, \quad V \mathbin{\underset{\bullet}{\cap}} W \neq \emptyset}{\langle [x]_W, \sigma, t\rangle \xrightarrow{a_{n+|V \mathbin{\underset{\bullet}{\cap}} W|,V}} \langle [x']_W, \sigma', t\rangle}$$

If no valuations are updated, the lock counter should not increase, as stated in rule 21 in Table 8.4. Since in that case $|V \mathbin{\underset{\bullet}{\cap}} W| = 0$, increasing with $|V \mathbin{\underset{\bullet}{\cap}} W|$ is allowed, we can merge rules 21 and 22 by dropping the premise $V \mathbin{\underset{\bullet}{\cap}} W \neq \emptyset$:

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{a_{n,V}} \langle x', \sigma', t\rangle}{\langle [x]_W, \sigma, t\rangle \xrightarrow{a_{n+|V \mathbin{\underset{\bullet}{\cap}} W|,V}} \langle [x']_W, \sigma', t\rangle}$$

Unlocking of actions is done in a similar way. First of all, we extend the unlocking operator with a valuation set V instead of the A we made use of in PAtrans. For example, deduction rule 30 in Table 8.6 is adapted as follows:

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{\mathcal{U}_V} \langle x', \sigma', t\rangle}{\langle x \parallel y, \sigma, t\rangle \xrightarrow{\mathcal{U}_V} \langle x' \parallel \lfloor y \rfloor_V, \sigma', t\rangle, \quad \langle y \parallel x, \sigma, t\rangle \xrightarrow{\mathcal{U}_V} \langle \lfloor y \rfloor_V \parallel x', \sigma', t\rangle}$$

Furthermore, in PAtrans we tested for $a \in A$, which meant that the variable is contained in the set of variables that should be unlocked. Similarly to the locking operator, the test whether the lock counter should be decreased by the unlocking operator can be determined by testing for emptiness of the intersection of valuation set V and

parameter W of the unlocking operator:

$$\begin{array}{lll} a \in A & \text{is replaced by} & V \underset{\bullet}{\cap} W \neq \emptyset \\ a \notin A & \text{is replaced by} & V \underset{\bullet}{\cap} W = \emptyset \end{array}$$

So a lockable action should be unlocked if the intersection of the updated variables by the lockable action and the set of unlocked variables is not empty. The lock counter is decreased by the number of variables in this intersection. To give an example, we adapt deduction rule 26 in Table 8.4 as follows:

$$\frac{\langle x, \sigma, t\rangle \xrightarrow{a_{n,V}} \langle x', \sigma', t\rangle, \quad (V \underset{\bullet}{\cap} W \neq \emptyset \wedge n > 0)}{\langle \lfloor x \rfloor_W, \sigma, t\rangle \xrightarrow{a_{n-|V \underset{\bullet}{\cap} W|,V}} \langle \lfloor x' \rfloor_W, \sigma', t\rangle}$$

8.5.5 Identifiers and Scoping

Up till now, we only took the deduction rules for the transactional operators into account. However, the deduction rules for the scope operator also need an adaptation to achieve the expected behaviour. This is caused by the fact that we parameterise the locking and unlocking actions with valuation sets. Of course, locking and unlocking of actions can only depend on variables available to the action. Suppose that we do not adapt the deduction rules for the scope operator. Have a look at the following process:

$$\begin{array}{l} [\ n \mapsto 0 : \mathbb{N} \mid \\ \qquad \langle\!\langle [\, i \mapsto 0 : \mathbb{N} \mid (i := i + 1) \cdot (n := n + i)\,] \rangle\!\rangle \\ \quad \| \\ \qquad \langle\!\langle [\, i \mapsto 1 : \mathbb{N} \mid (i := i + 1) \cdot (n := n \times i)\,] \rangle\!\rangle \\ ] \end{array}$$

As can be seen, both transactions executed in parallel have a local variable i which is (and can be) only used inside the transaction. Of course, since variable i is local in both transactions, we do not want updates of i to cause the transaction running in parallel being locked with respect to its own local variable i. However, since we extend locking and unlocking actions with valuation sets, these sets do pass scope operators and can therefore conflict with other scopes. To prevent this behaviour, when leaving a scope, the local variables should be filtered from the valuation sets attached to both locking and unlocking actions. We do this by using the without operator as defined in Definition 8.5.5. To update the valuation sets which are added to the locking and unlocking actions, we make a case distinction on the type of actions for the deduction rules of the state operator, as we do with all transactional

operators. This leads to three rules instead of the second rule in Table 6.3 on page 64:

$$\frac{\langle x, (i \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \xrightarrow{\mathcal{U}_V} \langle x', (i \mapsto c : T) :: \sigma', t\rangle}{\langle [\, i \mapsto e : T \mid x \,], \sigma, t\rangle \xrightarrow{\mathcal{U}_V \setminus \{i \mapsto e(\sigma,t):T\}} \langle [\, i \mapsto c : T \mid x' \,], \sigma', t\rangle}$$

$$\frac{\langle x, (i \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \xrightarrow{a_{n,W}} \langle x', (i \mapsto c : T) :: \sigma', t\rangle}{\langle [\, i \mapsto e : T \mid x \,], \sigma, t\rangle \xrightarrow{a_{n,W} \setminus \{i \mapsto e(\sigma,t):T\}} \langle [\, i \mapsto c : T \mid x' \,], \sigma', t\rangle}$$

$$\frac{\langle x, (i \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \xrightarrow{a} \langle x', (i \mapsto c : T) :: \sigma', t\rangle}{\langle [\, i \mapsto e : T \mid x \,], \sigma, t\rangle \xrightarrow{a} \langle [\, i \mapsto c : T \mid x' \,], \sigma', t\rangle}$$

As a result of using the new deduction rules, variable i in the example is no longer locking the i in the other transaction since it is stripped off of the valuation set of the locking action $i := i + 1$ when leaving its scope.

An overview of all deduction rules for the transactional operators combined with states can be found in Table 9.10 on page 128.

8.6 Degrees of Isolation

In Section 8.1 we shortly mentioned degrees of isolation. We discuss this concept in more detail in this section.

All transactions have a so-called *degree of isolation* which specifies the structure and dependencies between actions in the transactions. A distinction can be drawn between four degrees of isolation, mainly due to performance issues [GR93]. A short overview of the different degrees is given in Table 8.7.

A 0° isolated transaction is called *chaos*. It does not overwrite another transaction's dirty data if the other transaction is 1° or greater.

A 1° isolated transaction is called *browse*. It prevents data updates to get lost. It is both well-formed and two-phase with respect to writes. A transaction is said to be *well-formed with respect to writes* if all data updates are preceded by locks, locking data updates to the same data in other transactions until it is committed or rolled back. *Two-phase* means that all locks precede all unlocks.

A 2° isolated transaction, called *cursor stability*, has the same properties as a first degree transaction, but it also implements well-formedness with respect to reads. So also the reading of updated data by other transactions is locked.

Issue	Degree 0	Degree 1	Degree 2	Degree 3
Common name	Chaos	Browse	Cursor Stability	Isolated
Protection provided	Lets others run at higher isolation	0° and no lost updates	No lost updates, no dirty reads	No lost updates, no dirty reads, repeatable reads
Transaction structure	Well-formed w.r.t. write	Well-formed w.r.t. write and two-phase w.r.t. write	Well-formed and two-phase w.r.t. write	Well-formed and two-phase
Dependen-cies	None	write → write	write → write write → read	write → write write → read read → write

Table 8.7: Degrees of isolation as given in [GR93].

Finally, a 3° isolated transaction also locks data read by a transaction until it commits or rolls back. This is called *isolated*, *serialisable* or *repeatable reads*. Optimally, all transactions should be 3° isolated.

We can extend our process algebraic model such that we can distinguish between degrees of isolation. Although this does not increase the complexity of the transactional locking mechanism, we need to add lots of extra syntax to make this possible. One of the causes is that we need to introduce an explicit read action. On the other hand this is caused by the fact that actions from within 1° isolated transactions both are lockable and cause other actions to get locked. So the lock counter we make use of in combination with the set of locking variables is not only a counter which stores the number of times the action is locked, but it also specifies that execution of the action causes other actions to get locked (as stated in axiom M8). We call an action that causes other actions to get locked a *locking action*. In zeroth and second degree isolated transactions, lockable actions do not necessarily have to be locking and vice versa, as can be seen in Table 8.8. So apart from the lock counter (and the set of locking variables), we need to extend actions from within transactions with a locking attribute to indicate whether the action is a locking action. This of course depends on the degree of isolation, which should be added to the transactional operators.

In the previous section we introduced a function, U, for determining the variables that are updated by actions. When adding degrees of isolation, we also need to introduce a function on actions for determining the set of variables that are read.

Furthermore, we need to draw a distinction between *shared locks* and *exclusive locks* [GR93]. Shared locks are set by read actions from within 3° isolated transactions and cause other transactions not to write to variables read by the transactions. However, the other transactions are allowed to read them. On the other hand, write actions set exclusive locks on variables, causing other transactions to get locked when access-

degree of isolation	lockable write	lockable read	locking write	locking read
zeroth	yes	no	no	no
first	yes	no	yes	no
second	yes	yes	yes	no
third	yes	yes	yes	yes

Table 8.8: Lockable versus locking actions.

ing variables, even for read actions. A 3° isolated transaction can upgrade a shared lock to an exclusive lock by writing to a variable it has a shared lock on. The set of variables or valuations that we stored in the auxiliary transactional operator (the A in $\langle\langle x, A, y\rangle\rangle$) is used for keeping track of the variables the transaction has exclusively locked. By introducing shared locks, we need to either split this set into two sets, one for shared and one for exclusive locks, or we should extend the elements in the set with an attribute which indicates whether the element is shared locked or exclusively locked.

To conclude, adding degrees of isolation to the formalism presented leads to the introduction of several extensions. Although these extensions do not make the concepts more complex, they introduce a considerably large amount of syntactical overhead, which goes beyond the scope of this thesis.

8.7 Related Work

Since both transactions and process algebra are widely used concepts, much research is done in both areas.

Transactions can be considered as groups of actions. In process algebra there are mechanisms available to group actions. In [BKT85] a mechanism is introduced for specifying asynchronous communication between processes, based on process algebra. Each communication consists of (independent) write and read actions. Each read action should be preceded by its corresponding write action. If this is not the case, actions can get locked which can be compared with transactional locking. The semantics of this mechanism is given in [BKP92]. In [BK84b] the *tight multiplication* operator is introduced. This operator is used in the same way as the sequential composition operator. However, no interleaving can take place between two actions which are composed into a process using this tight multiplication operator. Transactions can roll back, causing actions being undone. In [BPW94], a mechanism for modelling this undoing of actions is added to process algebra. The choice for executing such undo actions is deterministic in contrast to the nondeterministic choice for

rolling back a transaction. In [BM01], Bruni and Montarnari introduce Zero-safe net models, which are Petri nets that can be used for the modelling of transactions. In [BLM02], join calculus is used for the modelling of transactions, based on Zero-safe petri nets.

The classical transaction concept appeared for the first time in [EGLT76]. In [Gra81] the ACID properties of transactions are explicitly indicated. A nice and complete overview of the main concepts of transactions is given and discussed by Gray and Reuter in [GR93]. Our model of nesting of transactions is an extension of the concept of nested transactions as developed by Moss [Mos81]. Other extensions of Moss' concept are for example multi-level transactions [Wei86] and open nested transactions [WS92]. For the concepts described in this chapter we make use of the two-phase locking (2PL) protocol as introduced in [EGLT76]. Many variations on the 2PL technique exist [AD76, GR93, SGMS94]. Apart from two-phase locking, other concurrency control mechanisms exist, like timestamp-ordering (TO) techniques [SM77] and optimistic schedulers [Bad79]. All techniques can be combined into hybrid techniques, e.g. 2PL-TO combinations [BG81].

9

Modelling Internet Applications

In this chapter we bring together the ideas and concepts introduced in former chapters of this thesis. We combine all elements to come to a language for specifying Internet applications. So in this chapter we give a formal definition of the syntax and semantics of *DiCons*, which can be used for modelling Internet applications (see also Section 1.4). The model is geared towards modelling the interaction behaviour of Internet applications with respect to its users. In Chapters 10 and 11 we make use of these specifications for the testing of running Internet applications and for the generation of executable applications, given a specification.

To express the semantics of the processes we are interested in, we make use of process graphs, which are defined using Structured Operational Semantics rules, as introduced by Plotkin [Plo81].

First of all, in Section 9.1 we give an overview of the types used in the specifications. We give an overview of the action alphabet of *DiCons* in Section 9.2. Next, in Section 9.3 we give a summary of the operators introduced before. A complete overview of the structured operational semantics is given in Section 9.4. To complete this chapter, we give an example of a formal specification of an Internet application and prove some properties in Section 9.5.

This chapter contains many lists, tables, and references to former pages, chapters and tables. Although we know that this makes this chapter less readable, we do include them here to present a complete overview of the model.

9.1 Types

In former chapters we introduced several types for several parts of the formal specifications. We revise them here, give short explanations, and show the way we use them.

$\mathbb{U}$ **The universe of users** *(page 68)*
This set contains all possible users. Each user interacting with an application is represented by a u of type $\mathbb{U}$.

$\mathbb{G}$ **The universe of groups** *(page 72)*
This set contains all possible groups, which themselves are sets of users. So, $\mathbb{G} = \mathcal{P}(\mathbb{U})$.

$\mathbb{M}$ **The universe of messages** *(page 68)*
All messages are uniquely identified by an element in $\mathbb{M}$ and can therefore be represented by their symbolic name m in $\mathbb{M}$.

$\mathbb{P}_i$ **The universe of input parameters** *(page 68)*
A message can be extended with parameters. For request messages these parameters are variables that are assigned by a user by filling in a Web form, i.e., input parameters denoted by $i, i_1, i_2, \ldots$

$\mathbb{P}_o$ **The universe of output parameters** *(page 68)*
A message can be extended with parameters. For response messages and e-mail messages, the parameters are values of expressions to be included in the actual message that needs to be sent to the requesting user, i.e., output parameters denoted by $o, o_1, o_2, \ldots$

$\mathbb{V}$ **The universe of valuations** *(page 103)*
This set contains all possible valuations where valuations of variables are as defined in Definition 6.2.1.

$\mathbb{S}$ **The universe of states** *(page 59)*
The set of all possible states. When making use of the term *state* we mean the representation of the state by a valuation stack as defined in Definition 6.2.3.

$\mathfrak{T}$ **Time** *(page 63)*
Time is measured on a discrete scale, as defined in Section 6.4.

$\mathbb{K}$ **The universe of session labels** *(page 74)*
The set of all possible session labels as defined in Definition 7.3.1. Session labels are used for uniquely identifying sessions of connected interactions.

We introduce one more set, viz. the set of all process terms.

$\mathbb{X}$	The set of all process terms
$\mathbb{U}$	The universe of users
$\mathbb{G}$	The universe of groups
$\mathbb{M}$	The universe of messages
$\mathbb{P}_i$	The universe of input parameters
$\mathbb{P}_o$	The universe of output parameters
$\mathbb{V}$	The universe of valuations
$\mathbb{S}$	The universe of states
$\mathfrak{T}$	Time
$\mathbb{K}$	The universe of session labels

Table 9.1: Types in the *DiCons* model.

$\mathbb{X}$ **Process terms**
This set contains all *DiCons* process terms.

A short overview of the types is given in Table 9.1.

9.2 Alphabet

The alphabet of actions, $\mathbb{A}$, consists of four sets of actions, viz. local or internal actions, communication primitives, unlocking actions and lockable actions:

$$\mathbb{A} \quad =_{\text{def}} \quad \mathbb{I} \cup \mathbb{C} \cup \mathbb{UL} \cup \mathbb{L}$$

Apart from the actions in $\mathbb{A}$, the alphabet of *DiCons* contains two more elements, viz. deadlock δ and the empty process ε.

δ **Unsuccessful termination** *(page 52)*
In Section 5.4 we introduced deadlock, which states unsuccessful termination.

ε **Successful termination** *(page 52)*
In Section 5.5 we introduced the empty process, which expresses the process that can only terminate successfully.

$\mathbb{I}$ **The internal actions** *(page 62)*
$\mathbb{I}$ is the set of all possible internal actions as briefly introduced in Section 6.3. As mentioned before, these actions can be simple and self-explaining. If functions show complex behaviour, the effect of the action on the state should be provided explicitly.

$\mathbb{A}$	The *DiCons* alphabet, $\{\delta, \varepsilon\} \cup \mathbb{I} \cup \mathbb{C} \cup \mathbb{UL} \cup \mathbb{L}$
δ	Unsuccessful termination
ε	Successful termination
$\mathbb{I}$	The internal actions
$\mathbb{C}$	The communication primitives
$\mathbb{UL}$	The unlocking actions
$\mathbb{L}$	The locking actions

Table 9.2: Alphabet of the *DiCons* model.

$\mathbb{C}$ **The communication primitives** *(page 69)*
The alphabet of communication primitives is introduced in Section 7.1:

$$\begin{array}{lll} \mathbb{C} =_{\text{def}} & \{\,\text{req}.u.\vec{\imath} & \mid u \in \mathbb{U}, i_1, \ldots, i_n \in \mathbb{P}_i \quad \} \cup \\ & \{\,\text{resp}.u.m.\vec{o} & \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \} \cup \\ & \{\,u \leftarrow m(\vec{o}) & \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \} \cup \\ & \{\,u \rightleftarrows m(\vec{o}) & \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \} \cup \\ & \{\,u \overset{\vec{\imath}}{\rightleftarrows} m(\vec{o}; \vec{\imath}) & \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o, i_1, \ldots, i_n \in \mathbb{P}_i \quad \} \cup \\ & \{\,u \overset{\vec{\imath}}{\leftarrow} m(\vec{o}; \vec{\imath}) & \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o, i_1, \ldots, i_n \in \mathbb{P}_i \quad \} \cup \\ & \{\,u \leftarrow m(\vec{o}) & \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \} \end{array}$$

$\mathbb{UL}$ **The unlocking actions** *(pages 85, 105)*
The alphabet of unlocking actions is introduced in Section 8.2.1. We adapted this alphabet in Section 8.5.2, where unlocking actions are combined with states:

$$\mathbb{UL} \quad =_{\text{def}} \quad \{\mathcal{U}_V \mid \mathcal{U} \in \{\mathcal{C}, \mathcal{R}\},\ V \subseteq \mathbb{V}\}$$

We model a commit action by a $\mathcal{C}$ and a rollback action by an $\mathcal{R}$.

$\mathbb{L}$ **The locking actions** *(pages 86, 108)*
The alphabet of locking actions is also introduced in Section 8.2.1. To combine locking actions with states, this alphabet in adapted in Section 8.5.4:

$$\mathbb{L} \quad =_{\text{def}} \quad \{a_{n,V} \mid a \in \mathbb{I} \cup \mathbb{C},\ n \in \mathbb{N},\ V \subseteq \mathbb{V}\}$$

The locking and unlocking actions are auxiliary actions, which are not used when giving specifications. A short overview of the elements of the actions in the alphabet is given in Table 9.2.

9.3 Operators

In this section we summarise the operators introduced in this thesis. We only give their syntax and informal definitions here. To be able to give these definitions, we

x, y	$\in \mathbb{X}$	*DiCons* process terms
$\mathbf{a}$	$\in \mathbb{A}$	an action
a	$\in \mathbb{I} \cup \mathbb{C}$	a non-auxiliary action
p	$\in \mathbb{I}$	an internal action
$\mathcal{U}$	$\in \{\mathcal{C}, \mathcal{R}\}$	an unlocking constant
u	$\in \mathbb{U}$	a user
G, H	$\in \mathbb{G}$	groups of users
m	$\in \mathbb{M}$	a message
i	$\in \mathbb{P}_i$	input parameters
o	$\in \mathbb{P}_o$	output parameters
k	$\in \mathbb{K}$	a session label
b	$\in \mathbb{B}$	a boolean expression
n	$\in \mathbb{N}$	a natural number
σ	$\in \mathbb{S}$	a state
t	$\in \mathfrak{T}$	a time stamp
V, W	$\subseteq \mathbb{V}$	valuation sets
j		an arbitrary variable identifier
T		an arbitrary (abstract data) type
c, d		arbitrary constants
e		an arbitrary expression

... and any use of primes and vector notation.

Table 9.3: Variables used in the *DiCons* model.

make use of some variables which are summarised in Table 9.3. These variables are also used in Section 9.4 where the operational semantics is presented. We identify the operators that should not be used when giving specifications, i.e. the auxiliary operators.

$x + y$ **Alternative composition** *(auxiliary, page 50)*
Alternative composition is used for constructing a choice between processes. Given processes x and y, $x + y$ is the process that executes either x or y.

$x \cdot y$ **Sequential composition** *(page 50)*
The sequential composition operator can be used for defining processes in which (sub)processes should take place sequentially. Given processes x and y, $x \cdot y$ is the process that first executes x, and after completion of x continues with executing y.

$x \triangleleft b \triangleright y$ **Conditional branching** *(page 54)*
The conditional branching operator behaves like the *if-then-else-fi* operator in

sequential programming: $x \triangleleft b \triangleright y \equiv \mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}$.

$b \rhd\!\rhd x$ **Conditional repetition** *(page 54)*
The conditional repetition can be compared with a while loop in traditional programming. We make use of the conditional repetition operator ($\_ \rhd\!\rhd \_$) to specify these repetitions: $b \rhd\!\rhd x \equiv \mathit{while}\ b\ \mathit{do}\ x\ \mathit{od}$.

$b \rhd\!\!- x$ **Conditional disrupt** *(page 56)*
Process $b \rhd\!\!- x$ specifies that process x is normally executed until b becomes true. At that moment the process terminates, independent of the (inter)actions that are taking place at that moment.

$[\, j \mapsto e : T \mid x \,]$ **Variable declaration** *(pages 60, 110)*
The scope operator is used for declaring variables. The left argument is a valuation, the right argument is the process which defines the scope of the valuation.

$x \parallel y$ **Parallel composition** *(pages 73, 89)*
Putting processes x and y in parallel, denoted by $x \parallel y$, means the execution of x and y takes place concurrently.

$?_u x$ **Anonymous replication** *(page 74)*
Process $?_u x$ expresses that all users u in the universe of users $\mathbb{U}$ can anonymously execute (inter)actions in process x between unknown user u and the application in parallel and more than once. The u can occur in x and is bound as soon as the first action of x is executed.

$!_{u \in G} x$ **Replication** *(page 75)*
Process $!_{u \in G} x$ expresses that all users u ($u \in G$) can execute (inter)actions in process x between user u and the application in parallel and more than once. The u can occur in x and is bound as soon as the first action of x is executed.

$!^H_{u \in G} x$ **Extended replication** *(page 76)*
process $!^H_{u \in G}\, x$ expresses that all users in group G are allowed to execute process x more than once. Group H contains the users that are known to the application. If $u \notin H$, registration takes place and the user is added to H. If $u \in H$, identification takes place.

$\|_{u \in G} x$ **Generalised parallel composition** *(page 75)*
Process $\|_{u \in G} x$ specifies that all users u ($u \in G$) execute (inter)actions in process x between user u and the application in parallel but only once. The u can occur in x and is bound as soon as the first action of x is executed.

$\|^H_{u \in G} x$ **Extended generalised parallel composition** *(page 76)*
Process $\|^H_{u \in G}\, x$ specifies that all users u ($u \in G$) execute (inter)actions in process

x between user u and the application in parallel but only once. Group H contains the users that are known to the application. If $u \notin H$, registration takes place and the user is added to H. If $u \in H$, identification takes place.

$\langle\langle x \rangle\rangle$ **Transactional composition** *(page 83)*
The transactional composition is used for turning processes into transactions such that the process shows transactional behaviour.

$\langle\langle x, V, y \rangle\rangle$ **Extended transactional composition** *(auxiliary, pages 83, 104)*
Process $\langle\langle x, V, y \rangle\rangle$ can be read as transactional process x, which has already updated valuations V and still has to execute process y before a commit statement can take place.

$[x]_V$ **Process locking** *(auxiliary, pages 88, 108)*
Process $[x]_V$ specifies that all lockable actions in x which update variables in valuation set V get locked with respect to these variables.

$\lfloor x \rfloor_V$ **Process unlocking** *(auxiliary, pages 88, 108)*
Process $\lfloor x \rfloor_V$ specifies that all lockable actions in x which update variables in valuation set V get unlocked with respect to these variables.

A short overview of the operators together with their types is given in Table 9.4.

9.4 Operational semantics

To express the operational semantics of *DiCons* specifications we make use of Plotkin-style SOS rules [Plo81]. As explained in Section 6.4.1, we give a process graph to represent transitions on tuples having as arguments a process term, a state and a moment in time/a time slice. In this section we give the deduction rules for the operators we make use of for specifying Internet applications.

9.4.1 Transition Labels

Before giving the deduction rules, we need to specify the transition labels. The set of possible action labels, $\mathbb{A}_l$, is based on a subset of the alphabet of *DiCons*, $\mathbb{A}$. It contains internal action labels, e-mail, request and response action label and both locking and unlocking labels:

$$\mathbb{A}_l \quad =_{\text{def}} \quad \mathbb{I} \cup \mathbb{C}_l \cup \mathbb{UL} \cup \mathbb{L}_l$$

$\_ + \_$	$: \mathbb{X} \times \mathbb{X} \to \mathbb{X}$	Alternative composition (auxiliary)
$\_ \cdot \_$	$: \mathbb{X} \times \mathbb{X} \to \mathbb{X}$	Sequential composition
$\_ \lhd \_ \rhd \_$	$: \mathbb{X} \times \mathbb{B} \times \mathbb{X} \to \mathbb{X}$	Conditional branching
$\_ \rhd\rhd \_$	$: \mathbb{B} \times \mathbb{X} \to \mathbb{X}$	Conditional repetition
$\_ \rhd\!\!- \_$	$: \mathbb{B} \times \mathbb{X} \to \mathbb{X}$	Conditional disrupt
$[\_ \mid \_]$	$: \mathbb{V} \times \mathbb{X} \to \mathbb{X}$	Variable declaration
$\_ \parallel \_$	$: \mathbb{X} \times \mathbb{X} \to \mathbb{X}$	Parallel composition
$?\_$	$: \mathbb{D} \times \mathbb{X} \to \mathbb{X}$	Anonymous replication
$!_{\_\in\_}$	$: \mathbb{D} \times \mathbb{G} \times \mathbb{X} \to \mathbb{X}$	Replication
$!^{\_}_{\_\in\_}$	$: \mathbb{D} \times \mathbb{G} \times \mathbb{G} \times \mathbb{X} \to \mathbb{X}$	Extended replication
$\parallel_{\_\in\_}$	$: \mathbb{D} \times \mathbb{G} \times \mathbb{X} \to \mathbb{X}$	Generalised parallel composition
$\parallel^{\_}_{\_\in\_}$	$: \mathbb{D} \times \mathbb{G} \times \mathbb{G} \times \mathbb{X} \to \mathbb{X}$	Extended generalised parallel composition
$\langle\langle \_ \rangle\rangle$	$: \mathbb{X} \to \mathbb{X}$	Transactional composition
$\langle\langle \_, \_, \_ \rangle\rangle$	$: \mathbb{X} \times \mathcal{P}(\mathbb{V}) \times \mathbb{X} \to \mathbb{X}$	Extended transactional composition (auxiliary)
$[\_]_{\_}$	$: \mathbb{X} \times \mathcal{P}(\mathbb{V}) \to \mathbb{X}$	Process locking (auxiliary)
$\lfloor \_ \rfloor_{\_}$	$: \mathbb{X} \times \mathcal{P}(\mathbb{V}) \to \mathbb{X}$	Process unlocking (auxiliary)

Table 9.4: Operators of the *DiCons* model.

$\mathbb{C}_l$ **The communication labels** *(page 70)*

The set of communication actions that serve as transition labels:

$$\begin{aligned} \mathbb{C}_l \quad =_{\text{def}} \quad & \{\, \text{mail}.u.m.\vec{o} \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \} \cup \\ & \{\, \text{req}.u.\vec{\imath}.\vec{d} \quad \mid u \in \mathbb{U}, i_1, \ldots, i_n \in \mathbb{P}_i, \vec{d} \in \text{type}(\vec{\imath}) \,\} \cup \\ & \{\, \text{resp}.u.m.\vec{o} \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \} \end{aligned}$$

Note that the type function used in the second set actually depends on the state the process is in. We use type($\vec{\imath}$) to express the types of $i_1, \ldots, i_n$.

$\mathbb{L}_l$ **The locking labels**

The subset of locking actions:

$$\mathbb{L}_l \quad =_{\text{def}} \quad \{a_{n,V} \mid a \in \mathbb{I} \cup \mathbb{C}_l,\ n \in \mathbb{N},\ V \subseteq \mathbb{V}\}$$

Internal action labels and unlocking labels can be one-to-one mapped to the internal actions and unlocking actions as defined in Section 9.2.

Furthermore, as explained in Section 7.3.1, transitions are also labelled with a session label $k \in \mathbb{K}$. We put the action label above and the session label below the transition arrow.

9.4.2 Deduction Rules

In the remainder of this section we give the deduction rules for all actions in the alphabet and for all operators of *DiCons*.

$$\frac{}{\langle \varepsilon, \sigma, t \rangle \downarrow}\,{}^{1} \qquad \frac{}{\langle p, \sigma, t \rangle \xrightarrow[\lambda]{\text{action}(p,\sigma,t)} \langle \varepsilon, \text{effect}(p, \sigma, t), t \rangle}\,{}^{2}$$

$$\frac{}{\langle x, \sigma, t \rangle \overset{tick}{\longmapsto} \langle x, \sigma, t+1 \rangle}\,{}^{3}$$

Table 9.5: Deduction rules for the empty process, internal actions and the time step.

In Table 9.5 the deduction rules for the empty process, the internal actions and the time step are given. The empty process terminates successfully, as stated in rule 1. Termination notation $\downarrow$ is explained in Section 5.5.

Rule 2 states that internal actions are evaluated in state σ at time t. The new state is determined by the effect of action p on state σ at time t.

The semantics of progression in time is given in rule 3. At any moment, a time step, denoted by $\overset{tick}{\longmapsto}$ can occur. As mentioned in Section 6.4, time is measured on a discrete scale. We use a different kind of arrow to indicate that time steps do not affect the actual state of the system. However, progression in time *can* influence process behaviour since time can be used in calculations and conditions.

$$\frac{u(\sigma,t)=c, \quad \vec{d} \in \mathrm{type}(\vec{\imath},\sigma)}{\langle \mathrm{req}.u.\vec{\imath},\sigma,t\rangle \xrightarrow[\lambda]{\mathrm{req}.c.\vec{\imath}.\vec{d}} \langle \varepsilon,\sigma[\vec{d}/\vec{\imath}],t\rangle}\,{}_{4} \qquad \frac{u(\sigma,t)=c, \quad \vec{o}(\sigma,t)=\vec{d}}{\langle \mathrm{resp}.u.m.\vec{o},\sigma,t\rangle \xrightarrow[\lambda]{\mathrm{resp}.c.m.\vec{d}} \langle \varepsilon,\sigma,t\rangle}\,{}_{5}$$

$$\frac{u(\sigma,t)=c, \quad c \neq \bot, \quad \vec{o}(\sigma,t)=\vec{d}}{\langle u \leftarrow m(\vec{o}),\sigma,t\rangle \xrightarrow[\lambda]{\mathrm{mail}.c.m.\vec{d}} \langle \varepsilon,\sigma,t\rangle}\,{}_{6} \qquad \frac{u(\sigma,t)=c}{\langle u \rightleftharpoons m(\vec{o}),\sigma,t\rangle \xrightarrow[\lambda]{\mathrm{req}.c.\varepsilon.\varepsilon} \langle \mathrm{resp}.u.m.\vec{o},\sigma,t\rangle}\,{}_{7}$$

$$\frac{u(\sigma,t)=c}{\langle u \overset{\cdot}{\rightleftharpoons} m(\vec{o};\vec{\imath}),\sigma,t\rangle \xrightarrow[\lambda]{\mathrm{req}.c.\varepsilon.\varepsilon} \langle \mathrm{resp}.u.m.\vec{o}\cdot \mathrm{req}.u.\vec{\imath},\sigma,t\rangle}\,{}_{8}$$

$$\frac{u(\sigma,t)=c, \quad \vec{o}(\sigma,t)=\vec{d}}{\langle u \overset{\cdot}{\rightleftharpoons} m(\vec{o};\vec{\imath}),\sigma,t\rangle \xrightarrow[\lambda]{\mathrm{resp}.c.m.\vec{d}} \langle \mathrm{req}.u.\vec{\imath},\sigma,t\rangle}\,{}_{9} \qquad \frac{u(\sigma,t)=c, \quad \vec{o}(\sigma,t)=\vec{d}}{\langle u \overset{\cdot}{\leftarrow} m(\vec{o}),\sigma,t\rangle \xrightarrow[\lambda]{\mathrm{resp}.c.m.\vec{d}} \langle \varepsilon,\sigma,t\rangle}\,{}_{10}$$

Table 9.6: Deduction rules for the communication primitives.

In Table 9.6 the deduction rules for the communication primitives are given. Except for the (empty) session labels, the rules in this table correspond to the deduction rules in Table 7.1 on page 71. Therefore, we only shortly explain them here.

The first two rules, rules 4 and 5, show the semantics of the atomic HTTP communication actions: they can do a step labelled with their evaluation in the current state and then successfully terminate. Only the request action affects the state.

Rule 6 describes the semantics of the mail sending action. Rules 7 to 10 describe the semantics of the composed communication primitives. These composed actions execute their first atomic request or response action, after which the rest of the HTTP interaction modelled by the primitive should take place. More information on the communication primitives can be found in Section 4.3 and in Chapter 7.

$$\frac{\langle x,\sigma,t\rangle\downarrow,\quad \langle y,\sigma,t\rangle\downarrow}{\langle x\cdot y,\sigma,t\rangle\downarrow}\,{}_{11} \qquad \frac{\langle x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle}{\langle x\cdot y,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x'\cdot y,\sigma',t\rangle}\,{}_{12}$$

$$\frac{\langle x,\sigma,t\rangle\downarrow,\quad \langle y,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle y',\sigma',t\rangle}{\langle x\cdot y,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle y',\sigma',t\rangle}\,{}_{13} \qquad \frac{\langle x,\sigma,t\rangle\downarrow}{\langle x+y,\sigma,t\rangle\downarrow,\quad \langle y+x,\sigma,t\rangle\downarrow}\,{}_{14}$$

$$\frac{\langle x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle}{\langle x+y,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle,\quad \langle y+x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle}\,{}_{15}$$

Table 9.7: Deduction rules for the alternative and sequential composition operators.

Table 9.7 shows the deduction rules for the alternative and sequential composition operators. These rules are also given in Table 6.4 on page 65. Again only session labels are added.

If both processes x and y terminate successfully, than the sequential composition of x and y also terminates successfully (rule 11). Rule 12 states that if x can do an **a**-step to x', then $x \cdot y$ can do an **a**-step to $x' \cdot y$. In rule 13 it is stated that if x terminates successfully and y can do an **a**-step to y', that $x \cdot y$ can do this **a**-step to y' as well. Next, deduction rule 14 states that if x terminates successfully, then both $x + y$ and $y + x$ can terminate successfully. Finally, rule 15 states that if x can do an **a**-step to x', then $x + y$ and $y + x$ can do a similar **a**-step to x'. Note that no deduction rules involving deadlock process δ are available. When reaching a δ state, neither successful termination nor a transition can occur.

$$\frac{\langle x,\sigma,t\rangle\downarrow,\quad b(\sigma,t)}{\langle x \triangleleft b \triangleright y,\sigma,t\rangle\downarrow}\,{}_{16} \qquad \frac{\langle x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle,\quad b(\sigma,t)}{\langle x \triangleleft b \triangleright y,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle}\,{}_{17}$$

$$\frac{\langle y,\sigma,t\rangle\downarrow,\quad \neg b(\sigma,t)}{\langle x \triangleleft b \triangleright y,\sigma,t\rangle\downarrow}\,{}_{18} \qquad \frac{\langle y,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle y',\sigma',t\rangle,\quad \neg b(\sigma,t)}{\langle x \triangleleft b \triangleright y,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle y',\sigma',t\rangle}\,{}_{19}$$

$$\frac{\langle x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle,\quad b(\sigma,t)}{\langle b \triangleright\!\!\triangleright x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x'\cdot(b \triangleright\!\!\triangleright x),\sigma',t\rangle}\,{}_{20} \qquad \frac{\neg b(\sigma,t)}{\langle b \triangleright\!\!\triangleright x,\sigma,t\rangle\downarrow}\,{}_{21}$$

$$\frac{\langle x,\sigma,t\rangle\downarrow}{\langle b \triangleright\!\!- x,\sigma,t\rangle\downarrow}\,{}_{22} \qquad \frac{b(\sigma,t)}{\langle b \triangleright\!\!- x,\sigma,t\rangle\downarrow}\,{}_{23} \qquad \frac{\langle x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle x',\sigma',t\rangle,\quad \neg b(\sigma,t)}{\langle b \triangleright\!\!- x,\sigma,t\rangle \xrightarrow[k]{\mathbf{a}} \langle b \triangleright\!\!- x',\sigma',t\rangle}\,{}_{24}$$

Table 9.8: Deduction rules for the conditional operators.

Table 9.8 shows the deduction rules for the conditional operators. These rules are given without session labels in Table 6.4 on page 65.

Rules 16 to 19 are introduced in Section 5.7.1 and describe the behaviour of the conditional branching operator. Rule 16 and 18 state that if, as result of the evaluation of the condition, the chosen alternative terminates, then the process terminates.

On the other hand, rule 17 and 19 state that if the alternative subprocess executes an action, then the action is also executed by the constructed process.

Rules 20 and 21 are introduced in Section 5.7.2. The first deduction rule states that if the process under conditional repetition can execute an action and the condition holds, that the constructed process also executes the action. As a result, the rest of the process should terminate successfully, after which the (conditional) process restarts. If the condition does not hold in the current state, then the process terminates successfully, as stated in rule 21.

Finally, rules 22 to 24 are introduced in Section 5.7.3. Rule 22 states that if the conditionally disrupted process terminates then the constructed process terminates. In rule 23 it is stated that if the condition holds, the process is disrupted and successfully terminates. If the process is not disrupted and can execute an action, then the

$$\frac{\langle x, (j \mapsto e(\sigma) : T) :: \sigma, t\rangle \downarrow}{\langle [\, j \mapsto e : T \mid x \,], \sigma, t\rangle \downarrow}\ {}_{25}$$

$$\frac{\langle x, (j \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \xrightarrow[k]{\mathcal{U}_V} \langle x', (j \mapsto c : T) :: \sigma', t\rangle}{\langle [\, j \mapsto e : T \mid x \,], \sigma, t\rangle \xrightarrow[k]{\mathcal{U}_V \nwarrow \{j \mapsto e(\sigma,t):T\}} \langle [\, j \mapsto c : T \mid x' \,], \sigma', t\rangle}\ {}_{26}$$

$$\frac{\langle x, (j \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \xrightarrow[k]{a_{n,W}} \langle x', (j \mapsto c : T) :: \sigma', t\rangle}{\langle [\, j \mapsto e : T \mid x \,], \sigma, t\rangle \xrightarrow[k]{a_{n,W} \nwarrow \{j \mapsto e(\sigma,t):T\}} \langle [\, j \mapsto c : T \mid x' \,], \sigma', t\rangle}\ {}_{27}$$

$$\frac{\langle x, (j \mapsto e(\sigma, t) : T) :: \sigma, t\rangle \xrightarrow[k]{a} \langle x', (j \mapsto c : T) :: \sigma', t\rangle}{\langle [\, j \mapsto e : T \mid x \,], \sigma, t\rangle \xrightarrow[k]{a} \langle [\, j \mapsto c : T \mid x' \,], \sigma', t\rangle}\ {}_{28}$$

Table 9.9: Deduction rules for the scope operator.

constructed process can also execute the action, as stated in rule 24.

Table 9.9 contains the deduction rules for the scope operator. Rule 25 states successful termination, which is also given in Table 6.1 on page 62. Except for the session labels, rules 26 to 28 are also given and explained in more detail on page 111. These rules are concerned with the execution of actions with a scope and their effect on valuations defined with the scope operator.

Rule 28 can be compared with the second rule in Table 6.1, stating that executing actions within a scope can affect the valuation of the declared variable. Rules 26 and 27 also express this behaviour, however, they define some different behaviour for the locking and unlocking actions. Rule 26 is concerned with the execution of an unlocking action. The set of unlocking valuations (the V in $\mathcal{U}_V$) is based on the scope of the unlocking action. Therefore, leaving a scope causes the variable to be stripped off of the set of unlocking valuations. The same holds for locking actions, as stated in deduction rule 27.

$$\frac{}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{\mathcal{R}_V} \langle\langle\langle x\rangle\rangle, \sigma[V], t\rangle}\ {}_{29} \qquad \frac{\langle x, \sigma, t\rangle \downarrow}{\langle\langle\langle x\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{\mathcal{C}_\emptyset} \langle \varepsilon, \sigma, t\rangle}\ {}_{30}$$

$$\frac{\langle y, \sigma, t\rangle \downarrow}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{\mathcal{C}_V} \langle \varepsilon, \sigma, t\rangle}\ {}_{31} \qquad \frac{\langle x, \sigma, t\rangle \xrightarrow[k]{\mathcal{U}_W} \langle x', \sigma', t\rangle}{\langle\langle\langle x\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{\mathcal{U}_\emptyset} \langle\langle\langle x, \emptyset, x'\rangle\rangle, \sigma', t\rangle}\ {}_{32}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow[k]{a_{n,W}} \langle x', \sigma', t\rangle}{\langle\langle\langle x\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{a_{n,\mathrm{U}(a,\sigma,t)}} \langle\langle\langle x, \mathrm{U}(a, \sigma, t), x'\rangle\rangle, \sigma', t\rangle}\ {}_{33}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow[k]{a} \langle x', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) \neq \emptyset}{\langle\langle\langle x\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{a_{0,\mathrm{U}(a,\sigma,t)}} \langle\langle\langle x, \mathrm{U}(a, \sigma, t), x'\rangle\rangle, \sigma', t\rangle}\ {}_{34}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow[k]{a} \langle x', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) = \emptyset}{\langle\langle\langle x\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{a} \langle\langle\langle x, \emptyset, x'\rangle\rangle, \sigma', t\rangle}\ {}_{35} \qquad \frac{\langle y, \sigma, t\rangle \xrightarrow[k]{\mathcal{U}_W} \langle y', \sigma', t\rangle}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{\mathcal{U}_\emptyset} \langle\langle\langle x, V, y'\rangle\rangle, \sigma', t\rangle}\ {}_{36}$$

$$\frac{\langle y, \sigma, t\rangle \xrightarrow[k]{a_{n,W}} \langle y', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) \,\dot{\setminus}\, V \neq \emptyset}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{a_{n,\mathrm{U}(a,\sigma,t)\,\dot{\setminus}\, V}} \langle\langle\langle x, V \,\dot{\cup}\, \mathrm{U}(a, \sigma, t), y'\rangle\rangle, \sigma', t\rangle}\ {}_{37}$$

$$\frac{\langle y, \sigma, t\rangle \xrightarrow[k]{a_{n,W}} \langle y', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) \,\dot{\setminus}\, V = \emptyset}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{a} \langle\langle\langle x, V, y'\rangle\rangle, \sigma', t\rangle}\ {}_{38}$$

$$\frac{\langle y, \sigma, t\rangle \xrightarrow[k]{a} \langle y', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) \,\dot{\setminus}\, V \neq \emptyset}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{a_{0,\mathrm{U}(a,\sigma,t)\,\dot{\setminus}\, V}} \langle\langle\langle x, V \,\dot{\cup}\, \mathrm{U}(a, \sigma, t), y'\rangle\rangle, \sigma', t\rangle}\ {}_{39}$$

$$\frac{\langle y, \sigma, t\rangle \xrightarrow[k]{a} \langle y', \sigma', t\rangle, \quad \mathrm{U}(a, \sigma, t) \,\dot{\setminus}\, V = \emptyset}{\langle\langle\langle x, V, y\rangle\rangle, \sigma, t\rangle \xrightarrow[k]{a} \langle\langle\langle x, V, y'\rangle\rangle, \sigma', t\rangle}\ {}_{40}$$

Table 9.10: Deduction rules for the transactional operators.

The deduction rules for the transactional operators are summarised in Table 9.10. They are based on the rules as given in Table 8.2 on page 88 on which the findings given in Section 8.5 are applied.

Rule 29 handles the case of a rollback of a transaction that has executed actions. The rule itself is introduced in Section 8.5.2 and can be found on page 105.

Rules 30 and 31 are concerned with termination of transactional processes: a transaction ends by committing, causing other transactions to become unlocked with respect to variables locked by the ending transaction. After committing, the process can successfully terminate.

Deduction rules 32 and 36 deal with the case of executing an unlocking action which comes from a nested transaction. The valuation set of unlocked variables (the W in $\mathcal{U}_W$) does *not* pass transactional operators, so this set is replaced by the empty set. This set is already rebuilt when locking actions from within the same nested transactions passed the transactional operator.

The semantics of execution of lockable actions that come from within nested transactions is given by rules 33, 37 and 38. As is the case for the unlocking actions, the valuation sets of locking variables (the W in $a_{n,W}$) do *not* pass the transactional operators. These sets are also rebuilt using the current state. Deduction rule 33 states that if a locking action is executed from within a transaction, then it is also executed by the transaction, however with its locking variable set replaced by a set containing those variables that are updated by executing the action in the current state. In rule 37 it is stated that if a locking action is executed and it *does* affect valuations in the current state which are not already affected by the transaction, then it is also executed by the constructed process, having its locking set replaced by the set of affected valuations. Otherwise, the action is no longer a locking action as stated in rule 38.

The deduction rules for internal and communication actions, rules 34, 35, 39 and 40, can be compared with those for the locking actions. Rules 34 and 35 are introduced on page 108, where we explain that we have to split rule 13 in Table 8.2 into two rules. Rules 35 and 40 state that is an action from within a transaction cannot affect the state, then it can simply be executed by the transaction. If, however, the state might change by executing the action, then the action becomes a locking/lockable action as stated in rules 34 and 39.

$$\frac{\langle x,\sigma,t\rangle \downarrow}{\langle [x]_V,\sigma,t\rangle \downarrow}\,41 \qquad \frac{\langle x,\sigma,t\rangle \xrightarrow[k]{\mathcal{U}_V} \langle x',\sigma',t\rangle}{\langle [x]_W,\sigma,t\rangle \xrightarrow[k]{\mathcal{U}_V} \langle [x']_W,\sigma',t\rangle}\,42$$

$$\frac{\langle x,\sigma,t\rangle \xrightarrow[k]{a_{n,V}} \langle x',\sigma',t\rangle}{\langle [x]_W,\sigma,t\rangle \xrightarrow[k]{a_{n+|V \cap W|,V}} \langle [x']_W,\sigma',t\rangle}\,43 \qquad \frac{\langle x,\sigma,t\rangle \xrightarrow[k]{a} \langle x',\sigma',t\rangle}{\langle [x]_W,\sigma,t\rangle \xrightarrow[k]{a} \langle [x']_W,\sigma',t\rangle}\,44$$

$$\frac{\langle x,\sigma,t\rangle \downarrow}{\langle \lfloor x\rfloor_W,\sigma,t\rangle \downarrow}\,45 \qquad \frac{\langle x,\sigma,t\rangle \xrightarrow[k]{\mathcal{U}_V} \langle x',\sigma',t\rangle}{\langle \lfloor x\rfloor_W,\sigma,t\rangle \xrightarrow[k]{\mathcal{U}_V} \langle \lfloor x'\rfloor_W,\sigma',t\rangle}\,46$$

$$\frac{\langle x,\sigma,t\rangle \xrightarrow[k]{a_{n,V}} \langle x',\sigma',t\rangle, \quad (V \cap W = \emptyset \vee n = 0)}{\langle \lfloor x\rfloor_W,\sigma,t\rangle \xrightarrow[k]{a_{n,V}} \langle \lfloor x'\rfloor_W,\sigma',t\rangle}\,47$$

$$\frac{\langle x,\sigma,t\rangle \xrightarrow[k]{a_{n,V}} \langle x',\sigma',t\rangle, \quad (V \cap W \neq \emptyset \wedge n > 0)}{\langle \lfloor x\rfloor_W,\sigma,t\rangle \xrightarrow[k]{a_{n-|V \cap W|,V}} \langle \lfloor x'\rfloor_W,\sigma',t\rangle}\,48$$

$$\frac{\langle x,\sigma,t\rangle \xrightarrow[k]{a} \langle x',\sigma',t\rangle}{\langle \lfloor x\rfloor_W,\sigma,t\rangle \xrightarrow[k]{a} \langle \lfloor x'\rfloor_W,\sigma',t\rangle}\,49$$

Table 9.11: Deduction rules for the locking and unlocking operators.

The deduction rules for the locking and unlocking operators, given in Table 9.11, can be compared with the rules as given in Table 8.4 on page 90. Again, the findings in Section 8.5 are applied to them.

Rules 41 and 45 state that termination is not affected by the locking and unlocking operators. Also, non-lockable actions are influenced by neither the locking (rules 42 and 44) nor the unlocking operator (rules 46 and 49).

In rule 43 it is stated that locking a lockable action causes its lock counter to be increased by the number of variables that overlap in the valuation set containing valuations (possibly) affected by the action and the set of valuations locking the action.

$$\frac{\langle x, \sigma, t\rangle \downarrow, \quad \langle y, \sigma, t\rangle \downarrow}{\langle x \parallel y, \sigma, t\rangle \downarrow}\ {}_{50}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow[k]{\mathcal{U}_V} \langle x', \sigma', t\rangle}{\langle x \parallel y, \sigma, t\rangle \xrightarrow[k0]{\mathcal{U}_V} \langle x' \parallel \lfloor y \rfloor_V, \sigma', t\rangle, \quad \langle y \parallel x, \sigma, t\rangle \xrightarrow[k1]{\mathcal{U}_V} \langle \lfloor y \rfloor_V \parallel x', \sigma', t\rangle}\ {}_{51}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow[k]{a_{0,V}} \langle x', \sigma', t\rangle}{\langle x \parallel y, \sigma, t\rangle \xrightarrow[k0]{a_{0,V}} \langle x' \parallel [y]_V, \sigma', t\rangle, \quad \langle y \parallel x, \sigma, t\rangle \xrightarrow[k1]{a_{0,V}} \langle [y]_V \parallel x', \sigma', t\rangle}\ {}_{52}$$

$$\frac{\langle x, \sigma, t\rangle \xrightarrow[k]{a} \langle x', \sigma', t\rangle}{\langle x \parallel y, \sigma, t\rangle \xrightarrow[k0]{a} \langle x' \parallel y, \sigma', t\rangle, \quad \langle y \parallel x, \sigma, t\rangle \xrightarrow[k1]{a} \langle y \parallel x', \sigma', t\rangle}\ {}_{53}$$

Table 9.12: Deduction rules for the parallel composition operator.

Unlocking of lockable actions causes the opposite to happen: the lock counter gets decreased by the number of overlapping variables if the action is locked as stated in rule 48. Rule 47 states that if the action is locking but not locked ($n = 0$), or if no overlap takes place ($V \underset{\bullet}{\cap} W = \emptyset$), then the action is not affected by the unlocking operator.

Table 9.12 contains the deduction rules for the parallel composition operator. These rules can be one-on-one matched to the first four rules given in Table 8.6 on page 91. As can be seen by looking at rules 51 to 53, the session label gets updated depending on the side of the process that executes the action.

If both processes put in parallel terminate, the composed process terminates, as stated in rule 50.

Executing an unlocking action in parallel to another process causes the process put in parallel to be unlocked with respect to the valuations with which the unlocking action is parameterised (rule 51).

A locking action executed in parallel to another process is only allowed if it is not locked, i.e., if its locking counter equals 0. In that case, the process running in parallel gets locked with respect to the valuations that might be affected by the locking action (rule 52).

Rule 53 states that non-locking actions executed in parallel to another process can

$$\frac{\langle [\, u \mapsto \bot \colon \mathbb{U} \mid x \,], \sigma, t \rangle \downarrow}{\langle ?_u x, \sigma, t \rangle \downarrow}\ 54 \qquad \frac{\langle [\, u \mapsto \bot \colon \mathbb{U} \mid x \,], \sigma, t \rangle \xrightarrow[k]{\mathcal{U}_V} \langle [\, u \mapsto \bot \colon \mathbb{U} \mid x' \,], \sigma', t \rangle}{\langle ?_u x, \sigma, t \rangle \xrightarrow[k0]{\mathcal{U}_V} \langle [\, u \mapsto \bot \colon \mathbb{U} \mid x' \,] \parallel \lfloor ?_u x \rfloor_V, \sigma', t \rangle}\ 55$$

$$\frac{\langle [\, u \mapsto \bot \colon \mathbb{U} \mid x \,], \sigma, t \rangle \xrightarrow[k]{a_{0,V}} \langle [\, u \mapsto \bot \colon \mathbb{U} \mid x' \,], \sigma', t \rangle}{\langle ?_u x, \sigma, t \rangle \xrightarrow[k0]{a_{0,V}} \langle [\, u \mapsto \bot \colon \mathbb{U} \mid x' \,] \parallel [?_u x]_V, \sigma', t \rangle}\ 56$$

$$\frac{\langle [\, u \mapsto \bot \colon \mathbb{U} \mid x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto \bot \colon \mathbb{U} \mid x' \,], \sigma', t \rangle}{\langle ?_u x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto \bot \colon \mathbb{U} \mid x' \,] \parallel ?_u x, \sigma', t \rangle}\ 57$$

Table 9.13: Deduction rules for the anonymous replication operator.

simply be executed.

Table 9.13 shows the deduction rules for the anonymous replication operator. These rules correspond to the rules as given in Table 7.3 on page 74. We adapt them here so that they apply to the alphabet of *DiCons*.

Since the anonymous replication operator puts processes in parallel, session labels are updated, as can be seen by looking at rules 55 to 57.

In rule 54 it is stated that if the processes put in parallel terminate, then the constructed process terminates.

Rules 55 and 56 state that execution of unlocking and locking actions causes the processes put in parallel to be unlocked or locked. Otherwise the process is simply forked off and put in parallel as stated in rule 57.

$$\frac{\forall_{c \in G(\sigma,t)} \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle \mathbf{!}_{u \in G} x, \sigma, t \rangle \downarrow} \; 58$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \mathbf{!}_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \parallel \lfloor \mathbf{!}_{u \in G} x \rfloor_V, \sigma', t \rangle} \; 59$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \mathbf{!}_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \parallel [\mathbf{!}_{u \in G} x]_V, \sigma', t \rangle} \; 60$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \mathbf{!}_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \parallel \mathbf{!}_{u \in G} x, \sigma', t \rangle} \; 61$$

$$\frac{\forall_{c \in G(\sigma,t)} \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle \mathbf{!}^H_{u \in G} x, \sigma, t \rangle \downarrow} \; 62$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \mathbf{!}^H_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \parallel \lfloor \mathbf{!}^H_{u \in G} x \rfloor_V, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle} \; 63$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \mathbf{!}^H_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \parallel [\mathbf{!}^H_{u \in G} x]_V, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle} \; 64$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \mathbf{!}^H_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \parallel \mathbf{!}^H_{u \in G} x, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle} \; 65$$

Table 9.14: Deduction rules for the replication operators.

In Table 7.4 on page 75 the deduction rules for the replication operator are given. Table 7.6 on page 77 contains the deduction rules for the extended replication operator. Here, in Table 9.14, we adapt these rules such that they apply to the alphabet of *DiCons*. This results in a case distinction on the type of action executed.

As is the case with the anonymous replication operator, session labels are updated, as can be seen by looking at rules 59 to 61 and rules 63 to 65.

Deduction rules 58 and 62 state that if all processes put in parallel terminate, then the composed process also terminates.

The executing of an unlocking action by one of the processes causes the other processes to be unlocked with respect to the valuations with which the unlocking action is parameterised (rules 59 and 63). The process for the user causing this unlocking event, $c \in \mathbb{U}$, is forked off and continues running in parallel to the replication process. Apart from that, in rule 63 the user is added to the group of registered users H.

Execution of a locking action is similar to the execution of an unlocking action: in rules 60 and 64 it is stated that if a locking, but not locked, action is executed by one of the processes, the other processes are locked with the the valuation set added to the locking action. Again, the interacting user's process is forked off and put in parallel to replication process and registered group H is extended with the user.

For all other actions, the action is executed, the process is forked off (rules 61 and 65) and registered group H is extended with the user.

The deduction rules for the generalised parallel composition operators, given in Table 9.15, can be compared with those for the replication operators as given in Table 9.14. The only difference is the dropping from the users who fork off a process from the domain of the operators: using the generalised parallel composition operators, all users in their domains can only fork off at most one process.

$$\frac{\forall_{c \in G(\sigma,t)}\ \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle \|_{u \in G} x, \sigma, t \rangle \downarrow} \quad 66$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, \lfloor \|_{u \in G \setminus \{c\}} x \rfloor_V, \sigma', t \rangle} \quad 67$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, [\|_{u \in G \setminus \{c\}} x]_V, \sigma', t \rangle} \quad 68$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, \|_{u \in G \setminus \{c\}} x, \sigma', t \rangle} \quad 69$$

$$\frac{\forall_{c \in G(\sigma,t)}\ \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \downarrow}{\langle \|^H_{u \in G} x, \sigma, t \rangle \downarrow} \quad 70$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|^H_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{\mathcal{U}_V} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, \lfloor \|^H_{u \in G \setminus \{c\}} x \rfloor_V, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle} \quad 71$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|^H_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a_{0,V}} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, [\|^H_{u \in G \setminus \{c\}} x]_V, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle} \quad 72$$

$$\frac{c \in G(\sigma,t), \quad \langle [\, u \mapsto c : \mathbb{U} \,|\, x \,], \sigma, t \rangle \xrightarrow[k]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,], \sigma', t \rangle}{\langle \|^H_{u \in G} x, \sigma, t \rangle \xrightarrow[k0]{a} \langle [\, u \mapsto c : \mathbb{U} \,|\, x' \,] \,\|\, \|^H_{u \in G \setminus \{c\}} x, \sigma'[H(\sigma',t) \cup \{c\}/H], t \rangle} \quad 73$$

Table 9.15: Deduction rules for the generalised parallel composition operators.

9.5 Example

In this section we give a small specification to show the expressiveness of *DiCons*. The application specified here corresponds to the Internet vote example given in Section 3.4.3 on page 22 to a great extent.

The specification is given in Figure 9.1. We shortly discuss the elements of the specification. In lines 1 and 2 we specify who can start the initialisation of a vote. In this specification we allow anyone to start a vote, as can be seen by the initialisation of Initiators by $\mathbb{U}$. Note that we do not use the anonymous replication operator ? since we want the results to be sent to the initiator. So the initiator must be known to the application. By initialising the group of possible users by universe $\mathbb{U}$ we achieve this intended semantics.

Lines 8 to 12 specify the initialisation phase which mostly corresponds to the specification given in Figure 3.11 on page 23. Before starting the initialisation phase, local variables used for saving application data are declared in line 4. The candidates (line 8), voters (line 9) and a deadline (line 10) are provided by the initiator. In line 11 the vote is started, which is confirmed in line 12. By looking at the communication primitives it can be easily seen which interactions form a session.

Next, in line 14, the results are declared. The results map the candidates to a number representing the number of voters who voted for the candidate. In line 15 they are all mapped to 0:

$$\text{effect}(\text{initialise}(results), \sigma, t) \quad = \quad \sigma' \text{ where } \forall c \in Candidates(\sigma')\ results[c](\sigma') = 0$$

In lines 19 to 27 the vote phase is specified. All voters receive an invitation (line 19). The remainder of this phase, lines 21 to 27, can be compared with the MSC specification in Figure 3.12 on page 24. Using the conditional disrupt operator, we specify that as long as the deadline is not reached ($now < deadline$) all voters are allowed to execute the vote process once. A vote is started by identification of the voter (using the generalised parallel composition operator $\|$ in line 22) after which he selects a candidate from the group of candidates in line 24. This candidate gets his votes increased in line 25. If something goes wrong during the vote it must be rolled back, so we turn the vote process into a transaction using the transactional brackets (lines 23 and 27).

Finally, after passing of the deadline or if all voters have voted, the winner is calculated in line 32 and sent to the initiator in line 33. This phase can be compared with the MSC in Figure 3.13 on page 24. To complete the example we give the effect of the calc_winners function:

```
1  [ Initiators ↦ 𝕌 : 𝔾 |
2    !_{initiator∈Initiators}
3
4      ⟨⟨[ Candidates ↦⊥: 𝔾 | [ Voters ↦⊥: 𝔾 | [ deadline ↦⊥: 𝔗 |
5
6        // initialisation phase
7
8        initiator ⇉ set_candidates(ε; Candidates) ·
9        initiator ⇉ set_voters(ε; Voters) ·
10       initiator ⇉ set_deadline(ε; deadline) ·
11       initiator ⇉ start(ε; ε) ·
12       initiator ⇄ ok(ε) ·
13
14       [ results ↦⊥: array Candidates of ℕ |
15         initialise(results) ·
16
17         // vote phase
18
19         ||_{v∈Voters} v ← vote_email(Candidates, deadline) ·
20
21         now < deadline ▷—
22           ||_{v∈Voters}
23             ⟨⟨[ candidate ↦⊥: 𝕌 |
24               v ⇉ vote(Candidates; candidate) ·
25               results[candidate] := results[candidate] + 1 ·
26               v ⇄ done(candidate)
27             ]⟩⟩ ·
28
29         // calculation phase
30
31         [ winner ↦⊥: 𝔾 |
32           winners := calc_winners(results) ·
33           initiator ← send_results(winners)
34         ]
35
36       ]
37
38     ]]]⟩⟩
39
40 ]
```

Figure 9.1: A *DiCons* specification of an Internet vote.

$$\begin{aligned}&\text{effect}(winners := \text{calc\_winners}(results), \sigma, t) \quad = \\ &\quad \sigma' \text{ where } winners(\sigma') = \{\, c \in Candidates(\sigma') \mid \\ &\qquad\qquad \forall d \in Candidates(\sigma')\ results[c](\sigma') \geq results[d](\sigma') \,\}\end{aligned}$$

9.5.1 Properties

In this section we specify and verify some properties of Internet applications. Properties can be divided into two classes: general properties and application-dependent properties. General properties are properties of all Internet applications we focus on where application-dependent properties are subject to a specific application.

Before we give these properties we explain the notion of blocking, which is an important concept with respect to (multi-client) Internet applications in general. Apart from that, we make some assumptions on the clients that make use of the application and the robustness of the server.

Blocking

A session in a process is blocked by a user if it is waiting for that user to interact with it, i.e., if it is waiting for the user to send a new request. In Section 4.3 we explained the way in which interaction primitives are constructed of HTTP requests and responses. From that point of view, a session in a process can be blocked if it is waiting for the user to send a URL request or to fill in and submit a Web form. Note that blocking has nothing to do with the locking mechanism of transactional processes: blocking is concerned with user interaction where locking handles valuation updates.

To be able to formally specify the blocking concept, we first introduce the way we reason about traces. This is based on [BB88]. As explained in Section 9.4, we label our transitions using several action labels. Apart from the action labels we add a session label to the transitions to be able to keep the actions within different sessions separated. In our semantics, a trace is a sequence of these action label/session label combinations which correspond to actions that subsequently can take place starting in some initial state.

Definition 9.5.1 *(Trace) Let a be an action label ($a \in \mathbb{A}_l$) and k be a session label ($k \in \mathbb{K}$). A trace α is then defined by*

$$\begin{array}{rcll}\alpha & =_{\text{def}} & \varepsilon & \textit{the empty trace} \\ & \mid & \langle a, k\rangle ; \alpha & \textit{a nonempty trace}\end{array}$$

We have some general functions on traces for determining the head, tail and length of a trace. Note that both the head and tail functions are partial.

Definition 9.5.2 *(General functions on traces) Let α be a trace, $a \in \mathbb{A}_l$ be an action label and $k \in \mathbb{K}$ be a session label. Then,*

$$\begin{aligned} \text{head}(\langle a,k\rangle;\alpha) &= \langle a,k\rangle \\ \text{tail}(\langle a,k\rangle;\alpha) &= \alpha \\ |\alpha| &= \begin{cases} 0 & \textit{if } \alpha = \varepsilon \\ 1 + |\text{tail}(\alpha)| & \textit{otherwise} \end{cases} . \end{aligned}$$

Apart from these general functions we have some specific functions in combination with our semantics. The first function we introduce is for determining whether a user is blocking a trace within a particular session. This means that the trace starts with a request label.

Definition 9.5.3 *(Blocking) Let α be a trace, c be a (constant) user, $c \in \mathbb{U}$, and k be a session label, $k \in \mathbb{K}$. The property of a trace α being* blocked *by a user c in session k is then defined by*

$$\boxed{\alpha}_{c,k} \quad \equiv \quad \exists \vec{\imath} \exists \vec{d}\, \text{head}(\alpha) = \langle \text{req}.c.\vec{\imath}.\vec{d}, k\rangle$$

Given these functions we can only reason about traces. However we want to reason about processes in a given state at a given time. Therefore we want to transform a process into a (possibly infinite) set of feasible traces. We do this using the traces function which returns all traces that can take place starting a process in a given state at a given time. Traces can be infinitely long if the process does not terminate.

Definition 9.5.4 *(Traces of a process) Let $x, x' \in \mathbb{X}$ be processes, $\sigma, \sigma' \in \mathbb{S}$ be states, $t, t' \in \mathfrak{T}$ be time stamps, $a \in \mathbb{A}_l$ be an action label and $k \in \mathbb{K}$ be a session label. At time t in state σ, the set of traces of process x is then defined by*

$$\begin{aligned} \text{traces}(\langle x,\sigma,t\rangle) \quad = \quad & \{\varepsilon \mid \langle x,\sigma,t\rangle \downarrow \ \vee \neg\exists a{\in}\mathbb{A}_l \exists k{\in}\mathbb{K}\ \langle x,\sigma,t\rangle \xrightarrow[k]{a} \langle x',\sigma',t'\rangle\} \cup \\ & \{\langle a,k\rangle;\alpha \mid \langle x,\sigma,t\rangle \xrightarrow[k]{a} \langle x',\sigma',t'\rangle \wedge \alpha \in \text{traces}(\langle x',\sigma',t'\rangle)\} . \end{aligned}$$

A (session in a) process is blocked by a user if, in a given state at a given time, a trace exists that is blocked by that user.

Definition 9.5.5 *(Blocking of a process) Let $x \in \mathbb{X}$ be a process, $\sigma \in \mathbb{S}$ be a state, $t \in \mathfrak{T}$ be a time stamp and $c \in \mathbb{U}$ be a constant user. At time t in state σ, a process x being blocked by*

user c is then defined by

$$\boxed{\langle x, \sigma, t\rangle}_c \quad \equiv \quad \exists \alpha \in \text{traces}(\langle x, \sigma, t\rangle) \exists k \in \mathbb{K}\ \boxed{\alpha}_{c,k}\ .$$

Apart from the application being blocked, a user can be blocked by an application. This takes place between the sending of a request and the receiving of a response. So a user is blocked if he is waiting for a reaction, i.e. a response from the server.

If a given specification answers the syntax and static assumptions, this means that a user is blocked by a process in a given state at a given time if there is a trace of that process in which the first occurrence of the receiving of a reactive or session-oriented message in a session precedes the first occurrence of the sending of a request in that session. To determine the number of steps before the n-th occurrence of an element in a set of given transition takes place, we make use of the $\#_n$ operator. Let A be a set of elements in a trace, $A \subseteq \mathbb{A}_l \times \mathbb{K}$. Then,

$$\#_n(A, \alpha) \quad = \quad \begin{cases} \infty & \text{if } \alpha = \varepsilon \\ 1 & \text{if } n = 1 \ \wedge\ \text{head}(\alpha) \in A \\ 1 + \#_{n-1}(A, \text{tail}(\alpha)) & \text{if } n > 1 \ \wedge\ \text{head}(\alpha) \in A \\ 1 + \#_n(A, \text{tail}(\alpha)) & \text{otherwise .} \end{cases}$$

Again, we first specify a function over traces which we use for specifying the function over processes.

Definition 9.5.6 *(User-blocking for traces) Let α be a trace, $c \in \mathbb{U}$ be a (constant) user and $k \in \mathbb{K}$ be a session label. The property of a user c being* blocked *in session k by a trace α is then defined by*

$$\boxed{c}_{\alpha,k} \quad \equiv \quad \exists \vec{d} \exists \vec{\imath} \exists \vec{d'}\ \ \#_1(\{\langle \text{resp}.c.m.\vec{d}, k\rangle \mid m \in \mathbb{M}\}, \alpha) < \#_1(\{\langle \text{req}.c.\vec{\imath}.\vec{d'}, k\rangle\}, \alpha)$$

Definition 9.5.7 *(User-blocking for processes) Let $x \in \mathbb{X}$ be a process, $\sigma \in \mathbb{S}$ be a state, $t \in \mathfrak{T}$ be a time stamp and $c \in \mathbb{U}$ be a (constant) user. At time t in state σ, the property of user c being* blocked *by process x is then defined by*

$$\boxed{c}_{\langle x, \sigma, t\rangle} \quad \equiv \quad \exists \alpha \in \text{traces}(\langle x, \sigma, t\rangle) \exists k \in \mathbb{K}\ \boxed{c}_{\alpha,k}$$

In order to prove properties of processes, it would be very useful to introduce the notion of invariants for traces and therefore for processes. To be able to verify properties after execution of each action means that we want to be able to verify properties at all semicolons in a trace. This can be done by proving that the property holds at the beginning of all possible tails of a trace. By using the tails function we can generate this set of tails.

$$\text{tails}(\alpha) \quad = \quad \begin{cases} \{\alpha\} & \text{if } \alpha = \varepsilon \\ \{\alpha\} \cup \{\text{tails}(\text{tail}(\alpha))\} & \text{otherwise} \end{cases}$$

We use this function for defining the properties below.

Assumptions

Since we have a notion of blocking, we have to make some assumptions on interactions to be able to define suitable properties of Internet applications. Furthermore, we make an assumption on the robustness of the server.

Client interactivity Web forms sent to a client are finally filled in and submitted. We need this assumption to prevent the application from being blocked.

Server robustness The server on which the application runs does not crash or shut down as long as the application is active.

Note that we only make assumptions with respect to the client and the robustness of the server. We do not make any assumptions on aspects of the application itself. Since we want to formally verify specifications of Internet applications we cannot and will not introduce these properties in our semantics. This would make the semantics unnecessarily complex.

General properties

In this section we define the independent responsiveness property and show how to prove this property for a simple example specification to show the usefulness of *DiCons*. This property is in general a desirable property of Internet applications.

Users who interact with the central system can block the application as explained before. However, we do not want users to be able to block other user's sessions.

Definition 9.5.8 *(Independent responsiveness) Let $x \in \mathbb{X}$ be a process, $\sigma \in \mathbb{S}$ be a state and $t \in \mathfrak{T}$ be a time stamp. At time t in state σ, the independent responsiveness IR property for process x is defined by*

$$\begin{aligned} &IR(\langle x, \sigma, t\rangle) \quad \equiv \\ &\quad \forall \alpha \in \text{tails}(\text{traces}(\langle x, \sigma, t\rangle))\ \neg \exists c \in \mathbb{U} \exists d \in \mathbb{U} \exists k \in \mathbb{K}\ (c \neq d \wedge \boxed{\alpha}_{c,k} \wedge \boxed{d}_{\alpha,k}) \end{aligned}$$

So this property states that for all traces in the set of tails of traces of the process it is not the case that a trace is blocked by a user within a session and at the same moment within the same session a different user is blocked by the trace. If this property does not hold it can be the case that a user submitting a form has to wait for another user to interact with the application before he receives a response to his submission. Below, we give an example of a specification in which independent responsiveness does *not* hold:

$$[\ u_1 \mapsto c_1 : U\ |[\ u_2 \mapsto c_2 : U\ |[\ n : \mathbb{N}\ |[\ m : \mathbb{N}\ |$$
$$u_1 \rightleftarrows \text{give}(\varepsilon;n)\ \cdot\ u_2 \rightleftarrows \text{give}(\varepsilon;m)\ \cdot\ u_1 \leftrightarrows \text{return}(m)\ \cdot\ u_2 \leftrightarrows \text{return}(n)$$
$$]]]]$$

As can be seen, user u_1 is blocked by user u_2, who should first provide value m before it can be sent to u_1.

If this property holds for the initial state of an application it follows from the use of the tails and traces functions that it holds in all possible states during the execution of the application. We can make use of a test generator and execution tool (e.g. using the TorX tools [BFV+99]) for testing applications with respect to this property. Using the voting example given on page 137 results in a set of traces which is too large to prove independent responsiveness by hand. Therefore we give a small application and prove that it answers the independent responsiveness property.

In the following example two users can simultaneously submit an integer number via a Web form which subsequently is returned. There may be interference so that the returned number may be the number submitted by the other user.

$$[\ u_1 \mapsto c_1 : U\ |[\ u_2 \mapsto c_2 : U\ |[\ n : \mathbb{N}\ |$$
$$u_1 \rightleftarrows \text{give}(\varepsilon;n) \cdot u_1 \leftrightarrows \text{return}(n)\ \|\ u_2 \rightleftarrows \text{give}(\varepsilon;n) \cdot u_2 \leftrightarrows \text{return}(n)$$
$$]]]$$

Intuitively the IR property holds since the two sessions of users c_1 and c_2 run in parallel and are independent of each other with respect to the user interactions.

Theorem 9.5.9 *Let $x \in \mathbb{X}$ be the process defined above and $t_0 \in \mathfrak{T}$ be the initial time. Then, $IR(\langle x, \varepsilon, t_0\rangle)$ holds.*

Proof Since there are only two users, c_1 and c_2, the IR property of the initial state of x, $IR(\langle x, \varepsilon, t_0\rangle)$, can be reduced to

$$\forall \alpha \in \text{tails}(\text{traces}(\langle x, \sigma, t\rangle))\ \neg\exists k \in \mathbb{K}\ (\boxed{\alpha}_{c_1,k} \wedge \boxed{c_2}_{\alpha,k}) \vee (\boxed{\alpha}_{c_2,k} \wedge \boxed{c_1}_{\alpha,k})$$

Since the merge operator $\|$ is, apart from the session label, commutative, this application is symmetric with respect to users u_1 and u_2. So by proving

$$\forall \alpha \in \text{tails}(\text{traces}(\langle x, \varepsilon, t_0\rangle))\ \neg\exists k \in \mathbb{K}\ (\boxed{\alpha}_{c_1,k} \wedge \boxed{c_2}_{\alpha,k})$$

and using symmetry, we have proved $IR(\langle x, \varepsilon, t_0\rangle)$.

Suppose that IR does *not* hold for $\langle x, \varepsilon, t_0\rangle$. Then there must be a trace α in the set of possible traces traces($\langle x, \varepsilon, t_0\rangle$) where a k exists such that $(\boxed{\alpha}_{c_1,k} \wedge \boxed{c_2}_{\alpha,k})$ holds.

Suppose we have an α and a k such that $\boxed{\alpha}_{c_1,k}$. Then, it follows from deduction rule 53 in Table 9.12 that $k = 0$. This means that $\boxed{c_2}_{\alpha,0}$ must hold. So

$$\exists \vec{d} \exists \vec{y} \exists \vec{d'} \; (\#_1(\{\langle \text{resp}.c_2.m.\vec{d}, 0\rangle \mid m \in \mathbb{M}\}, \alpha) < \#_1(\{\langle \text{req}.c_2.\vec{y}.\vec{d'}, 0\rangle\}, \alpha))$$

However, if there is an element $\langle \text{resp}.c_2.m.\vec{d}, 0\rangle$ or $\langle \text{req}.c_2.\vec{y}.\vec{d'}, 0\rangle$, it follows from deduction rule 53 in Table 9.12 that there must be a subprocess on the left-hand side of the merge operator $\parallel$ that can do a step where the user is c_2. This is not the case, so we get $\infty < \infty$ which proves false. So $\neg \boxed{c_2}_{\alpha,0}$. And therefore there is no k such that $\boxed{\alpha}_{c_1,k} \wedge \boxed{c_2}_{\alpha,k}$. So $IR(\langle x, \varepsilon, t_0\rangle)$ holds. □

We can give a similar proof for the Internet vote example given in Figure 9.1. The parallel execution of the vote phase proves to have the independent responsiveness property.

Another general property that must hold is the property that sessions must be valid, i.e., that, only considering interaction primitives, all sessions must start with a reactive pull, possibly followed by session-oriented pulls, and end with a session-oriented push. We call this property *session validity*. Operationally, this means that in all sessions, all responses must be preceded by a corresponding request. Since the transactional operator might break this behaviour by rolling back between requests and responses, we do not consider traces where a rollback takes place.

Definition 9.5.10 *(Session validity) Let $x \in \mathbb{X}$ be a process, $\sigma \in \mathbb{S}$ be a state and $t \in \mathfrak{T}$ be a time stamp. At time t in state σ, the session validity SV property for process x is then defined as follows:*

$$\begin{aligned}
&SV(\langle x, \sigma, t\rangle) \quad \equiv \\
&\quad \forall \alpha \in \text{traces}(\langle x, \sigma, t\rangle)\; \forall c \in \mathbb{U}\; \forall k \in \mathbb{K}\; \forall n \geq 1 \; (\\
&\qquad \#_1(\{\langle \mathcal{R}_V, k\rangle \mid V \subseteq \mathbb{V}\}, \alpha) = \infty \\
&\qquad \wedge \\
&\qquad \#_n(\{\langle \text{req}.c.\vec{\imath}.\vec{d}, k\rangle \mid i_1, \ldots, i_p \in \mathbb{P}_i, \vec{d} \in \text{type}(\vec{\imath})\}, \alpha) < \infty \\
&\quad) \Rightarrow (\\
&\qquad \#_n(\{\langle \text{req}.c.\vec{\imath}.\vec{d}, k\rangle \mid i_1, \ldots, i_p \in \mathbb{P}_i, \vec{d} \in \text{type}(\vec{\imath})\}, \alpha) \\
&\qquad < \\
&\qquad \#_n(\{\langle \text{resp}.c.m.\vec{o}, k\rangle \mid m \in \mathbb{M}, o_1, \ldots, o_q \in \mathbb{P}_o\}, \alpha) \\
&\qquad \wedge \\
&\qquad \#_n(\{\langle \text{resp}.c.m.\vec{o}, k\rangle \mid m \in \mathbb{M}, o_1, \ldots, o_q \in \mathbb{P}_o\}, \alpha) \\
&\qquad < \\
&\qquad \#_{n+1}(\{\langle \text{req}.c.\vec{\imath}.\vec{d}, k\rangle \mid i_1, \ldots, i_p \in \mathbb{P}_i, \vec{d} \in \text{type}(\vec{\imath})\}, \alpha) \\
&\quad)
\end{aligned}$$

So session validity states that in every session in which no rollback has occurred, for each user, the n-th occurrence of a response to a user in a session is preceded by the n-th occurrence of a request by that user in the session and vice versa, the n-th occurrence of a request by a user in a session is followed by the n-th occurrence of a response to that user in the session. Furthermore, if another request is sent, then this sending takes place after receiving the response. So session validity holds for an alternating occurrence of requests and responses in all traces in the process.

An example of a process with an invalid session is given below:

$$
\begin{array}{l}
[\ u \mapsto c : U \mid [\, n : \mathbb{N} \mid \\
\quad u \rightleftarrows \text{finished}(\varepsilon) \ \cdot\ u \rightleftarrows \text{give}(\varepsilon;n) \\
]\,]
\end{array}
$$

Execution of this process can contain the following trace, for which the SV property not holds:

$$\langle \text{resp}.c.\text{finished}.\varepsilon, \lambda\rangle; \langle \text{req}.c.\varepsilon.\varepsilon, \lambda\rangle; \langle \text{resp}.c.\text{give}.\varepsilon, \lambda\rangle; \langle \text{req}.c.n.1, \lambda\rangle$$

As can easily be seen by looking at the specification of the Internet vote in Figure 9.1, session validity holds since all communication primitives are used correctly.

Theorem 9.5.11 *Let x be the process defined in Figure 9.1 on page 137 and $t_0 \in \mathfrak{T}$ be a time stamp. Then $SV(\langle x, \varepsilon, t_0\rangle)$ holds.*

Proof To informally prove $SV(\langle x, \varepsilon, t\rangle)$ we take into account the process for only one of the instances, i.e., for only one of the initiators $initiator \in Initiators$. Since for all other instances the session labels differ from those in the process taken into account, proving it for one instance proves it for all instances. Let k be the process identifier of the process for the chosen initiator $initiator \in Initiators$. We split the process into three subprocesses, viz. the initialisation, vote and calculation phase:

- *initialisation phase*
  The set of trace of the initialisation phase, restricted to the request and response actions, is:

$$
\begin{array}{ll}
\{\ \langle \text{req}.initiator.\varepsilon.\varepsilon, k\rangle; & \langle \text{resp}.initiator.\text{set\_candidates}.\varepsilon, k\rangle; \\
\quad \langle \text{req}.initiator.Candidates.C, k\rangle; & \langle \text{resp}.initiator.\text{set\_voters}.\varepsilon, k\rangle; \\
\quad \langle \text{req}.initiator.Voters.V, k\rangle; & \langle \text{resp}.initiator.\text{set\_deadline}.\varepsilon, k\rangle; \\
\quad \langle \text{req}.initiator.deadline.d, k\rangle; & \langle \text{resp}.initiator.\text{start}.\varepsilon, k\rangle; \\
\quad \langle \text{req}.initiator.\varepsilon.\varepsilon, k\rangle; & \langle \text{resp}.initiator.\text{ok}.\varepsilon, k\rangle \\
\mid\ C \in \mathbb{G},\ V \in \mathbb{G},\ d \in \mathfrak{T} & \\
\}
\end{array}
$$

As can be seen by looking at the traces given above, all requests and responses occur in an alternating way, starting with a request and ending with a response. So SV holds for the initialisation phase.

- *vote phase*
  In the vote phase, multiple users vote in parallel. Since we make use of the generalised parallel composition operator, all processes split off for the users in group $Voters$ get a unique session label. Therefore, all traces restricted to requests and responses *and* restricted to one session label give the operational semantics of process

$$
\begin{array}{l}
[\, v \mapsto c : \mathbb{U} \mid \\
\quad \langle\!\langle [\, candidate \mapsto \bot : \mathbb{U} \mid \\
\qquad v \rightleftarrows \mathrm{vote}(Candidates; candidate) \cdot \\
\qquad v \leftrightarrows \mathrm{done}(candidate) \\
\quad]\rangle\!\rangle \\
\,]
\end{array}
$$

  for a user $c \in Voters$. As can be easily seen, this also results in a trace of alternating requests and responses, viz.

$$
\begin{array}{ll}
\{\ \langle \mathrm{req}.c.\varepsilon.\varepsilon, k\rangle; & \langle \mathrm{resp}.c.\mathrm{vote}.Candidates, k\rangle; \\
\quad \langle \mathrm{req}.c.candidate.d, k\rangle; & \langle \mathrm{resp}.c.\mathrm{done}.candidate, k\rangle; \\
\mid\ d \in Candidates & \\
\}
\end{array}
$$

  for voter $c \in Voters$ and session label $k \in \mathbb{K}$ of the voter's session.

- *calculation phase*
  As can be concluded, in the traces of the calculation phase no request and responses occur, so SV holds since all n-th occurrence operations evaluate to infinity.

Since all phases are combined into a process using the sequential composition operator, no interference of the subtraces is possible. Therefore, all traces of the process given in the example Figure 9.1 show alternating request response behaviour and thus SV holds for the process. □

III

Tools and Applications

10

Conformance Testing of Internet Applications

In this chapter, we adapt and extend the theories used in the general framework of automated software testing in such a way that they become suitable for black-box conformance testing of thin client Internet applications. That is, we automatically test whether a running Internet application conforms to its formal specification. The actual implementation of the application is not taken into account, only its externally observable behaviour, i.e., the requests received and responses sent. In this chapter, we show how to formally model this behaviour using so-called multi request/response transition systems and we explain how such formal specifications can serve as a basis for the automatic conformance testing of Internet applications. In the end of the chapter we show how *DiCons* specifications can be used as a basis for the testing. Parts of this chapter are presented in [BM03].

10.1 Introduction

As mentioned in Chapter 1, the complexity of Internet applications increases fast, which leads to a growing amount of errors (see e.g. [Neu05]). This increasing number of errors asks for better testing of the applications and, preferably, this testing should be automated.

Research has been done in the field of automated testing of applications that are not based on Internet communication. A nice overview can be found in [BT00]. In this chapter, we adapt and extend the theories used in the general framework of

automated software testing in such a way that they become suitable for the testing of Internet applications.

We focus on black-box conformance testing of thin client Internet applications. That is, given a running application and a (formal) specification, our goal is to automatically test whether the implementation of the application conforms to the specification. Black box testing means that the actual implementation of the application is not taken into account but only its externally observable behaviour: We test *what* the application does, *not how* it is done. Interaction with the application takes place using the interface that is available to normal users of the application. In this case, the interface is based on communication via the Internet using the HTTP protocol [FGM+99].

As a start, in Section 10.2 we introduce how we plan to automatically test Internet applications. In Section 10.3, we give a short introduction to conformance testing. In Section 10.4 we describe the formalism we make use of for this automatically testing. This formalism serves as a basis for the definition of the conformance relation in Section 10.5 and the generation of test suites in Section 10.6. To show the usefulness of the framework, we give a practical example in Section 10.7. In Section 10.8, we show how *DiCons* specifications can be used as a basis for generation of test suites. To complete this chapter, we discuss related work in Section 10.9 and draw some final conclusions in Section 10.10.

10.2 Testing of Internet Applications

In Section 3.3, we compared Internet applications to window-based applications. We concluded that the main differences between Internet-based and window-based applications are the failing of clients and Web servers, the failing of communication and overtaking of messages between clients and the application and the dependency on third parties. Furthermore, Internet applications are request/response based where window-based applications interact with the clients using a (graphical) user interface. Finally, most Internet applications focus on parallel communication with more than one client. Since multiple clients can share a common state space, testing Internet applications is basically different from testing window-based applications. Window-based applications are mostly based on single user interaction. More differences between Web-based and window-based systems can be found in e.g. [RFPG96].

First, we informally show how implementations of these applications can be tested. We focus on black-box testing, restricting ourselves to *dynamic testing*. This means that the testing consists of really executing the implemented system. We do this by simulating real-life interaction with the applications, i.e. by simulating the clients that interact with the application. The simulated clients interact in a similar way as

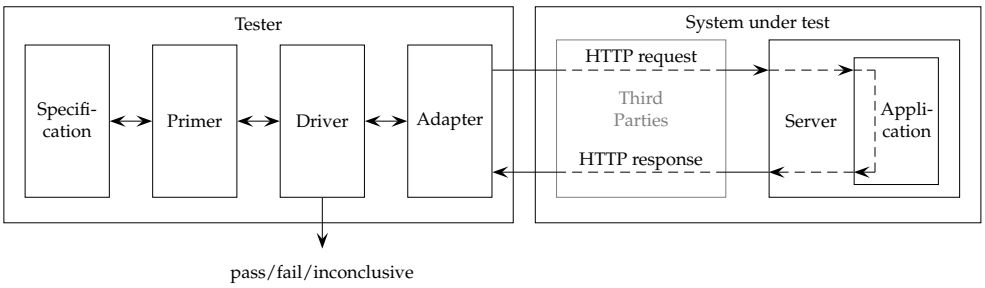


Figure 10.1: Automatic testing of Internet applications.

real-life clients would do. In this way, the application cannot distinguish between a real-life client and a simulated one. See Figure 10.1 for a schematic overview of the test environment.

We make use of a tester which generates requests and receives responses. This is called *test execution*. By observing the responses, the tester can determine whether they are expected responses in the specification. If so, the implementation passes the test, if not, it fails.

The tester itself consists of four components, based on [BFV+99]:

Specification The specification is the formal description of how the application under test is expected to behave.

Primer The primer determines the requests to be sent by inspecting the specification and the current state the test is in. So the primer interacts with the specification and keeps track of the state of the test. Furthermore, the primer checks whether responses received by the tester are expected responses in the specification at the state the test is in.

Driver The driver is the central unit, controlling the execution of the tests. This component determines what actions to execute. Furthermore, the verdict whether the application passes the test is also computed by the driver.

Adapter The adapter is used for encoding abstract representations of requests into HTTP requests and for decoding HTTP responses into abstract representations of these responses.

While executing a test, the driver determines if a request is sent or a response is checked. If the choice is made to send a request, the driver asks the primer for a correct request, based on the specification. The request is encoded using the adapter and sent to the application under test. If the driver determines to check a response, a response is decoded by the adapter. Next, the primer is asked whether the response

is expected in the specification. Depending on the results, a verdict can be given on the conformance of the implementation to its specification.

As mentioned in Section 3.3, clients, Web servers, their mutual communication and third parties can fail. In such a case, no verdict can be given on the correctness of the implementation of the Internet application. However, depending on the failure, it might be possible to determine the failing entity.

10.3 Introduction to Conformance Testing

As a basis for conformance testing of Internet applications, we take the formal framework as introduced in [BAL+90, Tre94, ISO96]. Given a specification, the goal is to check, by means of testing, whether an implemented system satisfies its specification. To be able to formally test applications, there is a need for implementations and formal specifications. Then, *conformance* can be expressed as a relation on these two sets.

Implementations under test are real objects which are treated as black boxes exhibiting behaviour and interacting with their environment. They are not amenable to formal reasoning, which makes it harder to formally specify the conformance relation. Therefore, we make the assumption that any implementation can be modelled by a formal object. This assumption is referred to as the *test hypothesis* [Ber91] and allows us to handle implementations as formal objects. We can express conformance by a formal relation between a model of an implementation and a specification, a so-called *implementation relation*.

An implementation is tested by performing experiments on it and observing its reactions to these experiments. The specification of such an experiment is called a *test case*, a set of test cases a *test suite*. Applying a test to an implementation is called *test execution* and results in a verdict. If the implementation passes or fails the test case, the verdict is *pass* or *fail*, respectively. If no verdict can be given, the verdict is *inconclusive*.

In the remainder of this chapter, we instantiate the ingredients of the framework as sketched above. We give a formalism for both modelling implementations of Internet applications and for giving formal specifications used for test generation. Furthermore, we give an implementation relation. By doing this, we are able to test whether a (model of an) implementation conforms to its specification. Apart from that, we give an algorithm for generating test suites from specifications of Internet applications.

10.4 Formal Model

To be able to formally test Internet applications, we need to formally model their behaviour. Since we focus on conformance testing, we are mainly interested in the communication between the application and its users. We do not focus on the representation of data. Furthermore, we focus on black-box testing, which means that the internal state of an application is not known in the model. Finally, we focus on thin client Internet applications that communicate using the Hypertext Transfer Protocol (HTTP) [FGM+99]. As a result, the applications show a request/response behaviour.

These observations lead to modelling Internet applications using labelled transition systems. Each transition in the model represents a communication action between the application and a client. The precise model is dictated by the interacting behaviour of the HTTP protocol, as explained in Section 3.2.

In general, an HTTP interaction is initiated by a client, sending a request for some information to an application. A request can be extended with parameters. These parameters can be used by the application. After calculating a response, it is sent back to the requesting client. Normally, successive requests are not grouped. However, the grouping can be done by adding parameters to the requests and responses. In such a way, alternating sequences of requests and responses are turned into sessions. All this can be found in Section 4.3.2 where client interaction is introduced.

Note that we test the interaction behaviour of Internet applications communicating via HTTP. We do *not* model the client-side tools to interact with Internet applications, i.e., we do not model the behaviour of the application when using browser buttons like *stop*, *back*, *forward* and *refresh*. The main reason for not including this behaviour is that different client implementations cause distinct interaction behaviour. For example, some browsers may store visited pages in a cache such that pressing a back button does not result in the sending of a request but in simply showing the stored page. Pressing the back button in other browsers may result in sending a new request for the previously visited page.

Furthermore, we do not add (failure of) components in the system under test, other than the application, to the specification. This means that failure of any of these components leads to tests in which the result is *inconclusive*. If all components in the system under test operate without failure, verdicts are pass or fail. This implies that testing of rollbacks is *not* taken into account.

The tester should behave like a set of thin clients. The only requests sent to the application are the initial request which models the typing in of a URL in the browser's address bar and requests that result from clicking on links or submitting forms which are contained in preceding responses.

Since we focus on HTTP based Internet applications, and thus on sessions of alternating request/response communication with applications, we make use of so-called *multi request/response transition systems* (MRRTSs) for modelling implementations of Internet applications and initially for giving formal specifications used for test generation. Later, in Section 10.8, we show how *DiCons* specifications can serve as a basis for the tester to test Internet applications. An MRRTS is a labelled transition system having extra structure. In the remainder of this section we explain MRRTSs in more detail and show how they relate to labelled transition systems and *request/response transition systems* (RRTSs).

10.4.1 Labelled Transition Systems

The formalism of *labelled transition systems* is widely used for describing the behaviour of processes. We provide the relevant definitions.

Definition 10.4.1 *A* labelled transition system *is a 4-tuple* $\langle S, L, \rightarrow, s_0 \rangle$ *where*

- *S is a countable, non-empty set of* states*;*
- *L is a countable set of* labels*;*
- $\rightarrow \subseteq S \times L \times S$ *is the* transition relation*;*
- $s_0 \in S$ *is the* initial state*.*

Note that we do *not* draw a distinction between states which can successfully terminate and deadlock states, as we do in *DiCons*. The testing theory is based on black-box testing using test suites that consist of traces of interactions. These traces are constructed using the specification. By black-box testing of Internet applications, we assume that we cannot determine whether an implementation under test reaches a terminating or deadlock state after acceptance of a trace. If all traces in the specification are accepted by the application, it conforms to the specification.

Definition 10.4.2 *Let s_i ($i \in \mathbb{N}$) be states and a_i ($i \in \mathbb{N}$) be labels. A (finite) composition of transitions*

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots s_n \xrightarrow{a_n} s_{n+1}$$

is then called a computation*. The sequence of actions of a computation, $a_1;a_2;\ldots;a_n$, is called a* trace*. The empty trace is denoted by ε. If L is a set of labels, the set of all finite traces over L is denoted by L^*.*

Definition 10.4.3 *Let $p = \langle S, L, \rightarrow, s_0 \rangle$, $s, s' \in S$, $S' \subseteq S$, $a_i \in L$ and $\varsigma \in L^*$. Then,*

$$
\begin{array}{rcll}
s \xrightarrow{a_1;\ldots;a_n} s' & =_{\text{def}} & \exists s_1,\ldots,s_{n-1}\; s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots s_{n-1} \xrightarrow{a_n} s' & \\
s \xrightarrow{a_1;\ldots;a_n} & =_{\text{def}} & \exists s'\; s \xrightarrow{a_1;\ldots;a_n} s' & \\
\text{init}(s) & =_{\text{def}} & \{a \in L \mid s \xrightarrow{a}\} & \textit{all possible transitions from } s \\
\text{traces}(s) & =_{\text{def}} & \{\varsigma \in L^* \mid s \xrightarrow{\varsigma}\} & \textit{all possible execution sequences from } s \\
\text{traces}(S') & =_{\text{def}} & \bigcup s' \in S' \; \text{traces}(s') & \textit{all possible execution sequences from } S' \\
s \;\textbf{after}\; \varsigma & =_{\text{def}} & \{s' \in S \mid s \xrightarrow{\varsigma} s'\} & \textit{reachable states from } s \textit{ after execution of } \varsigma \\
S' \;\textbf{after}\; \varsigma & =_{\text{def}} & \bigcup s' \in S' \;\; s' \;\textbf{after}\; \varsigma & \textit{reachable states from } S' \textit{ after execution of } \varsigma
\end{array}
$$

A labelled transition system $p = \langle S, L, \rightarrow, s_0 \rangle$ is identified by its initial state s_0. So, e.g., we can write traces(p) instead of traces(s_0) and p **after** ς instead of s_0 **after** ς.

We aim at modelling the behaviour of the HTTP protocol using labelled transition systems. Therefore, we need to add restrictions on the traces in the labelled transition system used for modelling this behaviour. One of these restrictions is that traces in the LTSs should answer the alternating request/response behaviour.

Definition 10.4.4 *Let A, B be sets of labels. Then* alt(A, B) *is the (infinite) set of traces having alternating structure with respect to elements in A and B, starting with an element in A. Formally,* alt(A, B) *is the smallest set such that*

$$\varepsilon \in \text{alt}(A, B) \quad \wedge \quad \forall \varsigma \in \text{alt}(B, A) \forall a \in A \; a\varsigma \in \text{alt}(A, B) \; .$$

As mentioned before, interactions with an Internet application can be grouped into sessions. To be able to specify the behaviour within each session, we make use of a projection function. This function is used for determining all interactions contained within one session.

Definition 10.4.5 *Let ς be a trace and A be a set of labels. Then $\varsigma|_A$, the* projection *of ς to A, is defined by*

$$
\begin{array}{rcl}
\varepsilon|_A & =_{\text{def}} & \varepsilon \\
(a;\varsigma)|_A & =_{\text{def}} & \begin{cases} a;(\varsigma|_A) & \text{if } a \in A \\ \varsigma|_A & \text{if } a \notin A \; . \end{cases}
\end{array}
$$

Definition 10.4.6 *A* partitioning *S of a set A is a collection of mutually disjoint non-empty subsets of A such that their union exactly equals A:*

$$\forall B \in S \; B \neq \emptyset \quad \wedge \quad \bigcup S = A \quad \wedge \quad \forall B, C \in S \; B \neq C \Rightarrow B \cap C = \emptyset$$

10.4.2 Request/Response Transition Systems

We give a formal definition of a request/response transition system, denoted by RRTS. RRTSs can be compared with input/output transitions systems (IOTSs) as defined in [Tre95]. As in IOTSs, we differentiate between two sets of labels, called *request labels* and *response labels*, respectively. Where in IOTSs inputs are always enables, RRTSs are based on pure request/response alternation.

Definition 10.4.7 *Let L be a countable set of labels and $\{L_?, L_!\}$ be a partitioning of L. Then, a* request/response transition system $\langle S, L_?, L_!, \rightarrow, s_0\rangle$ *is a labelled transition system $\langle S, L, \rightarrow, s_0\rangle$ such that*

$$\forall \varsigma \in \text{traces}(s_0) \quad \varsigma \in \text{alt}(L_?, L_!) \ .$$

Elements in $L_?$ are called request labels, *elements in $L_!$* response labels.

RRTSs resemble the notion of Mealy machines [Mea55], however, it turns out to be technically simpler to start from the notion of RRTSs since our focus is on modelling HTTP interaction instead of on the translation of input strings (requests) into output strings (responses). Furthermore, the alternating behaviour of requests and responses is contained in the structure of RRTSs where it should be contained in both the next-state and output functions of Mealy machines.

10.4.3 Multi Request/Response Transition Systems

IOTSs can be used as a basis for multi input/output transition systems (MIOTSs) [Hee98]. Similarly, in a multi request/response transition system (MRRTS), multiple request/response transition systems are combined into one. All subsystems behave like an RRTS, however interleaving between the subsystems is possible.

Definition 10.4.8 *Let L be a countable set of labels. Let $\mathbb{L} \subseteq \mathcal{P}(L) \times \mathcal{P}(L)$ be a countable set of pairs such that $\{A, B \in \mathcal{P}(L) \mid (A, B) \in \mathbb{L}\}$ is a partitioning of L. Then, a* multi request/response transition system $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$ *is a labelled transition system $\langle S, L, \rightarrow, s_0\rangle$ such that*

$$\forall (A, B) \in \mathbb{L}\ \forall \varsigma \in \text{traces}(s_0) \quad \varsigma\,|_{A \cup B} \in \text{alt}(A, B) \ .$$

The set of all possible request labels, $\mathbb{L}_?$, is defined by

$$\mathbb{L}_? \quad =_{\text{def}} \quad \bigcup_{(A,B) \in \mathbb{L}} A \ .$$

The set of all possible response labels, $\mathbb{L}_!$, is defined by

$$\mathbb{L}_! \quad =_{\text{def}} \quad \bigcup_{(A,B)\in\mathbb{L}} B \ .$$

Note that an RRTS $\langle S, L_?, L_!, \rightarrow, s_0\rangle$ can be interpreted as MRRTS $\langle S, \{(L_?, L_!)\}, \rightarrow, s_0\rangle$, i.e., each MRRTS having singleton $\mathbb{L}$ is an RRTS.

We introduce some extra functions on the sets of pairs as introduced in Definition 10.4.8.

Definition 10.4.9 *Let $\mathbb{L} \subseteq \mathcal{P}(L) \times \mathcal{P}(L)$ be a countable set of pairs such that $\{A, B \in \mathcal{P}(L) \mid (A, B) \in \mathbb{L}\}$ is a partitioning of L, where each tuple contains a set of request labels and a set of response labels. We define functions* req *and* resp *for determining corresponding requests or responses given either a request label or response label. For $x \in L$, we define functions* $\text{req}, \text{resp} : L \rightarrow \mathcal{P}(L)$*, such that*

$$(\text{req}(x), \text{resp}(x)) \in \mathbb{L} \quad \textit{and} \quad x \in \text{req}(x) \cup \text{resp}(x) \ .$$

10.5 Relating Multi Request/Response Transition Systems

An implementation conforms to a specification if an implementation relation exists between the model of the implementation and its specification. We model both the implementation and the specification as multi request/response transition systems, so conformance can be defined by a relation on MRRTSs.

While testing Internet applications, we examine the responses sent by the application and check whether they are expected responses by looking at the specification. So we focus on testing whether the implementation does what it is expected to do, not what it is not allowed to do.

Given a specification, we make use of function exp to determine the set of expected responses in a state in the specification.

Definition 10.5.1 *Let p be a multi request/response transition system $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$. For each state $s \in S$ and for each set of states $S' \subseteq S$, the set of expected responses in s and S' is defined as*

$$\begin{aligned} \exp(s) \quad &=_{\text{def}} \quad \text{init}(s) \cap \mathbb{L}_! \\ \exp(S') \quad &=_{\text{def}} \quad \bigcup s' \in S' \ \ \exp(s') \ . \end{aligned}$$

If a model of an implementation i conforms to a specification s, the possible responses in all reachable states in i should be contained in the set of possible responses in the corresponding states in s. Corresponding states are determined by executing corresponding traces in both i and s.

Definition 10.5.2 *Let* MRRTS *i be the model of an implementation and* MRRTS *s be a specification. Then i conforms to s with respect to request/response behaviour, i* **rrconf** *s, if and only if all responses of i are expected responses in s:*

$$i \textbf{ rrconf } s \quad =_{\text{def}} \quad \forall \varsigma \in \text{traces}(s) \;\; \exp(i \textbf{ after } \varsigma) \subseteq \exp(s \textbf{ after } \varsigma) \; .$$

Relation **rrconf** on MRRTSs is analogous to relation **conf** on LTSs as formalised in [BSS87].

10.6 Test Derivation

An implementation is tested by performing experiments on it and observing its reactions to these experiments. The specification of such an experiment is called a *test case*. Applying a test to an implementation is called *test execution*. By now we have all elements for deriving such test cases.

Since the specification is modelled by an MRRTS, a test case consists of request and response actions as well. However, we have some more restrictions on test cases. First of all, test cases should have finite behaviour to guarantee that tests terminate. Apart from that, unnecessary nondeterminism should be avoided, i.e., within one test case the choice between multiple requests or between requests and responses should be left out.

In this way, a test case is a labelled transitions system where each state is either a terminating state, a state in which a request is sent to the implementation under test, or a state in which a response is received from the implementation. The terminating states are labelled with a verdict which is a **pass** or **fail**.

Definition 10.6.1 *A* test case *t is an LTS $\langle S, \mathbb{L}_? \cup \mathbb{L}_!, \rightarrow, s_0 \rangle$ such that*

- *t is deterministic and has finite behaviour;*
- *S contains terminal states* **pass** *and* **fail** *with* $\text{init}(\textbf{pass}) = \text{init}(\textbf{fail}) = \emptyset$*;*
- *for all $s \in S \setminus \{\textbf{pass}, \textbf{fail}\}$,* $\text{init}(s) = \{a\}$ *for $a \in \mathbb{L}_?$ or* $\text{init}(s) = \mathbb{L}_!$.

We denote this subset of LTSs by TESTS. *A set of test cases $T \subseteq$* TESTS *is called a* test suite.

We do not include the possibility for reaching *inconclusive* states in test cases. Such verdicts are given if a component in the system under test, other than the application, fails. The tester (as described in Section 10.2) is able to identify errors caused by the application and lead to a **fail** state. Other errors result in an inconclusive verdict.

As mentioned, we call a set of test cases a *test suite*. Such a test suite is used for determining whether an implementation conforms to a specification. A test suite T is said to be *sound* if and only if all implementations that conform to the specification pass all test cases in T. If all implementations that do not conform to the specification fail a test case in T, T is called *exhaustive*. A test suite that is both sound and exhaustive is said to be *complete* [ISO96].

Definition 10.6.2 *Let* MRRTS *i be an implementation and T be a test suite. Then, implementation i passes test suite T if no traces in i lead to a fail state:*

$$i \ \textbf{passes}\ T \quad =_{\text{def}} \quad \neg\exists t \in T\ \exists \varsigma \in \text{traces}(i)\ \ \varsigma;\textbf{fail} \in \text{traces}(t)$$

We use the notation $\varsigma;\textbf{fail}$ to represent trace ς leading to a fail state, i.e.,

$$\varsigma;\textbf{fail} \in \text{traces}(t) \quad =_{\text{def}} \quad t \xrightarrow{\varsigma} \textbf{fail}\ .$$

Definition 10.6.3 *Let s be a specification and T be a test suite. Then for relation* **rrconf***:*

$$\begin{array}{llll} T \text{ is sound} & =_{\text{def}} & \forall i\ \ i\ \textbf{rrconf}\ s & \Longrightarrow\ \ i\ \textbf{passes}\ T \\ T \text{ is exhaustive} & =_{\text{def}} & \forall i\ \ i\ \textbf{rrconf}\ s & \Longleftarrow\ \ i\ \textbf{passes}\ T \\ T \text{ is complete} & =_{\text{def}} & \forall i\ \ i\ \textbf{rrconf}\ s & \Longleftrightarrow\ \ i\ \textbf{passes}\ T \end{array}$$

In practice, however, finite test suites are often incomplete and infinite test suites cannot be processed in finite time. So, we have to restrict ourselves to test suites for detecting non-conformance instead of test suites for giving a verdict on the conformance of the implementation. Such test suites are called *sound*.

To test conformance with respect to request/response behaviour, we have to check for all possible traces in the specification that the responses generated by the implementation are expected responses in the specification. This can be done by having the implementation execute traces from the specification. The responses of the implementation are observed and compared with the responses expected in the specification. Expected responses pass the test, unexpected responses fail the test. The algorithm given is based on the algorithm for generating test suites as defined in [Tre96].

Algorithm 10.6.4 *Let s be* MRRTS $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$. *For any non-empty set C of states of the specification, we define the collection of nondeterministic recursive algorithms* gentest^n

($n \in \mathbb{N}$) *for deriving test cases as follows:*

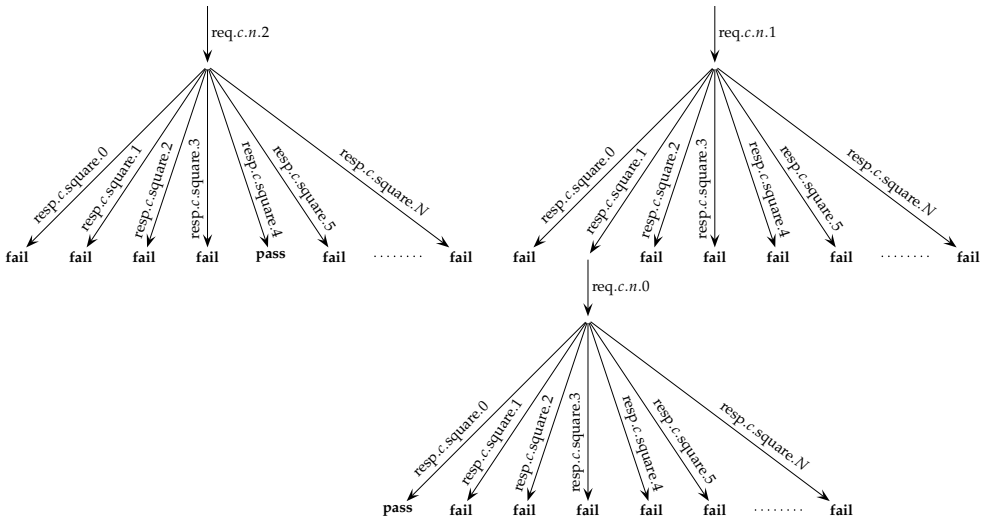
$$
\begin{array}{rcl}
\text{gentest}^n & : & \mathcal{P}(S) \to \text{TESTS} \\[1ex]
\text{gentest}^n(C) & =_{\text{def}} & [\ \ \text{return } \textbf{pass} \\
& & [\!]\ \ n > 0 \wedge a \in \mathbb{L}_? \ \wedge\ C \textbf{ after } a \neq \emptyset \quad \to \\
& & \qquad \text{return } a; \text{gentest}^{n-1}(C \textbf{ after } a) \\
& & [\!]\ \ n > 0 \quad \to \\
& & \qquad \text{return } \sum\{b; \textbf{fail} \mid b \in \mathbb{L}_! \setminus \exp(C)\} \\
& & \qquad\quad + \sum\{b; \text{gentest}^{n-1}(C \textbf{ after } b) \mid b \in \exp(C)\} \\
& &]
\end{array}
$$

Algorithm gentest^n can be used for the generation of test cases having a maximum of n transitions. In each recursive step, the algorithm makes a nondeterministic choice out of three options. The first option is to end the test case by returning a **pass**. Alternatively, if a request is enabled, this request can be added to the test case after which the algorithm is recursively called. Finally, a response can be inspected which either results in a **fail** state if it is unexpected or in a recursive call if it is an expected response. The algorithm returns an process-algebraic expression (that can be considered a test case). Algorithm gentest$^n(\{s_0\})$ generates all sound test cases with at most n transitions, starting in state s_0.

As mentioned in Definition 10.4.2, the ; infix notation is used for trace composition. So, e.g., trace $a;b$ relates to transitions $s \xrightarrow{a} s' \xrightarrow{b} s''$. As mentioned, notation $a;\textbf{pass}$ and $a;\textbf{fail}$ is used for representing transitions $s \xrightarrow{a} \textbf{pass}$ and $s \xrightarrow{a} \textbf{fail}$, respectively. We use Σ-notation, which is the generalisation of the alternative composition, to indicate that it is not known which of the responses is returned by the implementation. So, e.g. alternative composition $a + b$ (see Section 5.2) relates to transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{b} s''$. Depending on whether the response is expected, the algorithm might either continue or terminate in a **fail** state.

Although a choice for the first option can be made in each step, we added a parameter to the algorithm, $n \in \mathbb{N}$, to force termination. As mentioned, we want all test cases to be finite, since otherwise no verdict might take place.

To give an example, suppose we have an MRRTS where repetitively one type of request can be sent, which contains an integer value n. The corresponding response returns the square of the value: n^2. Then, e.g., these two test cases can be generated using the algorithm:



The set of derivable test cases from $\text{gentest}^n(C)$ is denoted by $\overline{\text{gentest}^n(C)}$. So set $\overline{\text{gentest}^n(C)}$ contains all possible test cases of at most n transitions starting in states C of the specification. Although our goal is to generate sound test suites, we prove that in the limit, as n goes to infinity, test suite $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$ is complete for specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. To prove this, we make use of some lemmas.

Lemma 10.6.5 *Let s be a specification* $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ *and* $\varsigma_0, \varsigma_1 \in L^*$, $\varsigma_1 \neq \varepsilon$. *Then*

$$\varsigma_0\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \iff \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0|}(s_0 \textbf{ after } \varsigma_0)})$$

where $|\varsigma|$ *is the length of trace* ς.

Proof This lemma can be proved by using induction on the structure of ς_0. The full proof can be found in Appendix C.1 on page 235. □

Lemma 10.6.6 *Let s be a specification* $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$, $\varsigma_0 \in L^*$ *and* $n > 0$. *Then*

$$\varsigma; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \implies \exists \varsigma' \in L^* \exists b \in \mathbb{L}_! \; \varsigma = \varsigma' b \;\; .$$

Proof This can be easily seen by looking at the definition of the gentest algorithm: State **fail** can only be reached after execution of a $b \in \mathbb{L}_!$. □

Theorem 10.6.7 *Let s be a specification* $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$*. Then test suite*

$$\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})} \text{ is complete.}$$

Proof We prove this by proving exhaustiveness ($\Leftarrow$) and soundness ($\Rightarrow$) separately using proofs by contradiction. The full proof can be found in Appendix C.2 on page 237. □

10.6.1 Adapting the Algorithm

If we know or assume that the implementation satisfies some soundness condition, we need not test for violations of that condition. So, then we need fewer tests. Here we assume that the implementation has the alternating request/response behaviour: the implementation can only send responses on requests sent by the tester.

As can be seen by looking at Algorithm 10.6.4, each choice for inspecting a response of the implementation leads to $|\mathbb{L}_!|$ new branches in the generated test case. However, as a result of the alternating request/response behaviour of the implementation, many of these branches never take place.

This means that we only want to contain this restricted set of responses in the test case. The generated test case itself then also is an MRRTS. As a result, while generating test cases, we keep track of the responses that might be sent by the implementation according to the definition of MRRTSs. We do this by watching the set of responses that can be received at each state in the test case and only add these responses to the test case. In this way, the alternating behaviour of the request/response interaction is maintained.

Algorithm 10.6.8 *Let s be* MRRTS $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$*. For any non-empty set C of states of the specification, we define the collection of nondeterministic recursive algorithms* gentest^n_E *($n \in \mathbb{N}$, $E \subseteq \mathbb{L}_!$) for deriving test cases as follows:*

$$\text{gentest}^n_E \quad : \quad \mathcal{P}(S) \rightarrow \text{TESTS}$$

$$\begin{array}{lll}
\text{gentest}^n_E(C) & =_{\text{def}} & [\ \ \text{return } \textbf{pass} \\
& & [] \ \ n > 0 \wedge a \in \mathbb{L}_? \ \wedge \ C \textbf{ after } a \neq \emptyset \quad \rightarrow \\
& & \quad\quad \text{return } a; \text{gentest}^{n-1}_{E \cup \text{resp}(a)}(C \textbf{ after } a) \\
& & [] \ \ n > 0 \wedge E \neq \emptyset \quad \rightarrow \\
& & \quad\quad \text{return } \sum\{b; \textbf{fail} \mid b \in E \setminus \text{exp}(C)\} \\
& & \quad\quad\quad + \sum\{b; \text{gentest}^{n-1}_{E \setminus \text{resp}(b)}(C \textbf{ after } b) \mid b \in \text{exp}(C)\} \\
& &]
\end{array}$$

Algorithm 10.6.8 extends Algorithm 10.6.4 with an extra parameter, $E \subseteq \mathbb{L}_!$. This set E is used for keeping track of the sets of *enabled responses*, i.e., responses that might be received by the tester at the current state of the test case. Initially, this set is empty, viz. a response cannot be received if no request has been sent. Sending a request adds the corresponding set of responses to E. If set E is not empty, the choice for examining a response is enabled. Receiving a response leads to dropping the set of corresponding responses, i.e. the set of responses in E that contains the received response. Abstracting from responses that cannot occur because they are elements of other sessions is allowed: they do not occur in the implementation since all responses have to be preceded by a corresponding request.

We prove that test suite $\bigcup_{n>0} \overline{\text{gentest}^n_\emptyset(\{s_0\})}$ is complete as well. To prove this, again, we make use of some lemmas.

Lemma 10.6.9 *Let s be an* MRRTS $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$*, L be the set of labels in s, ς be a trace in s and E_ς be the set of expected responses after execution of ς in s:*

$$E_\varsigma \quad = \quad \{b \in \mathbb{L}_! \mid \exists a \in \text{req}(b)\ \exists \varsigma_0, \varsigma_1 \in L^*\ (\varsigma = \varsigma_0 a \varsigma_1 \wedge \neg \exists b' \in \text{resp}(a)\ b' \in \varsigma_1)\}\ .$$

Then,

1. $\forall a \in \mathbb{L}_?\ E_\varsigma \cup \text{resp}(a) = E_{\varsigma a}$
2. $\forall b \in \mathbb{L}_!\ E_\varsigma \setminus \text{resp}(b) = E_{\varsigma b}$

Proof The proof can be found in Appendix C.3 on page 239. □

Lemma 10.6.10 *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$. Then*

$$\forall n > 0 \quad \text{traces}(\overline{\text{gentest}^n_\emptyset(\{s_0\})}) \subseteq \text{traces}(\overline{\text{gentest}^n(\{s_0\})})\ .$$

Proof By comparing algorithms 10.6.4 and 10.6.8, we see that the only difference is in the choice for option three and the number of fail traces generated via option three. All pass traces that are generated by Algorithm 10.6.8 are generated by Algorithm 10.6.4 since those do not depend on E. The fail traces generated by Algorithm 10.6.8 are also generated by Algorithm 10.6.4 since $E \subseteq \mathbb{L}_!$. □

Lemma 10.6.11 *Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$ and $\varsigma_0, \varsigma_1 \in L^*$, $\varsigma_1 \neq \varepsilon$. Then*

$$\varsigma_0\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n_\emptyset(\{s_0\})}) \quad \equiv \quad \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0|}_{E_{\varsigma_0}}(s_0\ \textbf{after}\ \varsigma_0)})$$

where $|\varsigma|$ is the length of trace ς and E_ς is as defined in Lemma 10.6.9.

Proof This Lemma follows from the definitions of Algorithms 10.6.4 and 10.6.8 and Lemmas 10.6.5, 10.6.9 and 10.6.10. □

Lemma 10.6.12 *Let* MRRTS *i be* $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. *Then,*

$$\forall b \in \mathbb{L}_! \quad \varsigma b \in \text{traces}(i) \;\Rightarrow\; b \in E_\varsigma$$

where E_ς *is as defined in Lemma 10.6.9.*

Proof The proof can be found in Appendix C.4 on page 241. □

Theorem 10.6.13 *Let s be a specification* $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. *Then test suite*

$$\bigcup_{n>0} \overline{\text{gentest}^n_\emptyset(\{s_0\})} \text{ is complete.}$$

Proof The proof can be found in Appendix C.5 on page 241. □

Up till now, we only took batch-wise testing into account, i.e., a test suite is generated after which test execution takes place. As mentioned before, test cases, and thus test suites, often grow very large. Therefore it is more suitable to execute the test cases on-the-fly. That is, while generating test cases, the actions are immediately executed by interacting with the implementation under test. The algorithms presented in this chapter are also suitable for on-the-fly testing: When a choice is made for sending a request (second option), the request is directly sent to the implementation. If the next step in the test case is receiving a response (third option), a response is inspected and, depending on whether the response is expected, the test case fails or continues.

10.7 Example

We show how the theory introduced in former sections can be used for testing real-life Internet applications. As an example, we take a voting protocol. All members of a group of voters are asked whether they are for or against a proposition. They are able to visit a Web site where they can either vote or have a look at the current score. They can vote at most once and they can check the score as often as they want to.

We start by giving an MRRTS that formally specifies the application. Let V be the set of voters and $\mathbb{P} = \{\text{for}, \text{against}\}$.

All voters $v \in V$ can start a vote. If a voter has indicated that he wants to vote, he is asked to vote for or against a proposition. The voter votes and his vote is confirmed or rejected, depending on whether the voter had voted before. As a result, a correct voting session can be modelled by the following sequence of requests and responses:

$$\text{start}.v_{?k} \overset{0}{;} \text{vote}.v_{!k} \overset{1}{;} \text{vote}.v.p_{?k} \overset{2}{;} \text{ok}.v_{!k}$$

If the vote is rejected, the final response is a $\neg\text{ok}.v_{!k}$. More details and the exact syntax and semantics of the actions are given below, in Example 10.7.1. We number the semicolons in order to be able to keep track of the state the session is in. This is also explained below.

A score-checking session consists of a single request/response interaction (a reactive push). A trace representing this interaction has the following structure:

$$\text{score}.v_{?k}\,;\text{score}.v.f.a_{!k}$$

In Example 10.7.1 we give an MRRTS specifying the behaviour described above.

Example 10.7.1 *Let $\langle S, \mathbb{L}, \rightarrow, s_0\rangle$ be the* MRRTS *having the following structure:*

- $\mathbb{L}$, *the set of pairs of transition labels is defined as follows:*

$$\begin{array}{rcl} \mathbb{L} & = & \{ \; (\; \{\text{start}.v_{?k}, \text{vote}.v.\text{for}_{?k}, \text{vote}.v.\text{against}_{?k}\}, \\ & & \quad \{\text{vote}.v_{!k}, \text{ok}.v_{!k}, \neg\text{ok}.v_{!k}\} \\ & & \quad) \mid v \in V, k \in \mathbb{K} \; \} \\ & \cup & \{ \; (\; \{\text{score}.v_{?k}\}, \\ & & \quad \{\text{score}.v.f.a_{!k}\} \\ & & \quad) \mid f, a \in \mathbb{N}, v \in V, k \in \mathbb{K} \; \} \end{array}$$

  *The first part specifies the interactions where voter v starts a vote and sends a request to vote for or against the proposition. The response on a start is the question to vote for or against the proposition. The response on a vote is a confirmation (*ok*) or a denial (*$\neg$ok*), depending on whether the voter had voted before. The second part specifies the requests for the score which are responded by the number of votes for (f) and against (a) the proposition. All labels are extended with an identifier k for uniquely identifying the sessions. Request labels are marked with a question-mark and response labels with an exclamation-mark.*

- S, *the set of states, is defined as follows:*

$$\begin{array}{rcll} S & = & \mathcal{P}(\mathbb{K} \times V \times \{0,1,2\} \times \mathbb{P}) & \textit{states of the vote sessions} \\ & \times & \mathcal{P}(\mathbb{K} \times V \times \mathbb{N} \times \mathbb{N}) & \textit{states of the score-checking sessions} \\ & \times & \mathcal{P}(V) & \textit{voters who voted} \\ & \times & \mathbb{N} & \textit{number of voters who voted for} \\ & \times & \mathbb{N} & \textit{number of voters who voted against} \end{array}$$

For $\langle Q, R, W, f, a\rangle \in S$*,*

- $Q \subseteq \mathbb{K} \times V \times \{0,1,2\} \times \mathbb{P}$ *are the states of the vote sessions.*
  For $\langle k, v, i, p\rangle \in Q$*,*
  - $k \in \mathbb{K}$ *is the session label;*
  - $v \in V$ *is the voter who is voting;*
  - $i \in \{0,1,2\}$ *is the state the session is in. This i indicates at which semicolon the vote session is, where we use the labelling as given above;*
  - $p \in \mathbb{P}$ *is the actual vote: for or against the proposition;*
- $R \subseteq \mathbb{K} \times V \times \mathbb{N} \times \mathbb{N}$ *are the states of the score-checking sessions.*
  For $\langle k, v, f_k, a_k\rangle \in R$*,*
  - $k \in \mathbb{K}$ *is the session label;*
  - $v \in V$ *is the voter who is checking the score;*
  - $f_k \in \mathbb{N}$ *is the number of voters who voted for the proposition at the time that session k started;*
  - $a_k \in \mathbb{N}$ *is the number of voters who voted against the proposition at the time that session k started;*
- $W \subseteq V$ *is the set of voters who voted;*
- $f \in \mathbb{N}$ *is the number of voters who voted for the proposition;*
- $a \in \mathbb{N}$ *is the number of voters who voted against the proposition.*

- *Transition relation* $\rightarrow$ *is defined by the following derivation rules.*

  A vote session, uniquely labelled by k, can be started by voter v by sending a start *request:*

$$\frac{k \in \mathbb{K},\ v \in V,\ \neg\exists k' \in \mathbb{K}\setminus\{k\}\ \exists v' \in V\ \exists i \in \{0,1,2\}\ \exists p \in \mathbb{P}\ \langle k', v', i, p\rangle \in Q}{\langle Q, R, W, f, a\rangle \xrightarrow{\text{start}.v_{?k}} \langle Q \cup \{\langle k, v, 0, \bot\rangle\}, R, W, f, a\rangle}$$

  If a start *request has been received from voter v in session k, a response, asking for a vote, can be sent:*

$$\frac{\langle k, v, 0, \bot\rangle \in Q}{\langle Q, R, W, f, a\rangle \xrightarrow{\text{vote}.v_{!k}} \langle (Q \setminus \{\langle k, v, 0, \bot\rangle\}) \cup \{\langle k, v, 1, \bot\rangle\}, R, W, f, a\rangle}$$

  After the voter received the response to vote, he can send a request containing the vote:

$$\frac{\langle k, v, 1, \bot\rangle \in Q,\ p \in \mathbb{P}}{\langle Q, R, W, f, a\rangle \xrightarrow{\text{vote}.v.p_{?k}} \langle (Q \setminus \{\langle k, v, 1, \bot\rangle\}) \cup \{\langle k, v, 2, p\rangle\}, R, W, f, a\rangle}$$

If the vote is accepted, i.e., if the voter had not voted before ($v \notin W$), the vote is counted and confirmed and the session ends:

$$\frac{\langle k, v, 2, \text{for}\rangle \in Q,\ v \notin W}{\langle Q, R, W, f, a\rangle \xrightarrow{\text{ok}.v_{!k}} \langle Q \setminus \{\langle k, v, 2, \text{for}\rangle\}, R, W \cup \{v\}, f+1, a\rangle}$$

$$\frac{\langle k, v, 2, \text{against}\rangle \in Q,\ v \notin W}{\langle Q, R, W, f, a\rangle \xrightarrow{\text{ok}.v_{!k}} \langle Q \setminus \{\langle k, v, 2, \text{against}\rangle\}, R, W \cup \{v\}, f, a+1\rangle}$$

If the voter had voted before, or is concurrently sending a vote in another session, the vote can be rejected and the session ends:

$$\frac{\langle k, v, 2, p\rangle \in Q, \quad v \in W \ \vee\ \exists k' \in \mathbb{K}\setminus\{k\}\ \exists p' \in \mathbb{P}\ \langle k', v, 2, p'\rangle \in Q}{\langle Q, R, W, f, a\rangle \xrightarrow{\neg\text{ok}.v_{!k}} \langle Q \setminus \{\langle k, v, 2, p\rangle\}, R, W, f, a\rangle}$$

A score-checking session, uniquely labelled by k, can be started by voter v by sending a score *request:*

$$\frac{k \in \mathbb{K},\ v \in V,\ \neg\exists k' \in \mathbb{K}\setminus\{k\}\ \exists v' \in V\ \exists f', a' \in \mathbb{N}\ \langle k', v', f', a'\rangle \in R}{\langle Q, R, W, f, a\rangle \xrightarrow{\text{score}.v_{?k}} \langle Q, R \cup \{\langle k, v, f, a\rangle\}, W, f, a\rangle}$$

If a request for the score has been sent, the scores can be returned to the requesting client. Since interactions can overtake each other, the result can be any of the scores between the sending of the request and the receiving of the response. So, the score must be at least the score at the moment of requesting the score and at most the number of processed votes plus the number of correct votes, sent in between requesting for the score and receiving the score:

$$\frac{\begin{array}{c}\langle k, v, f_k, a_k\rangle \in R,\\ f_k \leq f' \leq f + (\#v{\in}V\exists k{\in}\mathbb{K}\ \langle k, v, 2, \text{for}\rangle \in Q \wedge v \notin W),\\ a_k \leq a' \leq a + (\#v{\in}V\exists k{\in}\mathbb{K}\ \langle k, v, 2, \text{against}\rangle \in Q \wedge v \notin W)\end{array}}{\langle Q, R, W, f, a\rangle \xrightarrow{\text{score}.v.f'.a'_{?k}} \langle Q, R \setminus \{\langle k, v, f_k, a_k\rangle\}, W, f, a\rangle}$$

- *Initial state* $s_0 = \langle \emptyset, \emptyset, \emptyset, 0, 0\rangle$*: no requests to start a vote or inspect the scores have been sent yet, no one has voted for and no one has voted against the proposition.*

This MRRTS *specifies the example where voters can vote once and check for the scores as often as they want.*

As a proof of concept, we implemented an on-the-fly version of Algorithm 10.6.8. We used this algorithm to test eleven implementations of the Internet vote application: one correct and ten incorrect implementations. We tested by executing 26.000

implementation	% failures	verdict
1. correct implementation	0.00	**pass**
2. no synchronisation: first calculate results, then remove voter	33.30	**fail**
3. no synchronisation: first remove voter, then calculate results	32.12	**fail**
4. votes are incorrectly initialised	91.09	**fail**
5. votes for and against are mixed up	87.45	**fail**
6. votes by voter 0 are not counted	32.94	**fail**
7. voter 0 cannot vote	91.81	**fail**
8. unknown voter can vote	0.00	**pass**
9. voters can vote more than once	68.75	**fail**
10. voter 0 is allowed to vote twice	16.07	**fail**
11. last vote is counted twice	8.82	**fail**

Table 10.1: Test results.

test cases per implementation. This took approximately half a day per implementation. We tested using different lengths of test traces, varying between 10 and 110 transitions. We also varied the numbers of voters, choosing numbers between 5 and 50. The test results are briefly summarised in Table 10.1. The left column describes the error in the implementation. In the second column, the percentage of test cases that ended in a fail state is given.

As can be seen, in nine out of ten incorrect implementations, errors are detected. In all test cases, only requests are sent that are part of the specification, i.e., only requests for votes by known voters are sent. Because we did not specify that unknown voters are forbidden to vote, errors in the implementation that allow other persons to vote are not detected: the implementation conforms to the specification.

The percentages in Table 10.1 strongly depend on the numbers of voters and lengths of the test traces. Some errors can easily be detected by examining the scores, e.g. incorrect initialisation (4). This error can be detected by traces of length 2: request for the score and inspect the corresponding response. Other errors, however, depend on the number of voters. If the last vote is counted twice, all voters have to vote first, after which the scores have to be inspected. This error can only be detected by executing test traces with at least a length of two times the number of voters plus two.

10.8 Using *DiCons* Specifications

We showed how to formally test Internet applications by modelling them as multi request/response transition systems. In this section we explain how to use a *DiCons* specification as a basis for the tester. We do this by showing how such a *DiCons* specification can be transformed into a multi request/response transition system. We first informally show how this transformation is done after which we give a formalisation.

Given a *DiCons* specification, we explained in Section 9.5 how to transform such a specification into a (possibly infinite) set of traces. By projecting these traces to only the requests and responses, we get a (possibly infinite) set of traces that can be used as test cases. To come to an MRRTS, we make use of the users and session labels: all pairs containing a set of request labels and a set of response labels are concerned with one user in one session.

We construct an MRRTS from a *DiCons* specification in two steps. First we construct an LTS from the specification using the operational semantics of *DiCons*. This LTS equals the term deduction system induced by the deduction rules given in Section 9.4. Next, we construct an MRRTS from the LTS by abstracting from internal actions and mail actions and by partitioning the set of labels. In the remainder of this section we have a detailed look at these steps.

10.8.1 From *DiCons* Specifications to LTSs

The deduction rules of *DiCons* as given in Section 9.4.2 induce a labelled transition system where we write transition $s \xrightarrow[k]{a} t$ as a transition from s to t with label $\langle a, k\rangle$:

$$s \xrightarrow{\langle a,k\rangle} t$$

Definition 10.8.1 *(LTS from DiCons specification) Let $x \in \mathbb{X}$ be a process, $\sigma \in \mathbb{S}$ be a state and $t \in \mathfrak{T}$ be a time stamp. Then, the LTS that corresponds to the execution of process x in state σ at time t, $\mathrm{LTS}(\langle x, \sigma, t\rangle)$, is defined as follows:*

$$\mathrm{LTS}(\langle x, \sigma, t\rangle) \quad = \quad \langle S, L, \rightarrow, s_0\rangle$$

where

- *S is the (possibly infinite) set containing all tuples of processes, states and time stamps:*

$$S \quad =_{\mathrm{def}} \quad \mathbb{X} \times \mathbb{S} \times \mathfrak{T}$$

- *L is the set of pairs of action labels and session labels:*

$$L \quad =_{\text{def}} \quad \mathbb{A}_l \times \mathbb{K}$$

with $\mathbb{A}_l$ as defined in Section 9.4.1 on page 121 and $\mathbb{K}$ as defined in Section 7.3.1 on page 73.

- *$\rightarrow \subseteq S \times L \times S$ defines the transition relation. Let $T(DiCons)$ be the term deduction system induced by the deduction rules given in Section 9.4. Then,*

$$\forall s, t \in \mathbb{S}\ \forall \langle a, k\rangle \in L \quad s \xrightarrow{\langle a,k\rangle} t \iff T(DiCons) \models s \xrightarrow[k]{a} t\ .$$

- *$s_0 \in S$ is the initial state, which equals $\langle x, \sigma, t\rangle$.*

The specifications in *DiCons* are not only concerned with interaction behaviour, but they also specify the internal actions. These actions can influence the possible choices made in specifications and thus can influence the interaction behaviour. To abstract from these internal actions, we introduce a symmetric equivalence relation on states. This is explained in more detail in Definition 10.8.5 on page 171.

10.8.2 From *DiCons* LTSs to MRRTSs

Since we aim at black-box testing of Internet applications, we abstract from internal actions and mail actions: we focus on testing the request/response behaviour of implementations. By projecting the traces to only the request and response labels, this abstraction is achieved.

As mentioned in Section 10.4, we do not take rollbacks into account: we only want traces in the test suite which model correct behaviour of the implementation. By allowing rollbacks, the alternating request/response property of traces is no longer ensured. Therefore, we do not include rollback labels in the LTS. We turn an LTS $\langle S, L, \rightarrow, s_0\rangle$ into an LTS without rollback transitions, $\langle S, L, \rightarrow, s_0\rangle/\mathcal{R}$:

Definition 10.8.2 *(LTS without rollback transitions) Let $\langle S, L, \rightarrow, s_0\rangle$ be an LTS. Then, the LTS without rollback transitions, $\langle S, L, \rightarrow, s_0\rangle/\mathcal{R}$, is defined by*

$$\langle S, L, \rightarrow, s_0\rangle/\mathcal{R} \quad =_{\text{def}} \quad \langle S, L, \rightarrow', s_0\rangle$$

where

$$\rightarrow' \quad = \quad \rightarrow \setminus \{(s, l, s') \mid s, s' \in S, l \in \{\mathcal{R}_V \mid V \subseteq \mathbb{V}\}\}\ .$$

Locking actions wrap communication actions and internal actions. So they might also model requests and responses. By dropping lock counters and updated valuation sets, we transform them into the original actions. To do this, we introduce the function unwrap.

Definition 10.8.3 *(Unwrap lockable actions) Let* $\mathbf{a} \in \mathbb{A}_l$ *be an action label and* $k \in \mathbb{K}$ *be a session label. Then, unwrapping the label* $\langle \mathbf{a}, k \rangle$ *by stripping off the lock counter and updating the valuation set from locking labels,* $\mathrm{unwrap}(\langle \mathbf{a}, k \rangle)$*, is defined by*

$$\mathrm{unwrap}(\langle \mathbf{a}, k \rangle) \quad = \quad \begin{cases} \langle a, k \rangle & \textit{if } \mathbf{a} = a_{n,V} \in \mathbb{L}_l \\ \langle \mathbf{a}, k \rangle & \textit{otherwise} \ . \end{cases}$$

Before giving the transformation from *DiCons* LTSs to MRRTSs, we first introduce the set of request and response labels.

Definition 10.8.4 *(request and response labels) The set of all possible request and response transition labels,* $\mathbb{R}$*, is defined as follows:*

$$\begin{array}{lll} \mathbb{R} & =_{\mathrm{def}} & \{\, \mathrm{req}.u.\vec{\imath}.\vec{d} \quad \mid u \in \mathbb{U}, i_1, \ldots, i_n \in \mathbb{P}_i, \vec{d} \in \mathrm{type}(\vec{\imath}) \quad \} \cup \\ & & \{\, \mathrm{resp}.u.m.\vec{o} \quad \mid u \in \mathbb{U}, m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \} \end{array}$$

Definition 10.8.5 *(*MRRTS *from DiCons* LTS*) Let* $x \in \mathbb{X}$ *be a process,* $\sigma \in \mathbb{S}$ *be a state and* $t \in \mathfrak{T}$ *be a time stamp. Let* $\langle S', L', \rightarrow', s'_0 \rangle$ *be the LTS without rollback transitions of* $\langle x, \sigma, t \rangle$*:*

$$\langle S', L', \rightarrow', s'_0 \rangle \quad = \quad \mathrm{LTS}(\langle x, \sigma, t \rangle)/\mathcal{R}$$

Then, the MRRTS *modelling the request/response interaction behaviour of process* x *in state* σ *at time* t*,* $\mathrm{MRRTS}(\langle x, \sigma, t \rangle)$ *is defined by the following equation:*

$$\mathrm{MRRTS}(\langle x, \sigma, t \rangle) \quad = \quad \langle S, \mathbb{L}, \rightarrow, s_0 \rangle$$

where

- *S, the set of states in the MRRTS, is defined using an equivalence relation on states. For states* $s, t \in S'$*,* $s \simeq t$ *is the smallest symmetric relation such that*

  $$s \simeq t \quad \equiv \quad \exists l \in L' \ \ s \xrightarrow{l}{}' t \ \wedge \ \mathrm{unwrap}(l) \notin \mathbb{R} \times \mathbb{K}$$

  Then, S, the set of states in the MRRTS is defined as follows:

  $$S \quad = \quad S'/\simeq \ .$$

- $\mathbb{L}$, *the transition labels, partitioned by sets of requests and corresponding responses. The total set of labels, L, contains pairs of request and response labels and session labels:*

$$L \quad = \quad \mathbb{R} \times \mathbb{K}\,.$$

Each element of $\mathbb{L}$ *contains those request and response labels that are concerned with one user in one session. So,* $\mathbb{L}$*, the partitioning of L over unique users and session labels is defined as follows:*

$$\begin{array}{lcl} \mathbb{L} & = & \{\ (\ \{\ \langle \text{req}.c.\vec{\imath}.\vec{d}, k\rangle \quad \mid i_1, \ldots, i_n \in \mathbb{P}_i, \vec{d} \in \text{type}(\vec{\imath})\ \}, \\ & & \quad\ \{\ \langle \text{resp}.c.m.\vec{o}, k\rangle \ \mid m \in \mathbb{M}, o_1, \ldots, o_n \in \mathbb{P}_o \quad \}\ \ ) \\ & & \mid\ c \in \mathbb{U}, k \in \mathbb{K} \\ & & \} \end{array}$$

- $\rightarrow \subseteq S \times L \times S$ *is the transition relation which relates all states in S for which a relation in the LTS without rollback transitions exists that is labelled by a (possibly wrapped) request or response label:*

$$\begin{array}{l} \forall s, t{\in}S\ \forall l{\in}L \\ \quad s \xrightarrow{l} t \\ \Longleftrightarrow \\ \quad \exists s', t' \in S'\ \exists l' \in L'\ \ s \simeq s'\ \wedge\ t \simeq t'\ \wedge\ l = \text{unwrap}(l')\ \wedge\ s' \xrightarrow{l'}{}' t' \end{array}$$

- $s_0 \in S$ *is the initial state, which equals* $\langle x, \sigma, t\rangle$.

10.8.3 Example

In this section we revise the MRRTS given in Section 10.7. In the example, we have a group of voters V and a type $\mathbb{P} = \{\text{for}, \text{against}\}$.

Example 10.8.6 *All voters are allowed to vote once and check the score as often as they want. In DiCons, this can be specified as follows:*

```
1   [ f ↦ 0 : ℕ | [ a ↦ 0 : ℕ |
2       [ V ↦ V : 𝔾 |
3         ||_{v∈V}
4           ⟨⟨[ p ↦⊥: ℙ |
5             v⇉ vote(ε; p) ·
6             (f := f + 1 ◁ p = for ▷ a := a + 1) ·
7             v ⇄ ok(ε)
8           ]⟩⟩
9       ]
10    ||
11      [ V ↦ V : 𝔾 |
12        !_{v∈V}
13          v ⇄ score(f, a)
14      ]
15  ]]
```

In line 1, the number of voters who voted for (f) and against (a) are declared and initialised with 0. Next, two processes are put in parallel (line 10), viz. the voting process in lines 2 to 9 and the checking for the score in lines 11 to 14.

Using the generalised parallel composition operator, we specify in line 3 that all voters can execute the voting session once. The voting itself consists of the voting for or against the proposition by providing the p in line 5. The vote is counted in line 6. Finally, a confirmation is sent to the voter in line 7. Using the transactional operator, we specify that the voting session can be rolled back if something goes wrong.

Checking for the score is specified by the sending of the current score, i.e., the number of voters who voted for and against the proposition. All voters can do this as often as they want to, as stated in line 12 by the use of the replication operator. The score is sent using a reactive push message (line 13), which immediately terminates the session.

We can transform this specification into an MRRTS as explained before. When inspecting Example 10.7.1, we can easily see that the MRRTS constructed for the specification given in Example 10.8.6 is not equal to it: the constructed MRRTS does not have a ¬ok interaction. This is caused by the fact that we make use of the generalised parallel composition, in which identification is implicitly defined. In Example 10.7.1 this identification is explicitly contained in the specification. The result of this finding is that when using a *DiCons* specification, we *do* specify the intended behaviour, however, because of the implicit identification in *DiCons*, test suites generated from the specification can be different from those specified using MRRTSs. By using the specification in Example 10.8.6 as a basis for the generation of test cases, we cannot detect several errors in the implementation, like errors 8, 9 and 10 in Table 10.1 on

page 168: the test suites generated from Example 10.8.6 do not contain test cases in which a user is trying to vote more than once. So, the *DiCons* specification to use depends on the errors we want to detect.

Below, we give a specification which corresponds the the MRRTS given in Example 10.7.1.

Example 10.8.7 *Let* V *be a group of voters. Then, the DiCons specification from which an MRRTS can be constructed that corresponds to the* MRRTS *of Example 10.7.1 is defined as follows:*

```
1  [ W ↦ ∅ : 𝔾 | [ f ↦ 0 : ℕ | [ a ↦ 0 : ℕ |
2      [ V ↦ V : 𝔾 |
3        !_{v∈V}
4          ⟨⟨[ p ↦⊥: ℙ |
5            v⇀ vote(ε; p) ·
6            (  W := W ∪ {v} ·
7              (f := f + 1 ◁ p = for ▷ a := a + 1) ·
8              v ⇄ ok(ε)
9            ) ◁ v ∉ W ▷ (
10             v ⇄ ¬ok(ε)
11           )
12         ]⟩⟩
13     ]
14   ||
15     [ V ↦ V : 𝔾 |
16       !_{v∈V}
17         v ⇀ score(f, a)
18     ]
19 ]]]
```

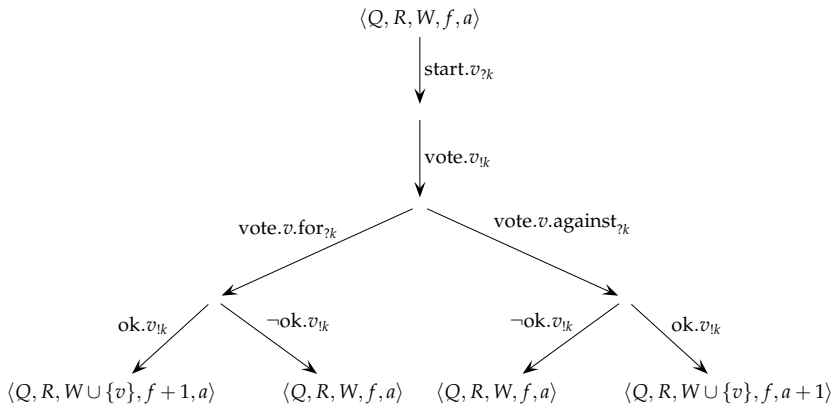
To complete this chapter, we show that the MRRTS constructed from the specification in Example 10.8.7 corresponds to the MRRTS specified in Example 10.7.1.

Lemma 10.8.8 *The* MRRTS *constructed from the specification in Example 10.8.7 corresponds to the* MRRTS *specified in Example 10.7.1.*

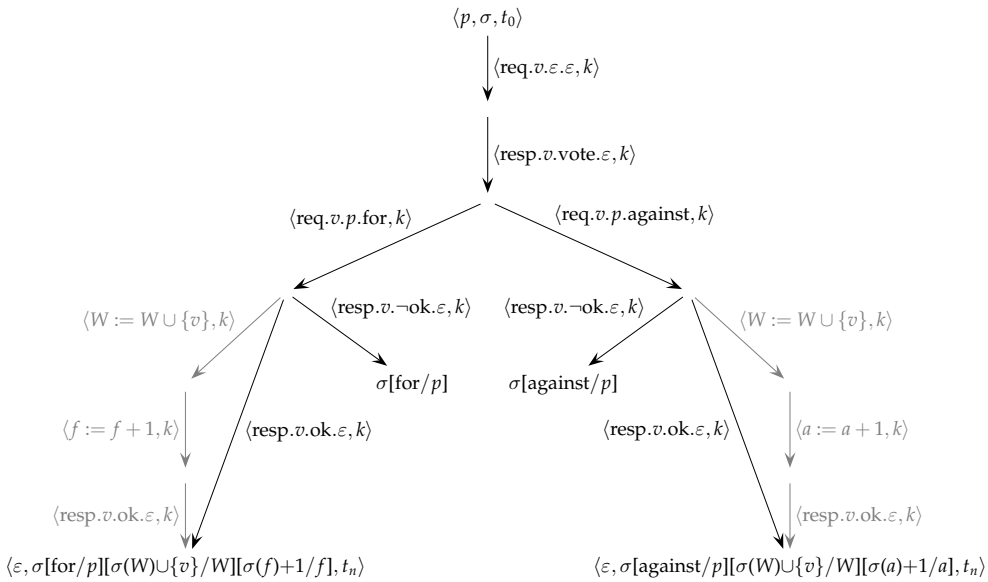
Proof To prove this lemma, we first show that the sessions themselves are bisimular, i.e., that a bisimulation relation between the states and transitions exists if no interleaving takes place. We do this by giving the process graphs induced by the deduction rules of both models. Next, we show that interleaving does not affect this relation.

Step 1. Relation on vote sessions

The process graph for a vote session induced by the deduction rules given in Example 10.7.1 is the following graph:



Let p be the vote process given in lines 4 to 12 of Example 10.8.7 and σ be the state when starting process p at time t_0. Then, the process graph induced by the deduction rules of *DiCons* is given below. Note that we abstract from internal actions when turning the graph into an MRRTS:



As can be seen, these processes show similar behaviour by relating the nodes and by using the following relation $\sim$ on transition labels:

$$\begin{array}{rcll} \text{start}.v_{?k} & \sim & \langle \text{req}.v.\varepsilon.\varepsilon, k\rangle & \\ \text{vote}.v.\text{p}_{?k} & \sim & \langle \text{req}.v.p.\text{p}, k\rangle & \text{for p} \in \mathbb{P} \\ \text{vote}.v_{!k} & \sim & \langle \text{resp}.v.\text{vote}.\varepsilon, k\rangle & \\ \text{ok}.v_{!k} & \sim & \langle \text{resp}.v.\text{ok}.\varepsilon, k\rangle & \\ \neg\text{ok}.v_{!k} & \sim & \langle \text{resp}.v.\neg\text{ok}.\varepsilon, k\rangle & \end{array}$$

There is a one-to-one correspondence on valuations in state stack σ and elements in the states of the MRRTS in Example 10.7.1.

Step 2. Relation on score-checking sessions

The score-checking sessions consist of only one request and one response. The process graph induced by the deduction rules of Example 10.7.1 is given on the left-hand side, the graph induced by the deduction rules of *DiCons* on the right-hand side. Process q equals the score checking process given in line 17 of Example 10.8.7.

$$\begin{array}{ccc} \langle Q, R, W, f, a\rangle & & \langle q, \sigma, t_0\rangle \\ \downarrow \text{score}.v_{?k} & & \downarrow \langle \text{req}.v.\varepsilon.\varepsilon, k\rangle \\ \downarrow \text{score}.v.f.a_{!k} & & \downarrow \langle \text{resp}.v.\text{score}.(\sigma(f),\sigma(a)), k\rangle \\ \langle Q, R, W, f, a\rangle & & \langle \varepsilon, \sigma, t_n\rangle \end{array}$$

As is the case with vote sessions, states in the score-checking sessions in both examples can be related. We use the following relation on transition labels of both examples:

$$\begin{array}{rcl} \text{score}.v_{?k} & \sim & \langle \text{req}.v.\varepsilon.\varepsilon, k\rangle \\ \text{score}.v.f.a_{!k} & \sim & \langle \text{resp}.v.\text{score}.(f,a), k\rangle \end{array}$$

Note that $\sigma(f)$ and $\sigma(a)$, the evaluations of f and a in state σ of the *DiCons* process, correspond to the f and the a in the states of Example 10.7.1.

Furthermore, as a result of the implicit adding of session labels, no two sessions are labelled by the same label. This means that all labels in all vote sessions and all score-checking sessions differ. As a result, the k in all relations on $\langle \text{req}.v.\varepsilon.\varepsilon, k\rangle$ differs:

$$\begin{array}{rcl} \text{start}.v_{?k} \sim \langle \text{req}.v.\varepsilon.\varepsilon, k\rangle & \Longrightarrow & \text{score}.v_{?k} \not\sim \langle \text{req}.v.\varepsilon.\varepsilon, k\rangle \\ \text{score}.v_{?k} \sim \langle \text{req}.v.\varepsilon.\varepsilon, k\rangle & \Longrightarrow & \text{start}.v_{?k} \not\sim \langle \text{req}.v.\varepsilon.\varepsilon, k\rangle \end{array}$$

Step 3. Interleaving of sessions

When sessions interleave, things can go wrong: users might be able to vote twice, votes might get lost, or incorrect scores might be sent. We prove that if one voter

votes simultaneously in two sessions, the vote is counted only once. Furthermore, we prove that simultaneous votes by two different users are processed correctly and we prove that the results sent when checking the score correspond to the score-checking response given in Example 10.7.1.

If two vote sessions of the same voter v interleave, the first three actions can be executed for both processes. When reaching the processing of the vote, the first thing done by both processes is the inspection of W and the adding of voter v to W. Since we have a transactional process, this update can only be done by one of the two voters, causing the other process to get locked. As a result, when the first transaction terminates, the second process cannot execute the update since v is already in V. So $v \notin W$ evaluates to false and therefore the second process is ended by a resp.v.$\neg$ok.ε interaction.

Interleaving of two processes by different voters do not cause any problem since the updating of both f and a are synchronised by the transactional operator.

When interleaving of votes take place with a score-checking session, the values of f and a can get updated in between the receiving of a request for the score and the sending of the response containing the score. The f and a sent are always at least the values when receiving the request (f_k and a_k) since both f and a are increased only. This corresponds to $f_k \leq f'$ and $a_k \leq a'$ in the premise of the deduction rule for the score-checking response in Example 10.7.1. The score responded can be at most f_k and a_k, increased by the number of vote sessions which are in a state between increasing the f or a and the sending of response resp.v.ok.ε. This exactly corresponds to $f' \leq f + (\#v{\in}V\exists k{\in}\mathbb{K}\ \langle k, v, 2, \text{for}\rangle \in Q \wedge v \notin W)$ and $a' \leq a + (\#v{\in}V\exists k{\in}\mathbb{K}\ \langle k, v, 2, \text{against}\rangle \in Q \wedge v \notin W$ in the premise.

This completes the proof that both MRRTSs describe similar behaviour. □

10.9 Related Work

Automatic test derivation and execution based on a formal model has been an active topic of research for more than a decade. This research led to the development of a number of general purpose black box test engines. However, the domain of Internet applications induces some extra structure on the interacting behaviour of the implementation which enforces the adaptation of some of the key definitions involved. Therefore, our work can be seen as an extension to and adaptation of the formal testing framework as introduced in [BAL+90, Tre94, ISO96]. The major difference stems from our choice to model an Internet application as a multi request/response transition system. We expect that existing tools (such as TorX [BFV+99]) can be adapted

to this new setting. The reader may want to consult [BT00] for an overview of other formal approaches and testing techniques.

Approaching the problem of testing Internet applications from another angle, one encounters methodologies and tools based on capture/replay, which can be found in e.g. [Col03, OSGL03]. In the case of capture/replay testing, test cases are produced manually and recorded once, after which they can be applied to (various) implementations. These tools prove very beneficial for instance for regression testing. However, automatic generation of test cases has several advantages. In general it proves to be a more flexible approach, yielding test suites that can be maintained more easily and more completely and test suites can be generated more quickly (and thus more cheaply). The main disadvantage of automatic black-box testing is that it requires a formal model of the implementation under test.

A methodology that comes very close to ours is developed by Ricca and Tonella [RT01]. The starting point of their semi-automatic test strategy is a UML specification of a Web application. This specification is manually crafted, possibly supported by re-engineering tools that help in modelling existing applications. Their UML specification is at the same level as our *DiCons* specification. However, *DiCons* is much more expressive, since it allows e.g. the specification of transactions and the sharing of data between sessions. Moreover, the fact that parallel sessions can influence each others behaviour led us to the introduction of MRRTSs. Phrased in our terms, Ricca and Tonella consider RRTSs as their input format (which they call *path expressions*). Another difference is that we perform black-box testing, whereas they consider white-box testing. This implies that their approach considers implementation details (such as cookies), while we only look at the observable behaviour. White-box testing implies a focus on test criteria instead of a complete testing algorithm. Finally, we mention the difference in user involvement. In our approach the user has two tasks, viz. building an abstract specification and instantiating the test adapter which relates abstract test events to concrete HTTP-events. In their approach, the user makes a UML model, produces tests and interprets the output of the implementation. For all of this, appropriate tool support is developed, but the process is not automatic. In this way derivation and execution of a test suite consisting of a few dozens of tests takes a full day, whereas our on-the-fly approach supports many thousands of test cases being generated, executed and interpreted in less time.

Jia and Liu [JL02] propose a testing methodology which resembles Ricca and Tonella's in many respects, so the differences with our work are roughly the same. Their focus is on the specification of test cases (by hand), while our approach consists of the generation of test cases from a specification of the intended application's behaviour. Their approach does not support on-the-fly test generation and execution. Like Ricca and Tonella, their model is equivalent to RRTSs which makes it impossible to test parallel sessions (or users) that share data.

Wu and Offutt [WO02] introduce a model for describing the behaviour of Web applications, which can be compared with the *DiCons* language. In contrast to the model presented in this chapter, their model supports the use of special buttons that are available in most Web browsers. The main difference with our model is that they focus on stateless applications, i.e., responses only depend on the preceding request. We model stateful applications which are based on sessions executed in parallel.

Another functional testing methodology is presented by Niese, Margaria and Steffen in [NMS02]. Where we focus on modelling Internet applications only, they model other subsystems in the system under test as well. In their approach, test cases are not generated automatically, but designed by hand using dedicated tools. Test execution takes place automatically via a set of co-operating subsystem-specific test tools, controlled by a so-called test co-ordinator.

Our research focuses on conformance testing only. Many other properties are important for the correct functioning of Web applications, such as performance, user interaction and link correctness [BFG02]. Testing such properties is essentially different from conformance testing. They focus on *how well* applications behave instead of *what* they do. Plenty of tools are available for performance testing, e.g., [Die01, Ful02].

10.10 Conclusions

Due to the focus of *DiCons* on interaction, rather than on presentation, it is likely that developers prefer to use a less formal approach that supports the need for a nice user interface. However, our current research shows that development of a formal interaction model, like in *DiCons*, still has benefits. Our research shows that there is a point in making a formal model, even if it is not used for generating Internet applications, since a formal model can be used for (automated) conformance testing of the application. In the near future, this might become a legal claim for, e.g., Internet votes.

The input of the testing process described in this chapter is a multi request/response transition system which is a theoretically simple model, but which is very hard to use in practice for the specification of real applications. Since *DiCons* is targeted to specify Internet applications and since its operational semantics can be transformed into an MRRTS, we can connect the *DiCons* execution engine to our prototype testing tool. This provides an effective tool for automated on-the-fly conformance testing of Internet applications without having to give complex specifications using e.g. MRRTSs.

As the development of a formal model of an Internet application is quite an invest-

ment, we expect that only in cases where it is vital that the application shows the correct interaction behaviour, automated formal testing will be applied. However, there will be a huge gain in reliability and maintainability of the application (e.g. because of automated regression testing), compared with e.g. capture and replay techniques.

Although we have only built a simple prototype, we can conclude that the proposed testing approach works in practice, since it quickly revealed (planted) errors in erroneous implementations. Interestingly enough, playing with the prototype made it clear that the response times in the HTTP-protocol are much slower than in traditional window-based applications, resulting in less test runs per time unit. We cannot foresee if the unreliability of an Internet connection will prevent us from executing lengthy test runs over the Internet.

An interesting point is that the actual HTTP-response of an Internet application has to be matched against the expected abstract event from the specification. In our current prototype tool we simply scan for the occurrence of certain strings, but this does not seem to be a safe and generic approach. Future research should answer the question of how to match actual HTTP-replies against abstract events. This problem is more or less the inverse of the presentation problem of *DiCons* as is explained in Chapter 11, where *DiCons* specifications serve as a basis for generating executable code.

11

Generation of Internet Applications

Apart from testing Internet applications against a formal specification, specifications can also serve as a basis for a compiler, turning specifications into running applications. In this chapter, we investigate how such a compilation can be done; we do *not* give a complete specification of a compiler.

In an early stage of our project we made a feasibility study [Bee00] which we discuss in this chapter. This study was based on a preliminary version of *DiCons*. In Section 11.1 we introduce how the current programming of Internet applications is done and what techniques are available. These techniques lead to many possibilities but of course they also introduce insuperable limitations. In Section 11.2 we show how a *DiCons* specification can be turned into a running application. Apart from the result that a compiler should produce, we also shortly mention how Java can be used for programming the compiler itself in Section 11.3. The feasibility study results in suggestions for future work in Section 11.4, after which we draw some final conclusions in Section 11.5.

11.1 Programming Internet Applications

Although programming Internet applications can be compared with programming stand-alone window-based applications, there are many differences, as explained in Section 3.3. The main differences between the two styles of programming depend on the different ways of user interaction with the applications.

In the specifications given in this thesis we abstracted from the actual contents of all interactions. We stated that HTTP is used for the communication and that input and output parameters can be attached.

As a start, we explain in Section 11.1.1 what interactions actually look like. Interactions are represented using hypertexts, which are pieces of text extended with interactive elements. We explain how the actual presentation of hypertext documents can be detached from the HTML document. In this way, e.g. fonts, colours and images can be specified on a central, application-independent, place.

We show how this representation can be extended with input parameters using links and Web forms in Section 11.1.2. Apart from that, we also show how we can make use of client-side scripting languages to pre-process information filled in in Web forms. In this way, we can reduce the number of interactions containing incorrect values for the input parameters.

Next, in Section 11.1.3, we explain how the representation can be pre-processed such that output parameters of the interactions can be included. In this way, static hypertexts can be adapted such that they become dynamic. Several languages are available for doing this adaptation. We shortly summarise them and, to give an example, we show how Java can be used for this purpose.

We explain how these dynamic hypertext documents can be sent to interacting clients in Section 11.1.4. Since we make use of HTTP, parameters can be attached to requests in several ways, which will also be explained. Furthermore, we explain how subsequent interactions can be combined into sessions. Since session management is not contained in the HTTP protocol itself, special facilities are available to implement this, which we discuss in short.

11.1.1 Hypertext Documents

Many documents can be retrieved over the Internet. They can be of any type, e.g., PostScript files, PDF files or plain ASCII files. However, the documents we focus on are represented by so-called *hypertexts*. A hypertext is a piece of text containing links to other (hyper)texts. Selecting such a link causes the text to be replaced by the linked text. Apart from these *hyperlinks*, a hypertext can also contain other elements, like headers, images, tables, and interactive elements such as fields that can be filled in, boxes that can be checked and menus from which an element can be chosen. Of course, since many possible elements can be included in hypertexts, a specification language for such documents is needed. The most widely used language for doing this is the *Hypertext Markup Language* (HTML) [RLHJ99].

Hypertext Markup Language

Hypertext Markup Language [RLHJ99] is an application of the Standard Generalised Markup Language (SGML) [Int86], an international standard formally called ISO 8879 by the International Organization for Standardization (ISO). SGML is a formal definition for defining data and document interchange languages. HTML is a standard which is maintained by the World Wide Web Consortium (W3C).

An HTML document is a text file containing so-called *tags*. These tags are used for embracing pieces of text which gives the text a special meaning. The syntax of an opening tag is `<identifier>`. Its corresponding closing tag is denoted by `</identifier>`. The opening tag can be extended with so-called *attributes*, specifying properties of the tag. For example `<table width="50%">` specifies that a table is opened having a width of 50% of the document it is contained in. Tags can be nested, result in in a completely structured document. A small example of an HTML document together with its representation is given in Figure 11.1.

Adapting Styles

By using attributes of tags, we are able to adjust styles of the text, as shown in the example in Figure 11.1. We used the attribute `center` to centre the header text. Plenty more style elements can be set, e.g., fonts, colours, margins and other aspects of a hypertext document, without having to change its structure. These visual design issues can be addressed separately from the logical structure using so-called *Cascading Style Sheets* (CSS). In this way it is easier to give a set of related hypertext documents a similar look and feel. By putting the style in a separate document, adapting the style can take place in a central place. The style sheet to be used is referred to in the header of the hypertext document using the `<link>` tag. An example of a style that adapts all texts in the body part of a document is given below.

```
body {
  font-family: Helvetica;
  font-size: 10pt;
  font-style: normal;
  text-align: justify;
  margin: 5px;
}
```

The font is set to Helvetica, with a size of 10 points and normal thickness. Text is justified with a margin of 5 pixels. If this style is attached to a hypertext document, the body is adapted so that this style is used. An overview of what can be modified using style sheets can be found at [WLB99].

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Example HTML Specification</title>
  </head>
  <body>
    <h1 align="center">This is an example</h1>
    <p>
      This is an example of an HTML specification.
      The header above is centred. The table below,
      with a width of 60% of the body's width, has
      three rows and two columns.
    </p>
    <table width="60%">
      <tr>
        <th>First column header</th>
        <th>Second column header</th>
      </tr>
      <tr>
        <td>Text in field one</td>
        <td>Text in field two</td>
      </tr>
      <tr>
        <td>Text in field three</td>
        <td>Text in field four</td>
      </tr>
    </table>
    <p>
      The following link leads to the W3C
      home page:
      <a href="http://www.w3c.org/">
        World Wide Web Consortium
      </a>
    </p>
  </body>
</html>
```

This is an example

This is an example of an HTML specification. The header above is centred. The table below, with a width of 60% of the body's width, has three rows and two columns.

First column header	**Second column header**
Text in field one	Text in field two
Text in field three	Text in field four

The following link leads to the W3C home page:
World Wide Web Consortium

Figure 11.1: Example of an HTML document.

Since we are not concerned with representation but only with logic, this is a very useful feature, as the application can generate output to which a style can be attached at a later stage. This ensures that the application does not need to be concerned with the representation at all.

11.1.2 Adding Input Parameters to Hypertexts

Up till now we only looked at fixed elements in HTML documents. However, the interaction primitives introduced in Section 4.3 and Chapter 7 contain both input and output parameters for the communication of data between the application and its users. In this section, we explain how input parameters can be added to hypertexts, such that they can be used to collect data from clients. In the next section, we look at

a way of adding output parameters.

Apart from tags for adding layout and links to other documents, tags are available for adding forms containing interactive elements. These elements can be used for inquiring information from interacting users. A form can be added to an HTML document by using the `<form>` tag. The tag can be extended with several attributes, among which the most important one is the `action` attribute. The action specifies what should be done when the form is submitted, i.e., what should be done with the information provided by the user.

Between the opening and closing tag of a form, input fields of different types can be added using several tags, like `<input>`, `<select>` and `<textarea>`. We summarise the possible elements here in short, together with some HTML examples.

Single-line input fields This type of input is used for the collection of pieces of simple text, like a name, an address or a telephone number. An initial value can be provided.

```
Address: <input type="text" name="address" value="">
```

The example specifies a text field (preceded by the text "Address:") with the empty string as initial value. Note that no closing tag is needed.

Multi-line input fields If more information should be filled in in one text field, a multi-line field can be used. A multi-line input field is called a text area.

```
<textarea name="remark">Type your remarks here</textarea>
```

The example specifies a multi-line text field with the string "Type your remark here" as initial value.

Radio buttons If the user is asked to choose one item from a limited number of elements, radio buttons can be used. Radio buttons with the same name are grouped and only one of them can be chosen.

```
<input type="radio" name="sex" value="male"> Male
<input type="radio" name="sex" value="female"> Female
```

This example shows two radio buttons from which only one can be selected. The first button is followed by the text "Male", the second one by "Female".

Checkboxes A checkbox gives the user the possibility to select one or more options of a limited number of choices.

```
Hobbies:
<input type="checkbox" name="read"> Reading books
<input type="checkbox" name="play"> Playing games
<input type="checkbox" name="chat"> Chatting
```

The user can check zero or more of the hobbies.

Menus and lists Apart from radio buttons and checkboxes it might be preferable to use a pull-down menu or a list. A pull-down menu takes less space than radio buttons since only the selected item is visible. Lists can be useful if multiple elements might be selected. Using the `size` attribute, the number of concurrently shown elements can be set.

```
Sex:
<select name="sex">
  <option value="male">Male</option>
  <option value="female">Female</option>
</select>
```

This example specifies a pull-down menu where the choice for male or female can be made.

```
Hobbies:
<select name="hobby" size="3" multiple>
  <option value="read">Reading books</option>
  <option value="play">Playing games</option>
  <option value="chat">Chatting</option>
</select>
```

This second example shows a list with three elements from which zero or more can be selected.

Buttons Buttons can be added, which can be pressed. This may cause the form to be e.g., submitted or cleared.

```
<input type="submit" value="Submit form">
<input type="reset" value="Clear form">
```

The example shows how a submit button with the text “Submit form” and a reset button with the text “Clear form” can be added to a form.

Apart from the examples given above, more interactive elements can be included like password fields, images, etcetera. So by adding forms containing input elements to a hypertext, input parameters of the interaction can be implemented.

Client-Side Scripting Languages

We developed *DiCons* from the point of view that clients have no ability to do calculations, so we put all logic in the application on the server side. However, Web browsers become more and more complex applications and most Web applications make use of client-side scripting. Therefore, we discuss this scripting technique here in short, and show how we can make use of it in generating *DiCons* specifications.

With client-side scripts one has the ability to run programs, catch events, etc. without having to communicate with the server. Scripting languages can be directly embedded into a hypertext document. These elements can e.g. respond to user actions like clicking a button or they can generate dynamic documents where content and/or presentation changes. Furthermore, external programs exist which are executed from within a hypertext document. Such programs, e.g. applets, ActiveX-components and Flash movies, are applications which run inside a Web browser. We give a short overview of client-side scripting languages.

ECMAScript [Eur99] Nowadays, the Web's only standard scripting language is ECMAScript, whose name comes from the European Computer Manufacturers Association (ECMA). The ECMA is an international, Europe-based industry association dedicated to the standardisation of information and communication systems. The first scripting language to fully conform to ECMAScript is Microsoft JScript. Except for JScript two other major scripting languages are VBScript and JavaScript.

Microsoft JScript [Rog01] JScript is an object-based, loosely typed language. This means that data types of variables do not have to be declared explicitly. It is Microsoft's variant of JavaScript. Therefore, it is not a cut-down version of any other existing language. JScript is a pure interpreter that processes source code that is directly embedded in hypertext documents. JScript communicates with host applications using ActiveX Scripting.

Microsoft Visual Basic Scripting Edition [LCP03] Microsoft Visual Basic Scripting Edition, VBScript in short, is a subset of the Microsoft Visual Basic programming language. It can be used in Web browsers and other applications that use Microsoft ActiveX Controls. VBScript also communicates with host applications using ActiveX Scripting.

JavaScript [Fla98] JavaScript is the most used scripting language. JavaScript is not Java! JavaScript is a scripting language developed by Netscape for use within HTML documents. Java on the other hand, is an object-oriented programming language developed by Sun Microsystems that can be used to create (standalone) applications. A Java-enabled browser is not automatically a JavaScript-enabled browser, and vice versa, since the two technologies require entirely

JavaScript	Java
Interpreted (not compiled) by client.	Execution via compiled byte-codes.
Object-based. No distinction between types of objects. Inheritance is through the prototype mechanism and properties and methods can be added to any object dynamically.	Object-oriented. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Variable data types not declared (loose typing).	Variable data types must be declared (strong typing).
Dynamic binding. Object references are checked at runtime.	Static binding. Object references must exist at compile-time.

source: JavaScript Guide, Netscape Communications Corporation, 1997.

Table 11.1: JavaScript compared to Java.

separate interpreters. In Table 11.1 an overview of the main differences between Java and JavaScript is given.

With respect to *DiCons* we can use scripting languages for client-side checking of constraints on data which has to be filled out in Web forms. To give an example, suppose that a client has to fill in a Dutch zip code (such as 5600 MB). Then, the syntax of the zip code can be checked client-side before it is sent to the server. This can be done as follows:

```
<script language="JavaScript">
  function check() {
    var zipcode = document.getElementById("zipcode").value;
    if ( zipcode != null && /[1-9][0-9]{3} ?[a-zA-Z]{2}/.test(zipcode) ) {
      return true;
    } else {
      alert("This zip code is not correct.");
      return false;
    }
  }
</script>
<form onSubmit="return check();">
  Zip code: <input type="text" id="zipcode">
  <input type="submit" value="Submit">
</form>
```

We define a function in JavaScript specifying that the zip code filled in in the text field should conform to the regular expression representing a Dutch zip code: four

digits of which the first one should not be a 0, possibly followed by a space and ended by two letters. On submission of the form, this function is executed. If the zip code is wrong, an alert is shown with the text "This zip code is not correct". If the function returns `true`, the form is submitted, otherwise, the submission is cancelled.

Using client-side checks reduces the number of interactions. However, not all clients have the possibility to execute client-side code, and there is always a possibility to disable the client-side scripting feature of a Web browser. Therefore, a server-side check for correctness of the values received is always needed.

To conclude, using forms with input fields, we have a mechanism for implementing the input parameters of *DiCons* interactions. Next, we have a look at how output parameters can be implemented.

11.1.3 Adding Output Parameters to Hypertexts

Where the input parameters can be presented by elements in the hypertext documents, output parameters should be evaluated and their value should be sent to the client, as is specified in the operational semantics in Table 9.6 on page 124. This asks for a server-side pre-processor, making the hypertexts dynamic. Several techniques and specification languages are available for doing this pre-processing. We discuss some of them in short.

Common Gateway Interface [RC04] A *Common Gateway Interface (CGI)* script is a program that is stored on a Web server and executed on the Web server in response to a request from a user. A CGI script file is written in a programming language which can be either compiled to run on the server or interpreted by an interpreter on the server. Examples of languages used to write CGI scripts are *C*, C^{++} and *Perl*. Each time a CGI script is requested, the server must create a new process, run the script and terminate the (just created) process. So, like HTTP, CGI scripts are both connection-less and stateless.

ColdFusion [Mac05] Macromedia's *ColdFusion* is a cross-platform Web application server. It provides a platform for building and deploying Web systems that integrate browser, server, and database technologies. Database access takes place with ColdFusion templates. Such templates look like normal HTML documents. By using special tags, a template sends an SQL[1] query to a database and the result is sent to the user.

Active Server Pages [Mic05] *Active Server Pages (ASP)* is a language-independent framework designed by Microsoft for efficient coding of server-side scripts that are designed to be executed by a Web server in response to a user's request for

[1] *Structured Query Language (SQL)* query, allows users to access data in a relational database.

a URL. ASP is actually an ActiveX Scripting Host and can, therefore, be written in JScript or in VBScript. ASP makes accessing databases easier using *ActiveX Data Objects (ADO)*, which allows easy access to any ODBC[2] compliant data source. ASP can be seen as Microsoft's variant of ColdFusion.

PHP: Hypertext Preprocessor [PHP05] PHP can be compared with ASP, using a *Perl*-like language as its server-side scripting language. PHP is mostly used on Linux/Apache and is the open source alternative to ASP.

Servlets and Java Server Pages [SUN05a, SUN05c] Java servlets are server-side Java applications that can be compared with CGI scripts. Main difference is that servlets do not run a separate process for every single request. A servlet stays in memory between requests where CGI scripts need to be loaded and started every time a request is placed. A servlet can handle client-side HTTP requests. Such a request can e.g. be posting a Web form or getting an HTML page.

A servlet has an internal state. This state can be used to respond in different ways to one and the same HTTP request. In such a way an application can be executed sequentially. If, e.g., the initialisation phase of an application has ended, the servlet might respond by sending a Web page to the initiator in which it thanks the initiator for initialising the application.

Since servlets are Java classes, database access can be added using JDBC[3].

A JSP page [SUN05c] is a hypertext containing Java code. When a JSP document is accessed for the first time via a Web browser, the document is transformed into a servlet which subsequently is executed. So JSPs can be considered servlets.

Since one servlet can be used for the handling of several requests, we focus on servlets in the remainder of this chapter. As mentioned, we aim at compiling a complete *DiCons* specification into a running application, which can be done by turning it into a servlet.

We give some examples of how output parameters can be added to hypertexts using JSP documents (i.e. servlets) as a basis. Output parameters should be evaluated and their values should be inserted somewhere in the hypertext which is sent to the interacting client. A JSP document actually is an hypertext document containing tags with Java code in it. This Java code can, amongst other things, write data to the hypertext document. We show how this can be done by means of an example.

Let (name $\mapsto$ "Harm" : String) be a valuation in the state of the application. If we want to send a hypertext document containing the text "Hello Harm" to the client,

[2] *Open Database Connectivity (ODBC)* is a widely accepted application programming interface (API) for database access.

[3] *Java Database Connectivity (JDBC)* is the Java counterpart of ODBC.

this can be done using the following code:

```
Hello <%=name%>
```

In the example in Figure 9.1 on page 137 the voters can select a candidate from a group of candidates. This combines an input parameter with an output parameter. So we need to add a form element based on an output parameter. This example specifies a pull-down menu which is filled with the list of candidates:

```
<select name="candidate">
  <% for (int i=0; i<Candidates.length; i++) { %>
    <option><%=Candidates[i]%></option>
  <% } %>
</select>
```

A pull-down menu named "candidate" is specified using the `select` tag. The options, i.e. the possible candidates, are added using a for loop. The `<%..%>` tag can be used for inserting arbitrary Java code, so the opening brace `{` can be in a different tag than its corresponding closing brace `}`. The code above is compiled into the following code when the JSP document is transformed into a servlet:

```
out.write("<select name=\"candidate\">\r\n");
for (int i=0; i<Candidates.length; i++) {
  out.write("<option>");
  out.write(Candidates[i]);
  out.write("</option>\r\n");
}
out.write("</select>\r\n");
```

As can be easily seen, a JSP document is much more readable and thus more suitable for specifying documents than Java (servlet) code.

Now that we have an idea of how both input and output parameters can be added to hypertext documents, we have a look at how these documents can be sent to clients and how values of input parameters can be sent from clients to the server.

11.1.4 Communicating Hypertext Documents

It is not our goal to give a complete outline containing all possible technologies available. The techniques that are described here serve as a base for the introduction of the current Internet application communication. We do not take into account the

lower layers of the TCP/IP model [Bra89], but only focus on those parts that developers are concerned with when developing the specifications we focus on in this thesis, i.e., the application layer.

As mentioned before, the communication we focus on depends on the HTTP protocol. In Section 3.2 we introduced the Hypertext Transfer Protocol.

HTTP [FGM+99] is connection-less, meaning that a client opens a connection, sends a request, receives a response and closes the connection. Furthermore, HTTP is stateless, which means that it has no memory of former connections and cannot distinguish one client's request from the other. This is a problem when specifying sessions with a user, since we need to interconnect the responses and the requests following on that response as explained in Section 4.3 where we introduce the interaction primitives. This connection can be made by adding one or more parameters to the requests containing session identifiers. This adding of parameters can be done using two different request methods, viz. by attaching it to the uniform resource locator (URL) encoding the request (the GET method), or by adding it to the body of the request (the POST method). To give a more detailed explanation of this, we first introduce the format of a URL and the GET and POST methods of HTTP requests.

Uniform Resource Locators

All documents that are available on the Internet, like hypertexts but also images and e.g. PDF documents, can be uniquely located using a so-called Uniform Resource Locator [BMM94]. The (simplified) syntax for a URL that can be accessed using the HTTP protocol is given below.

```
http://host[:port]/[path/]resource_name[#section][?query_string]
```

The host name points to (a part of) a server connected to the Internet. Connection to a server takes place via a port. For HTTP, port 80 is reserved. If a non-standard port is used, an additional port number can be attached. A specific resource name (possibly preceded by a path) can be accessed. Such a resource can, among other things, be a hypertext document or a servlet. If a resource contains different anchors, one can immediately jump to an anchored part of the resource by using the `#` sign followed by the anchor name. An example of such a URL is given below.

```
http://www.win.tue.nl/ipa/archive/springdays2001/Abstracts.html#vanbeek
```

This URL leads to the abstracts in the archive of the IPA Spring Days 2001. It is located at the server named `www.win.tue.nl` and we directly go to the abstract labelled

`vanbeek`.

An important part of the URL is the query string. By using this string one can pass parameters to the resource. Such parameter can be added to the URL by typing them directly in a browser's address field. Another way for adding parameters is by using HTML forms. Using these forms one can type the parameters into a text field as described above and subsequently send them to the server by clicking on a submit button. An example of such a URL is given below.

```
http://www.google.com/search?q=DiCons
```

This URL leads to the result page of a search for the word "DiCons" using the Google search engine. After sending this request, the resource `search` on server `www.google.com` processes this request, evaluating variable `q` to `DiCons`.

The query string can also be used for user authentication. E.g., a user can send a unique identifier as query part of the URL. In the same way, sessions can be implemented. A user can add a unique identifier to several requests which indicates that these requests form a session.

The HTTP Request Methods

According to the standard [FGM+99], eight HTTP request methods are available: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT. However, when implementing Internet applications, we are mostly concerned with only the GET and POST methods. Therefore, we have a closer look at these two.

GET method Using a GET method, all information is encoded in the request URL. Parameters can be attached to the request using the query string as explained above. The query string encodes pairs of identifiers and values, coupling them using the equals sign (=). The pairs of identifiers and values are composed into a query string using the ampersand sign (&). Since in this way several characters (e.g., =, & and spaces) cannot be used in the values, the query string is encoded using two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code for special characters. So, e.g., the request

```
GET /resource?name=Harm%20van%20Beek&email=harm@win.tue.nl HTTP/1.1
```

is a GET request (from a client using HTTP version 1.1) for a resource containing the query string encoding two valuations, viz.

$$\begin{array}{l} \text{name} \mapsto \text{“Harm van Beek”} : \text{String} \quad \text{and} \\ \text{email} \mapsto \text{“harm@win.tue.nl”} : \text{String} \ . \end{array}$$

POST method If large pieces of text or binary data need to be included in a request, using query strings is not sufficient. In that case, the data to be sent is put in the requests' body using the POST method. The query string above can be put directly in the request body, resulting in the following POST request:

```
POST /resource HTTP/1.1
Content-Length: 44

name=Harm%20van%20Beek&email=harm@win.tue.nl
```

In the next section, we explain how requests and responses can be combined into sessions.

Clustering Interactions in Sessions

The problem with implementing sessions is that HTTP is both a connection-less and stateless protocol. In this section, we present a simple but very useful technique for adding states to HTTP. Our goal is to cluster subsequent requests and responses into sessions as explained in Section 4.3 and Chapter 7. This can be done by parameterising requests with a unique session label. At the moment that a first request is sent to the application, a unique session label is constructed. This label is added to all responses and all forms and links in the responses are extended with the session label. In this way, all subsequent requests contain this unique label and can therefore be identified as requests in the session. So how can this label be added to links and forms?

The first method for implementing this is putting the session label in the request string of all links, coupling it to a unique identifier, e.g. `sessionid`. We then get links of the form:

```
http://www.example.com/resource?sessionid=6861726D4077696E
```

If we use forms, we can add the session label to a hidden field which is added to the URL when submitted in case of a GET method, or put in the request body when using the POST method. The method can be defined as an attribute of the `<form>` tag:

```
<form method="POST" action="resource">
  <input type="hidden" name="sessionid" value="6861726D4077696E">
  ...
</form>
```

In this way, the session label is not visible to the client.

A second method for coupling subsequent requests is by setting a so-called cookie as soon as the first response is sent. A cookie is a piece of data, e.g. a session label, which is sent to a Web browser by a Web server. The browser stores the data in a text file. Then, this data is included in each request that is sent by the browser to the server. Advantage of using cookies is that they have to be set only once and thus the session label does not have to be attached to each link and in each form. Disadvantage is that users can disable cookies, not allowing servers to set them.

It is possible to tell the browser what will be done with the cookie so that, e.g., cookies for session managements can be allowed and cookies storing personal information can be denied. For this purpose the Platform for Privacy Preferences is initiated by the World Wide Web Consortium. They provide a standard, The Platform for Privacy Preferences 1.0 Specification [CLM+02], in which it is specified how to express the privacy practices of an Internet application. This means that it can be specified what data of a user is collected and how this data is used. If in this policy it is stated that data is only used for session management, cookies can be accepted. More information on how to communicate the privacy practices and policies, for example using HTTP headers, can be found in [CLM+02].

11.2 Generating Executable Code

Now that we have an idea of what hypertext documents look like and how they are used for communication, we have a closer look at the different aspects of a *DiCons* specification. We show in short how they can be implemented using the Java servlet technology [SUN05a]. As mentioned, a servlet can be seen as a Java object, which implements a (part of a) Web service. The interface of a servlet contains amongst other methods the *service* method, which is extended with two parameters, viz. the request and the response. Each time that a request is sent to a servlet, this method is called. The complete interface of a servlet can be found in the Servlet Application Programming Interface (API) specification which is available at [SUN05a]. The request parameter is a Java object which wraps the actual HTTP request. It has several methods for e.g. getting the HTTP headers, information on the path, names and values of parameters added in the query string, and session information. The response parameter is an object which wraps the response to be sent to the requesting client. This parameter is an output parameter which has several methods for setting information on the response, for example headers can be added, data can be attached to the body of the response, and cookies can be included. Each instance of a servlet can process multiple requests in parallel as long as the instance exists.

In Section 11.2.1 we handle the compiling of the actions in a *DiCons* specification, i.e.,

the elements of the alphabet. Next, is Section 11.2.2 we discuss how the operators can be compiled into a Java implementation.

11.2.1 Alphabet

The actions contained in the alphabet are internal actions (which can be of any form) and the communication primitives.

Internal Actions

In Section 9.2 we stated that an internal action can be any action, from simple, self-explaining to very complex. Furthermore, we did not put any restrictions on the types of variables and thus on result types and types of the arguments of functions. If we compile specifications into Java servlets, the easiest way for implementing internal actions is by defining a Java function for each of them and by calling them when the action is executed.

Furthermore, we did not put any restrictions on the representation chosen for specifying types and internal actions. We allow e.g. the use of notation $N \mapsto \emptyset : \mathcal{P}(\mathbb{N})$. The format of types and functions has to be restricted, for example to only the use of Java types and Java functions. In this way, we can copy them easily into the Java code generated from the specification. Syntax errors in the internal actions will be discovered when compiling the generated servlet into Java byte-code. Run-time errors in the sources of internal actions can be detected by executing the servlet.

Tools exist for testing of pre- and postconditions and invariants of Java functions. They can be of use if, e.g., the effect function can only be evaluated under certain conditions. First of all, assertions can be used in Java using the `assert` key word. When an assert statement is reached, its argument, which is a boolean expression, is evaluated and should evaluate to true. If not, an exception is thrown. Using assertions, pre- and postconditions and invariants and their evaluation can be added to the source code. This only helps in checking the conditions at run-time. Apart from this adding of assertions, tools exist to test code by using test suites dedicated to testing the specific functions. An example of such a tool is JUnit [Obj05].

Communication Primitives

The active server push interaction, the sending of an e-mail, can be easily implemented using one of the available e-mail implementations in Java.

On the contrary, the implementation of the other communication primitives is not as straightforward. Main issue is that servlets (like all Web services) are request/re-

sponse based: The servlet is waiting for a request to come in after which it is processed and the corresponding (calculated) response is sent back. Since the specification is a sequential program, this sequence should be turned into an event-driven servlet implementation. This can be done by keeping track of the location and the state "the application is in" at the moment that a request is received. If this request is expected, the application continues by returning the response.

The event-driven implementation causes the servlet only to make progress if requests are sent. Therefore, several threads should be used. A thread is a single sequential flow of control within a program. A thread executing the sequential application is started and as soon as a request is expected, the thread waits for this request to take place. Since many requests can take place in parallel, synchronisation of the threads is important.

Implementation of the communication primitives is done by using their internal structure: the primitives consist of alternating requests and responses. By turning the sequence of interactions and local actions into a sequence of requests, responses and local actions, we are able to keep track of the state at a lower level.

As explained, a servlet uses the *service* method which has a (wrapped) request and response as its parameters, coupling the response to the request. The communication primitives we make use of can be implemented using this service method. A reactive server push, the simple providing of a plain Web page, can be implemented by a single service call. Sessions are implemented by a sequence of service calls. Internal actions contained in a session are executed by calling the function implementing the internal action from within the service method.

11.2.2 Operators

In this section we have a closer look at how the operators can be implemented.

A Java program specifies a sequentially executable process where the actions are separated using semicolons, so the sequential composition, putting actions in sequence, is implemented using the semicolon. For example, we implement $x \cdot y$ as follows:

```
x ; y
```

We already explained that conditional branching can be compared with an if-statement which is available in Java. So, e.g, $x \triangleleft b \triangleright y$ is implemented as follows:

```
if ( b ) {
  x
} else {
  y
}
```

In $x \triangleleft b \triangleright y$, evaluation of boolean expression b is not a step, in contrast to execution of the `if(b){x}else{y}` statement in Java. Since extra states (after evaluation of guard b but before execution of the first action of x or y) are added, this might influence possible deadlock situations.

The conditional repetition can be implemented using a while in Java. Thus, $b \triangleright\!\triangleright x$ is implemented by the following statement:

```
while ( b ) {
  x
}
```

Again, this implementation introduces extra states between evaluation of the guard and execution of the first action from the conditionally executed process.

Conditional Disrupts

To implement the conditional disrupt, we can make use of a separate thread, running it in parallel to the process which is conditionally executed. At the moment that the condition holds, the process is stopped.

Alternatively, using the `assert` key word, we can add assertions to the specification. This is explained in Section 11.2.1 on page 196. Advantage of using assertions is that we do not have to implement and execute separate threads. Disadvantage is that the statement has to be put in between all actions in the process which is conditionally executed. Therefore, thread implementation is preferred.

Scope Operator

The scope operator as introduced in Section 6.3 can be implemented in Java using curly braces. So, e.g., $[n \mapsto 0 : \mathbb{N} \mid n := n + 1 \cdot n := n \times 2]$ can be implemented as follows:

```
{ int n = 0; n := n + 1; n := n * 2; }
```

However, if the scope contains interactions or even multiple sessions it becomes more complex. Since the implementation is not a sequential program but an event-driven program, scoping must be implemented by maintaining a context for each session in which interaction with the application takes place. The variables in the context have different scopes, like application-scope and session-scope. In Java, these contexts are available. Java Server Pages for example also use these context objects for implementing scope.

Access Control

In Section 7.3 we introduced access control. We identified three types of access control, viz. anonymous interaction, identification and registration. The last two occur in two shapes, viz. using the bang operator and the generalised parallel composition operator.

Both users and groups can be implemented by objects of specific classes. The class specifying a group is extended with several methods for adding users, removing users, and checking if a user is a member.

The groups used as a basis for the identification and registration can be implemented by sets of objects representing the users. Depending on the information and identification method, a user has several properties, like a name, a password, an object representing a fingerprint, etcetera. The implementation uses the properties for the generation of forms for identification and registration.

For the implementation of anonymous sessions, no registration step needs to be added at all. As a result, the application does not know who the interacting client is.

Transactions

In Java, several techniques exist for the synchronisation of pieces of code with respect to variable access and updates. First of all, the `synchronized` key word can be used for synchronising execution of functions: only one instance of the function can be executed at a certain time. Apart from that, the `synchronized` statement can be used, which synchronises access to specific variables. Have a look at the following two pieces of Java code:

```
synchronized (a) {
 a := 1;
 a := a * 2;
}
```

```
synchronized (a) {
  a := 0;
  a := a + 2;
}
```

If we execute the code in parallel, all actions which access variable `a` are synchronised. So, the state after execution of both pieces of code in parallel results in variable *a* evaluating to 2. Using the `synchronized` key word results in a locking mechanism that can be compared with third degree isolation (see Table 8.7 on page 112): variables are also locked when read.

For implementing transactional behaviour in Java, an interface is available: the Java Transaction API (JTA) [SUN05b]. The Java Transaction Service (JTS) [Che99] is an implementation of a transaction manager that supports the Java Transaction API. A transaction manager can be used in Java programs for the implementation of transactions with lower than third degree of isolation.

11.3 Implementing the Compiler

To compile our specified application into a running Internet application we need a parser to parse the specification. To implement a parser we can choose e.g. for using the Java parser generator *Java Compiler Compiler (JavaCC)* [VS+03]. This choice is preferred because we are specifying an Internet application and Java is the Internet specification language par excellence. JavaCC is a parser generator that produces parsers in Java from grammar specifications written in a lex/yacc-like manner. More information on JavaCC can be found at [VS+03].

JavaCC is both a lexical analyser and a grammar parser. First, all terminals like key words, predefined object and special characters are defined. Furthermore one can define which characters to skip—end of line, spaces, tabs—and how comments can be inserted. After specifying the lexical analyser the grammar specification is defined. Each non-terminal is specified by a Java method.

Using JavaCC, we are able to build a compiler for compiling a *DiCons* specification into a piece of Java code specifying a servlet class. Of course, an ASCII-like version of the specification language is needed, such that the Java Compiler Compiler can parse it. This generated servlet class can subsequently be compiled into Java bytecode, which can be executed by a Web server.

11.4 Future Work

In [Bee00] we present the implementation of a compiler based on the first version of *DiCons*. The specifications that can be compiled using that compiler contain amongst other things the internal actions, specified as Java functions, the communication primitives as explained in Chapter 7 and deadlines, which can be compared with conditional disrupts having a condition of the form $deadline < now$. This compiler

```
WEB_FORM ::= '{' · title · ':' · SPECS · body · ':' · SPECS · '}'
E_MAIL   ::= '{' · from · ':' · SPECS · to · ':' · SPECS ·
             subject · ':' · SPECS · contents · ':' · SPECS · '}'
SPECS    ::= (SPEC · ';')+
SPEC     ::= text · ':' · TEXT
             | (input | textarea) · ':' · VARIABLE ·
               [default · (TEXT | VARIABLE)] ·
               [check · TEXT · else · TEXT]
             | output · ':' · (VARIABLE | URL)
             | (select | submit | radiobutton) · ':' ·
               VARIABLE · '=' · ('(' · TEXTLIST · ')' | VAR_NAME)
             | checklist · ':' · VARIABLE
TEXT     ::= '"' · (ANYCHAR)* · '"'
ANYCHAR  ::= {any character except '"'}
VARIABLE ::= VAR_NAME · ('.' · VARIABLE)*
VAR_NAME ::= IDENTIFIER
IDENTIFIER ::= CHAR · (CHAR | DIGIT | '_')*
CHAR     ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
DIGIT    ::= '0' | '1' | ... | '9'
TEXTLIST ::= TEXT · (',' · TEXT)+
```

Table 11.2: Syntax specification of the interaction primitives.

served as a proof of concept in the early stage of our research.

By implementing the compiler, we encountered several issues which had to be taken into account. In this section we discuss these issues in short and give suggestions on how they can be tackled.

11.4.1 Using XML

First of all, we use HTML to make up the documents. These hypertexts have useful properties like the possibility for easily specifying Web forms. However, by using HMTL, we restrict ourselves to one interface, viz. Web browsers, where other interfaces and devices like WAP on a mobile phone, could also be used for interacting. We conclude that the Extensible Markup Language (XML) [YBP+04] serves better for both specifying and implementing the Web pages and forms that are communicated.

In the experiment in [Bee00] we used an ad hoc syntax for specifying presentations of interactions of which the BNF notation is given in Table 11.2.

This BNF specification shows that we can define two different interactions, viz. Web forms and e-mails. A Web form consists of a title and a body where an e-mail consists of a sender (from), a receiver (to), a subject and a body. All these elements themselves are specified using pieces of plain text, input fields and values of (output) variables. Depending on the part and type of an interaction, certain specifications are not allowed. E.g. one cannot add an input field to the subject field of an e-mail. For defining these constraints we have a set of static rules which must be answered by the specification.

In [Bee00, BBM01c] we gave some specifications of *DiCons* applications. To give an idea of what such an interaction looks like we give a small example. Note that the BNF given in Table 11.2 only specifies the part on representation of the interaction.

```
session of Initiator → add_voter(out I: Initiator, in v: Voter, in s: String) =
 {title:
    text: "Internet Vote";
   body:
    text: "Hello";
    output: I.name;
    text: "Insert voter:";
    text: "name: ";
    input: v.name
      check "/\S/"
      else "Fill out a name, please";
    text: "email: ";
    input: v.email
      check "/^\w+((-\w+)|(\.\w+))*\@\w+((\.|-)\w+)*\.\w+$/"
      else "Incorrect email address.";
    text: "Add more voters?";
    submit: s from ("yes", "no");
 };
```

As can be seen from the example and the BNF specification the experiment was feasible for a restricted setting for the communication, viz. Web forms and e-mails only. However, it is preferred to specify interactions in a more general and protocol independent way. So we do not want to make a separation between e-mails and Web forms. Depending on the device or tool used for interacting with a *DiCons* application, the corresponding representation should be used.

Furthermore, restricting to predefined input types like 'input', 'select' and 'checkbox' is not preferred. The syntax of interaction specifications should be outside the *DiCons* syntax to make it easier to adapt *DiCons* to future extensions on tools and techniques. To achieve this, a choice for using XML [YBP+04] can be made. XML

makes it possible to write (human-readable) code which can both serve as specification language for the interactions and output format for a *DiCons* program. We give a short introduction to XML, mostly based on the technical introduction to XML by Walsh [Wal98].

XML [YBP+04], developed by the World Wide Web Consortium (W3C), is a subset of the Standard Generalized Markup Language (SGML), an international standard used for defining the rules to write markup languages. One reason to prefer XML over, e.g., HTML is to be able to separate the data from its representation. This led to three components of XML:

1. The content;
2. The specification of the elements, the structure;
3. The specification of the visual aspects, the representation.

The content of an XML document contains one or more elements. These elements have a type and possibly some attributes and a content. An element is written down using HTML-like tags:

<element_type_name attribute_name="attribute_value">
  element's content
</element_type>

By giving the structure of an XML document, the content is bound to a specific set of elements. This structure can be given as a *Document Type Definition (DTD)* [YBP+04], which can be compared with a BNF specification. A DTD contains the definition rules of element tags. It is used to denote the elements, their attributes and the order in which elements appear in an XML document. DTDs themselves are not extensible and not written in XML. Furthermore, they do not specify data types which can be very useful for a lot of applications. To add these properties to DTDs, XML Schemas [FW04, TBMM04, BM04] are introduced.

The representation of an XML document is specified using style sheets. These style sheets map an XML element to its representation. This representation does not necessarily have to be a Web-based representation. One can use several style sheets to transform one and the same XML document into different formats like HTML, WML but also PDF. Style sheets can be defined using the Extensible Style sheet Language (XSL), which itself is divided into three parts:

1. A language for referencing specific parts of an XML document (XML Path Language – XPath);

2. A language for transforming XML documents into other XML documents (XSL Transformations – XSLT);
3. An XML vocabulary for specifying formatting semantics.

In general, XSL Transformations are used for transforming XML into e.g. HTML for generating Web pages or WML for interaction via mobile devices using WAP. Depending on the device accessing the XML document a corresponding transformation takes place.

For *DiCons* applications it is necessary to transform the XML specifications into interactive representations, like Web forms. By parameterising the *DiCons* compiler with a Document Type Definition, we can use one and the same compiler to generate different interaction sets.

Another advantage of using XML is that making use of multiple style sheets leads to different representations of one and the same interaction. This leads to using XML at the representation side of the application, i.e. after compilation of the specification.

By using XML at the specification side of the application, this XML data can be compiled into a Document Object Model (DOM) [LLW+04] which can be used at run-time to produce XML as output format. By looking at the syntax of the former *DiCons* versions it can be easily seen that this was based on interaction using Web forms and e-mails only. We specified the interactions using their commonly used and well known naming schemes, viz. title and body for Web forms and from, to, subject and contents for e-mails. However, analysing the two representations shows that e-mails and Web forms actually contain more or less the same elements. The subject and contents of an e-mail can be compared with the title and body of a Web form, respectively. The 'from' field of an e-mail usually contains the initiator of an application or a virtual sender having the application's name. The sender of a Web form can also be seen as the initiator of the application or as the application itself. So this 'from' field can be omitted without losing information.

An e-mail's 'to' field can be left out as well since the receiver is already known: the interacting party should receive the message. However, by omitting this field we make it harder to generate the XML documents used for interactions. That is because we want to be able to use style sheets independent of the *DiCons* syntax. If we omit the 'to' field we lose the interactor's information since we drop the possibility for adding a receiver to the XML document.

This observation leads to a different way of giving interactions. We generalise both specifications of Web forms and e-mails to one basic interaction specification. By doing this we can drop the technique-specific naming of different parts of interactions. So we use one general naming scheme for both e-mails and Web forms. An overview is shown in Table 11.3. By generalising these presentation specifications we cannot

e-mail	Web forms	general
from	(application/initiator)	sender
to	(interactor)	receiver
subject	title	subject
contents	body	message

Table 11.3: General naming for e-mails and Web forms.

make a distinction between interactions that are allowed to have input fields or not, i.e. we are not able to check the static demands on distinct types of interactions. So preferably, we extend the XML part over the complete interaction specification, including their types. We can do this by giving a Document Type Definition based on an interaction instead of only its components.

Another observation we make is that output variables which could be included using the 'output' key word actually are references to parameters of the interactions. They are replaced by their values when interacting with a user at run-time. To correspond with this, we introduce an XML 'output' element which can occur as child element of both a subject and a message.

Furthermore, we made use of the 'input' key word which was used for giving specifications of Web form elements. These elements returned a value and, depending on its type, could produce error messages. To turn this into an XML specification, we introduce another element named 'input'. This element can have several child elements, depending on its type.

Note that the new *DiCons* compiler will be independent of the elements given above. We will parameterise the compiler with a DTD which is used for parsing the interaction specifications. As a result of the former observation we can give a straightforward Document Type Definition.

A *DiCons* interaction has a name, a direction (*push* or *pull*) and an activity (*active* or *reactive*). These are chosen to be attributes of the interaction since they are always fixed. The contents of an interaction consists of four parts, viz. sender, receiver, title and message. These elements themselves can contain several elements depending on the type of the interaction. The base components of an interaction can be distributed over two classes:

Plain text components can contain plain texts, which, depending on the component it takes part in, represent a value. So, e.g. the text value

"somebody@somedomain.com"

specifies an e-mail address if it is placed in a sender or receiver element.

```
<!ELEMENT dicons-interaction (sender, receiver, title, message)>
<!ATTLIST dicons-interaction direction (push|pull) #REQUIRED
                             activity (active|reactive|session) #REQUIRED>
<!ELEMENT sender (user)>
<!ELEMENT receiver (user)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT message (#PCDATA|input)+>
<!ELEMENT user (name, e-mail-address, telephonenumber, password)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT e-mail-address (#PCDATA)>
<!ELEMENT telephonenumber (#PCDATA)>
<!ELEMENT password (#PCDATA)>
<!ELEMENT input (EMPTY)>
<!ATTLIST input result CDATA #REQUIRED
                type (textfield|checkbox|radiobutton|submit)
                check CDATA
                error CDATA>
```

Table 11.4: Document Type Definition of *DiCons* interactions.

Input elements Within the contents of interactions we can place input elements which lead to the interaction becoming interactive. This means that users who receive such an interaction can react on it by "filling in" the input elements. Note that this can only take place if the direction of the interaction is *pull*.

By using the DTD given in Table 11.4 we can add input fields to an interaction whose direction is a push. This is not desired. To solve this problem, we can abstract from the `direction` attribute by introducing two elements `dicons-interaction-push` and `dicons-interaction-pull`. However, in that case we can still define an active pull interaction which is not allowed. To solve this we have to abstract from the `activity` attribute as well. This leads to five elements, all defining one of the communication primitives explained in Chapter 7. As a result of this observation it would be very suitable if Document Type Definitions can be extended with a set of rules to which an XML specification must answer.

In [Bee00, BBM01c] we gave some specifications of *DiCons* applications. An example of the notation we made use of together with the XML notation is given in Table 11.5.

```
session of Initiator → add_voter(
    out I: Initiator,
    in v: Voter, in s: String) =
{title:
   text: "Internet Vote";
 body:
   text: "Hello";
   output: I.name;
   text: "Insert voter:";
   text: "name: ";
   input: v.name
     check "/^\S/"
     else "Fill in a name, please";
   text: "email: ";
   input: v.email
     check "/^\w+((-\w+)|(\.\w+))*\
          \@\w+((\.|-)\w+)*\.\w+$/"
     else "Wrong email address";
   text: "Add more voters?";
   submit: s from ("yes", "no");
};
```

```
session of Initiator → add_voter(
    out I: Initiator,
    in v: Voter, in s: String) =
<title>Internet Vote</title>
<message>
  Hello <output>this.name</output>,
  Insert voter:
  name:
  <input type="textfield">
    <result>v.name</result>
    <check>/\S/</check>
    <error>Fill in a name, please</error>
  </input>
  email:
  <input type="textfield">
    <result>v.email</result>
    <check>/^\w+((-\w+)|(\.\w+))*\
        \@\w+((\.|-)\w+)*\.\w+$/</check>
    <error>Wrong email address</error>
  </input>
  Add more voters?
  <input type="submit">
    <result>s</result>
  </input>
</message>
```

Table 11.5: Example of an XML *DiCons* interaction.

11.4.2 Adding Scope Control

As mentioned before, scope can be implemented using context objects. In the compiler we have currently available, all variables must be declared at one place, in the beginning of the specification, which leads to having a fixed scope in which all variables are available. This should of course be extended to multiple scopes where, amongst other things, session-scope variables can be declared.

11.4.3 Adding Access Control

In the compiler currently available, no predefined users can be used for interacting with the application: all groups can only be initialised by the empty set. As a result, the adding of users to groups has to be specified explicitly. This is done by asking for a user's name and e-mail address, after which a password is generated for the specific user. Having the possibility to implement groups using e.g. a database table would be preferable. In that way it is much easier to initialise and maintain groups.

11.4.4 Adding Transactional Processes

The compiler does not implement transactional behaviour at all. As stated in Section 11.2.2, this can be added by implementing the transactional behaviour specified in Chapter 8 in Java by using implementations of the Java Transaction API (JTA) [SUN05b] which are publicly available, e.g. Sun's Java Transaction Service (JTS) [Che99].

11.5 Conclusions

Although many elements are not available in the (meanwhile outdated) compiler built during the feasibility study, we have shown that generating running applications from formal *DiCons* specifications is possible using a proof-of-concept compiler.

The choice for using HTML led to a more or less fixed representation of interactions, which is not very useful for real-life applications. As mentioned in Section 10.10, this problem with the representation of interactions is the inverse of the problem of matching HTTP-replies against abstract events when using the *DiCons* specifications for testing.

The specifications used as a basis for the compiler can be compared with the formal specifications in this thesis to a great extent. Main differences are the lack of transactional operators, the explicit inclusion of presentations of interactions (see e.g. Table 11.5), the use of roles instead of groups, and declarations of variables on a fixed place instead of using a scope operator.

Choosing for Java servlets as a target interface gives many possibilities, but of course also brings in some restrictions. The types used for variables and functions are restricted to the classes defined in Java. However, using specific application programming interfaces (APIs), we can extend this set of classes and implement our own data types which can be used for giving specifications.

IV

Conclusions

12

Related Work

Closest to our work is the development of the Web-language *Mawl* [ABBC99, LR95]. This is also a language that supports interaction between an application and a single user, and adds a state concept to HTML. Mawl provides the control flow of a single session, but does not provide control flow across several sessions (the only thing that persists across sessions are the values of global variables). This is a distinguishing feature of *DiCons*: interactions involving several users are supported. On the other hand, Mawl does allow several sessions with a single user to exist in parallel, using an atomicity concept to execute sequences of actions as a single action.

A descendant of the Mawl project is <bigwig> [BMS02]. It inherits the concepts of sessions and document templates. However, <bigwig> is no longer under development and passed into JWIG [CMS03], its Java-based successor. As in our research, the main goal of the JWIG project is to simplify development of complex Internet applications. Like *DiCons*, JWIG is based on sessions, where a session consists of a sequence of interactions between a server and a client. In *DiCons* we have the possibility to interconnect sessions. JWIG allows the sharing of data between sessions, but it does not provide a mechanism for the sequential composition of them. Transactional behaviour is available using either Java's built-in serialisation mechanism or serialisation based on XML representations. JWIG provides a mechanism for sending e-mails, which can be compared with the active server push interactions in *DiCons*.

A so-called embedded domain specific language (EDSL) for programming Internet applications is *WASH* [Thi02] developed by Thiemann. Actually, it is a family of languages which are all embedded in the functional language Haskell. The language that comes closest to *DiCons* is WASH/CGI, which is a language for programming server-side applications with sessions and forms. This can be compared to *DiCons* sessions and forms to a great extent. E-mail interaction is not included in WASH/

CGI.

Groupware is a technology designed to facilitate the work of groups. This technology may be used to communicate, co-operate, co-ordinate, solve problems, compete, or negotiate. Groupware can be divided into two main classes: asynchronous and synchronous groupware. Synchronous groupware concerns an exchange of information, which is transmitted and presented to the users instantaneously by using computers. An example of synchronous groupware is chatting via the Internet. On the other hand, asynchronous groupware is based on sending messages which do not have to be read and replied to immediately. Examples of asynchronous groupware that can be specified in *DiCons* are work-flow systems to route documents through an office and group calendars for scheduling projects. More information on groupware can be found in [Ude99].

In the early days of the Internet, *Visual Obliq* [KB94] was developed, which is an environment for designing, programming and running distributed, multi-user GUI applications. Its interface builder outputs code in an interpreted language called *Obliq* [Car94]. Unlike *DiCons* applications, Obliq applications do not have to run on one single server: an application can be distributed over several so-called sites. After setting up a connection, sites can communicate directly. In this way, an application can be partitioned over different servers. Another difference with respect to *DiCons* is that a client has to install a special interpreter to view Visual Obliq applications whereas *DiCons* makes use of standard client-side techniques like HTML pages which can be viewed using a Web browser. In [BC95], embedding distributed applications in a hypermedia setting is discussed and in particular how applications generated in the Visual Obliq programming environment are integrated with the World Wide Web. Here, a Web browser is used to refer to a Visual Obliq application, but it must still be viewed using an interpreter.

Collaborative Objects Coordination Architecture (COCA) [LM98] is a generic framework for developing collaborative systems. In COCA, participants are divided into different roles, having different rights like in *DiCons*. Li, Wang and Muntz [LWM98] used this tool to build an online auction. A COCA Virtual Machine runs at each client site to control the interactions between the different clients. On the other hand, any client connected to the Internet can communicate with a *DiCons* application without having to reconfigure his machine.

The *Describing Collaborative Work Programming Language (DCWPL)* [CM96] helps programmers to develop customisable groupware applications. DCWPL does not concern the computational part of an application. As in *DiCons*, this part is specified in a computational language like Java, Pascal or C++. A DCWPL application also runs on an interpreter, here called control engine. DCWPL is based on synchronous groupware in contrast to *DiCons* in which the asynchronous aspect is more important.

The World Wide Web Consortium defined so-called Web Services. Following their definition [BHM+04], "a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards." To give operational semantics of Web Services, BPEL [ACD+03] and WS-CDL [KBR+04] are available. To describe the computational logic of a single client, the Business Process Execution Language for Web Services (BPEL) is available. BPEL is also identified as BPELWS or BPEL4WS. It combines and replaces IBM's WebServices Flow Language (WSFL) [Ley01] and Microsoft's XLANG specification [Tha01]. Using BPEL, which is an XML-based language, control-flow (including conditions, sequential composition, parallelism and loops), observable behaviour, variables and functions, event handlers, time-out management and exception handlers can be defined. On the other hand, the Web Services Choreography Description Language (WS-CDL) is used for the description of collaboration between multiple parties. It is based on a variant of π-calculus [MPW92], viz. the Explicit Solos calculus [GLW02]. WS-CDL specifications focus on the global description of Web Services in a more general way than *DiCons* does. Where we focus on distributed consensus problems, WS-CDL can be used for the specification of any multi-party distributed Web Service. The communication takes place between parties directly, instead of with a centralised *DiCons* application. Furthermore, where we focus on communication with real users using a Web browser and e-mail to communicate, Web Services focus on communication with clients in general. Communication takes place by both the receiving and sending of XML documents using SOAP [Mit03, GHM+03a, GHM+03b, HHK+03] in contrast to simple HTTP requests and HTTP responses extended with input parameters.

Further, there are languages that allow to program browsing behaviour. These, for instance, allow to program the behaviour of a user who wants to download a file from one of several mirror sites. For so-called *Service Combinators* see [CD99, KM98]. A further development is the so-called *ShopBot*, see [DEW97].

Our implementation is based on existing Internet programming techniques, viz. Java servlets and HTML. In Udell's book on groupware [Ude99] an Internet vote is implemented using a Java servlet. Also in [O'B99] an election servlet is presented. Furthermore, there are many commercial voting servlets put on the market. One of them can be found at [Col02]. To set up an Internet auction one can use commercial software like the *Auction Engine* [Sit05] developed by SiteOption.

Other useful Internet programming techniques are Java Server Pages (JSP) [SUN05c], Active Server Pages (ASP) [Mic05], ColdFusion [Mac05] and PHP Hypertext Preprocessor [PHP05]. We can extend these techniques with customised tags for distributed

consensus. However, these techniques are library-based and therefore not as suitable for formal verification as our language-based *DiCons* technique.

13

Conclusions

Formal methods in combination with Internet applications are not a well-known combination. Most Internet applications are ad hoc implementations where the code and logic is spread over several files of several types. Therefore, it is hard to use formal methods in this field of computer science. In this thesis we tried to make this link by formalising the Internet communication process using process algebra.

We succeeded in defining a formalism which is suitable for specifying Internet applications for distributed consensus, i.e., applications which help several users to come to consensus without having to meet physically.

The first thing to conclude is that nowadays most Internet applications serve more than only the goal of reaching consensus: most applications implement complex behaviour of which some parts can be specified as an actual distributed consensus protocol. Therefore, only those parts of the Internet application can be specified using *DiCons*. Although this is a disadvantage, it is still very useful to have a mechanism for proving aspects of the application, for testing it or even for generating it from its formal specification.

Since transition systems grow very large as a result of the parallel composition of sessions, proving properties of applications can, and should be, supported using computers instead of doing it by hand. This is a conclusion that can be drawn for almost all process algebraic specification tools and techniques [FGR04].

When using *DiCons* specifications as a basis for testing, the errors found in Internet applications highly depend on the chosen specification: applications conforming to specifications do not necessarily have to be correct. We gave examples of errors that could not be detected as a result of the chosen specification. As a result, if we want to give useful specifications, we first need to determine the errors we want to be able to

find. Next, we need to give a specification which is suitable for testing for the errors.

In general, when implementing (and thus actually specifying) Internet applications, one reasons from the point of view of the client interacting with it. This is caused by the fact that requests are one-on-one connected to responses. However, by reasoning from the point of view of the application, i.e., by interconnecting the responses to the subsequent requests, interactions get functional behaviour: by specifying the interactions using the communication primitives as specified in this thesis, they can be looked at as being functions with input and output parameters.

Formalising transactions in combination with states leads to complex overhead which is caused by the explicit locking and unlocking of actions. The semantics of transactional processes highly depends on the determination of valuations that might be updated by actions. In theory, this is no problem, however in practice, implementation of a function for determining the possibly updated valuations is very hard, if not, impossible. Most theories make use of explicit synchronisation of shared variables. We chose to use implicit synchronisation in *DiCons*, using first degree isolated transactions. Adding an operator to explicitly define the variables in the state that might be updated by an action or a transaction is possible. However, since this method leads to a lot of specification work, we have chosen to leave this out of the specification and use implicit synchronisation using the transactional operator which depends on the U function for determining the possibly updated valuations.

Another major issue when using formal specifications is the matching of hypertext documents to their abstract representations and the other way around, creating hypertext documents from abstract representations. The first method is needed when testing applications: the response received from an application should be matched to a abstract communication action representing the response. The second method is used when constructing hypertext documents by an application generated from a specification.

A final conclusion we draw is that using *DiCons* helps in getting a better understanding of what Internet applications look like, how they (should) behave, and that specifying and implementing Internet applications is not as easy and straightforward as it looks like.

V

Appendices

A

Overview of PAtrans

In this appendix we give an overview of the axioms and deduction rules for PAtrans, the process algebra with transactions. More detailed information can be found in Chapter 8. First of all, we give the constants:

- $\mathbb{A}$, the action alphabet, contains the set of atomic actions;
- δ, deadlock, representing unsuccessful termination;
- ε, the empty process, representing successful termination;
- $\mathbb{UL} = \{\mathcal{U}_A \mid \mathcal{U} \in \{\mathcal{C}, \mathcal{R}\}, A \subseteq \mathbb{A}\}$, the set of unlock actions;
- $\mathbb{L} = \{a_n \mid a \in \mathbb{A}, n \in \mathbb{N}\}$, the set of lockable actions.

For x and y processes in PAtrans, $a \in \mathbb{A}$ and $A \subseteq \mathbb{A}$, the operators of PAtrans are:

- $x + y$, alternative composition;
- $x \cdot y$, sequential composition;
- $x \parallel y$ and $x \lfloor\!\lfloor y$, the parallel composition operators;
- $[x]_a$, the locking operator;
- $\lfloor x \rfloor_A$, the unlocking operator;
- $\langle\langle x \rangle\rangle$ and $\langle\langle x, A, y \rangle\rangle$, the transactional operators.

The axiomatic semantics of PAtrans is given by the axioms in Table A.1:

$$\text{A1–9} + \text{TR1–14} + \text{L1–7} + \text{UL1–7} + \text{M1–11}$$

The operational semantics of PAtrans is given by T(PAtrans), the term deduction system induced by deduction rules 1–36 as shown in Table A.2. In both tables, **a** ranges over $\mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$ and a over $\mathbb{A}$.

A.1 Axioms of PAtrans

$x+y$	$=$	$y+x$	A1	$x \parallel y$	$=$	$x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x$		M1
$(x+y)+z$	$=$	$x+(y+z)$	A2	$\delta \mathbin{\lfloor\!\lfloor} x$	$=$	δ		M2
$x+x$	$=$	x	A3	$\varepsilon \mathbin{\lfloor\!\lfloor} \delta$	$=$	δ		M3
$(x+y)\cdot z$	$=$	$x\cdot z + y\cdot z$	A4	$\varepsilon \mathbin{\lfloor\!\lfloor} \varepsilon$	$=$	ε		M4
$(x\cdot y)\cdot z$	$=$	$x\cdot(y\cdot z)$	A5	$\varepsilon \mathbin{\lfloor\!\lfloor} \mathbf{a}x$	$=$	δ		M5
				$\varepsilon \mathbin{\lfloor\!\lfloor} (x+y)$	$=$	$\varepsilon \mathbin{\lfloor\!\lfloor} x + \varepsilon \mathbin{\lfloor\!\lfloor} y$		M6
$x+\delta$	$=$	x	A6	$\mathcal{U}_A x \mathbin{\lfloor\!\lfloor} y$	$=$	$\mathcal{U}_A(x \parallel \lfloor y \rfloor_A)$		M7
$\delta\cdot x$	$=$	δ	A7	$a_0 x \mathbin{\lfloor\!\lfloor} y$	$=$	$a_0(x \parallel [y]_a)$		M8
				$a_n x \mathbin{\lfloor\!\lfloor} y$	$=$	δ	if $n>0$	M9
$x\cdot\varepsilon$	$=$	x	A8	$ax \mathbin{\lfloor\!\lfloor} y$	$=$	$a(x \parallel y)$		M10
$\varepsilon\cdot x$	$=$	x	A9	$(x+y) \mathbin{\lfloor\!\lfloor} z$	$=$	$x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$		M11

$[\delta]_b$	$=$	δ		L1
$[\varepsilon]_b$	$=$	ε		L2
$[\mathcal{U}_B x]_b$	$=$	$\mathcal{U}_B \cdot [x]_b$		L3
$[a_n x]_b$	$=$	$a_{n+1}\cdot[x]_b$	if $a=b$	L4
$[a_n x]_b$	$=$	$a_n\cdot[x]_b$	if $a\neq b$	L5
$[ax]_b$	$=$	$a\cdot[x]_b$		L6
$[x+y]_b$	$=$	$[x]_b+[y]_b$		L7
$\lfloor\delta\rfloor_A$	$=$	δ		UL1
$\lfloor\varepsilon\rfloor_A$	$=$	ε		UL2
$\lfloor\mathcal{U}_B x\rfloor_A$	$=$	$\mathcal{U}_B\cdot\lfloor x\rfloor_A$		UL3
$\lfloor a_n x\rfloor_A$	$=$	$a_{n-1}\cdot\lfloor x\rfloor_A$	if $a\in A\wedge n>0$	UL4
$\lfloor a_n x\rfloor_A$	$=$	$a_n\cdot\lfloor x\rfloor_A$	if $a\notin A\vee n=0$	UL5
$\lfloor ax\rfloor_A$	$=$	$a\cdot\lfloor x\rfloor_A$		UL6
$\lfloor x+y\rfloor_A$	$=$	$\lfloor x\rfloor_A+\lfloor y\rfloor_A$		UL7
$\langle\langle x\rangle\rangle$	$=$	$\langle\langle x,\emptyset,x\rangle\rangle$		TR1
$\langle\langle x,\emptyset,\delta\rangle\rangle$	$=$	δ		TR2
$\langle\langle x,A,\delta\rangle\rangle$	$=$	$\mathcal{R}_A\cdot\langle\langle x\rangle\rangle$	if $A\neq\emptyset$	TR3
$\langle\langle x,\emptyset,\varepsilon\rangle\rangle$	$=$	$\mathcal{C}_\emptyset$		TR4
$\langle\langle x,A,\varepsilon\rangle\rangle$	$=$	$\mathcal{C}_A+\mathcal{R}_A\cdot\langle\langle x\rangle\rangle$	if $A\neq\emptyset$	TR5
$\langle\langle x,\emptyset,\mathcal{U}_B y\rangle\rangle$	$=$	$\mathcal{U}_\emptyset\cdot\langle\langle x,\emptyset,y\rangle\rangle$		TR6
$\langle\langle x,A,\mathcal{U}_B y\rangle\rangle$	$=$	$\mathcal{U}_\emptyset\cdot\langle\langle x,A,y\rangle\rangle+\mathcal{R}_A\cdot\langle\langle x\rangle\rangle$	if $A\neq\emptyset$	TR7
$\langle\langle x,\emptyset,a_n y\rangle\rangle$	$=$	$a_n\cdot\langle\langle x,\{a\},y\rangle\rangle$		TR8
$\langle\langle x,A,a_n y\rangle\rangle$	$=$	$a_n\cdot\langle\langle x,A\cup\{a\},y\rangle\rangle+\mathcal{R}_A\cdot\langle\langle x\rangle\rangle$	if $a\notin A\wedge A\neq\emptyset$	TR9
$\langle\langle x,A,a_n y\rangle\rangle$	$=$	$a\cdot\langle\langle x,A,y\rangle\rangle+\mathcal{R}_A\cdot\langle\langle x\rangle\rangle$	if $a\in A$	TR10
$\langle\langle x,\emptyset,ay\rangle\rangle$	$=$	$a_0\cdot\langle\langle x,\{a\},y\rangle\rangle$		TR11
$\langle\langle x,A,ay\rangle\rangle$	$=$	$a_0\cdot\langle\langle x,A\cup\{a\},y\rangle\rangle+\mathcal{R}_A\cdot\langle\langle x\rangle\rangle$	if $a\notin A\wedge A\neq\emptyset$	TR12
$\langle\langle x,A,ay\rangle\rangle$	$=$	$a\cdot\langle\langle x,A,y\rangle\rangle+\mathcal{R}_A\cdot\langle\langle x\rangle\rangle$	if $a\in A$	TR13
$\langle\langle x,A,y+z\rangle\rangle$	$=$	$\langle\langle x,A,y\rangle\rangle+\langle\langle x,A,z\rangle\rangle$		TR14

Table A.1: PAtrans: Process Algebra with Transactions.

A.2 Deduction Rules for T(PAtrans)

$$\frac{}{\varepsilon\downarrow}\,1 \qquad \frac{}{\mathbf{a}\xrightarrow{\mathbf{a}}\varepsilon}\,2 \qquad \frac{x\downarrow,\quad y\downarrow}{x\cdot y\downarrow}\,3 \qquad \frac{x\xrightarrow{\mathbf{a}}x'}{x\cdot y\xrightarrow{\mathbf{a}}x'\cdot y}\,4 \qquad \frac{x\downarrow,\quad y\xrightarrow{\mathbf{a}}y'}{x\cdot y\xrightarrow{\mathbf{a}}y'}\,5$$

$$\frac{x\downarrow}{x+y\downarrow,\quad y+x\downarrow}\,6 \qquad \frac{x\xrightarrow{\mathbf{a}}x'}{x+y\xrightarrow{\mathbf{a}}x',\quad y+x\xrightarrow{\mathbf{a}}x'}\,7 \qquad \frac{A\neq\emptyset}{\langle\langle x,A,y\rangle\rangle\xrightarrow{\mathcal{R}_A}\langle\langle x\rangle\rangle}\,8$$

$$\frac{x\downarrow}{\langle\langle x\rangle\rangle\xrightarrow{\mathcal{C}_\emptyset}\varepsilon}\,9 \qquad \frac{y\downarrow}{\langle\langle x,A,y\rangle\rangle\xrightarrow{\mathcal{C}_A}\varepsilon}\,10 \qquad \frac{x\xrightarrow{\mathcal{U}_B}x'}{\langle\langle x\rangle\rangle\xrightarrow{\mathcal{U}_\emptyset}\langle\langle x,\emptyset,x'\rangle\rangle}\,11 \qquad \frac{x\xrightarrow{a_n}x'}{\langle\langle x\rangle\rangle\xrightarrow{a_n}\langle\langle x,\{a\},x'\rangle\rangle}\,12$$

$$\frac{x\xrightarrow{a}x'}{\langle\langle x\rangle\rangle\xrightarrow{a_0}\langle\langle x,\{a\},x'\rangle\rangle}\,13 \qquad \frac{y\xrightarrow{\mathcal{U}_B}y'}{\langle\langle x,A,y\rangle\rangle\xrightarrow{\mathcal{U}_\emptyset}\langle\langle x,A,y'\rangle\rangle}\,14 \qquad \frac{y\xrightarrow{a_n}y',\quad a\notin A}{\langle\langle x,A,y\rangle\rangle\xrightarrow{a_n}\langle\langle x,A\cup\{a\},y'\rangle\rangle}\,15$$

$$\frac{y\xrightarrow{a_n}y',\quad a\in A}{\langle\langle x,A,y\rangle\rangle\xrightarrow{a}\langle\langle x,A,y'\rangle\rangle}\,16 \qquad \frac{y\xrightarrow{a}y',\quad a\notin A}{\langle\langle x,A,y\rangle\rangle\xrightarrow{a_0}\langle\langle x,A\cup\{a\},y'\rangle\rangle}\,17 \qquad \frac{y\xrightarrow{a}y',\quad a\in A}{\langle\langle x,A,y\rangle\rangle\xrightarrow{a}\langle\langle x,A,y'\rangle\rangle}\,18$$

$$\frac{x\downarrow}{[x]_b\downarrow}\,19 \qquad \frac{x\xrightarrow{\mathcal{U}_A}x'}{[x]_b\xrightarrow{\mathcal{U}_A}[x']_b}\,20 \qquad \frac{x\xrightarrow{a_n}x',\quad a\neq b}{[x]_b\xrightarrow{a_n}[x']_b}\,21 \qquad \frac{x\xrightarrow{a_n}x'}{[x]_a\xrightarrow{a_{n+1}}[x']_a}\,22$$

$$\frac{x\xrightarrow{a}x'}{[x]_b\xrightarrow{a}[x']_b}\,23 \qquad \frac{x\downarrow}{\lfloor x\rfloor_A\downarrow}\,24 \qquad \frac{x\xrightarrow{\mathcal{U}_B}x'}{\lfloor x\rfloor_A\xrightarrow{\mathcal{U}_B}\lfloor x'\rfloor_A}\,25 \qquad \frac{x\xrightarrow{a_n}x',\quad (a\notin A\vee n=0)}{\lfloor x\rfloor_A\xrightarrow{a_n}\lfloor x'\rfloor_A}\,26$$

$$\frac{x\xrightarrow{a_n}x',\quad (a\in A\wedge n>0)}{\lfloor x\rfloor_A\xrightarrow{a_{n-1}}\lfloor x'\rfloor_A}\,27 \qquad \frac{x\xrightarrow{a}x'}{\lfloor x\rfloor_A\xrightarrow{a}\lfloor x'\rfloor_A}\,28 \qquad \frac{x\downarrow,\quad y\downarrow}{x\parallel y\downarrow}\,29$$

$$\frac{x\xrightarrow{\mathcal{U}_A}x'}{x\parallel y\xrightarrow{\mathcal{U}_A}x'\parallel\lfloor y\rfloor_A,\quad y\parallel x\xrightarrow{\mathcal{U}_A}\lfloor y\rfloor_A\parallel x'}\,30 \qquad \frac{x\xrightarrow{a_0}x'}{x\parallel y\xrightarrow{a_0}x'\parallel[y]_a,\quad y\parallel x\xrightarrow{a_0}[y]_a\parallel x'}\,31$$

$$\frac{x\xrightarrow{a}x'}{x\parallel y\xrightarrow{a}x'\parallel y,\quad y\parallel x\xrightarrow{a}y\parallel x'}\,32 \qquad \frac{x\downarrow,\quad y\downarrow}{x\lfloor\!\lfloor y\downarrow}\,33 \qquad \frac{x\xrightarrow{\mathcal{U}_A}x'}{x\lfloor\!\lfloor y\xrightarrow{\mathcal{U}_A}x'\parallel\lfloor y\rfloor_A}\,34$$

$$\frac{x\xrightarrow{a_0}x'}{x\lfloor\!\lfloor y\xrightarrow{a_0}x'\parallel[y]_a}\,35 \qquad \frac{x\xrightarrow{a}x'}{x\lfloor\!\lfloor y\xrightarrow{a}x'\parallel y}\,36$$

Table A.2: Deduction rules for $T(PAtrans)$.

B

Proofs for PAtrans

In this appendix we prove soundness of the Process Algebra with Transactions, PAtrans, and we prove that all terms in this process algebra can be eliminated to a term in a $\mathrm{BPA}_{\delta\varepsilon}$ with guarded linear recursion ($\mathrm{BPA}_{\delta\varepsilon}$rec).

B.1 Soundness of PAtrans

Theorem (Soundness of PAtrans) The set of closed PAtrans terms modulo bisimulation equivalence, $T(\mathrm{PAtrans})/\leftrightarrows$, is a model for PAtrans.

Proof We prove this theorem by proving that each axiom is sound, i.e., we prove that for all closed instantiations of the axiom, both sides of the axiom correspond to the same element of the bisimulation model. This proof outline is taken from [Ver97, BV95]. For each axiom, as given in Table A.1, we take the relation which relates each process to itself (identity) and which relates the left-hand side of the equation to its right-hand side. So e.g. for proving axiom A1, we take relation

$$R = \{(x, x), (x + y, y + x) \mid x, y \text{ closed PAtrans terms}\} \ .$$

Furthermore, x and y are closed PAtrans terms. We use subscript notation to indicate the deduction rules we make use of. The deduction rules can be found in Table A.2.

Axiom A1 $x + y \leftrightarrows y + x$.

- Suppose $x + y \downarrow$, then$_6$, $x \downarrow$ or $y \downarrow$, but then also$_6$ $y + x \downarrow$.
- Suppose $x + y \xrightarrow{\mathbf{a}} x'$, then$_7$ $x \xrightarrow{\mathbf{a}} x'$ or $y \xrightarrow{\mathbf{a}} x'$, but then also$_7$ $y + x \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.

Using symmetry in x and y this proves $x + y \leftrightarrows y + x$.

Axiom A2 $(x + y) + z \Leftrightarrow x + (y + z)$.

- Suppose $(x + y) + z \downarrow$, then$_6$, $x + y \downarrow$ or $z \downarrow$.
  - If $x + y \downarrow$, then$_6$ $x \downarrow$ or $y \downarrow$.
    - If $x \downarrow$, then$_6$ $x + (y + z) \downarrow$.
    - If $y \downarrow$, then$_6$ $y + z \downarrow$ and thus$_6$ $x + (y + z) \downarrow$.
  - If $z \downarrow$, then$_6$ $y + z \downarrow$ and thus$_6$ $x + (y + z) \downarrow$.
- Suppose $(x + y) + z \xrightarrow{\mathbf{a}} x'$, then$_7$, $x + y \xrightarrow{\mathbf{a}} x'$ or $z \xrightarrow{\mathbf{a}} x'$.
  - if $x + y \xrightarrow{\mathbf{a}} x'$, then$_7$ $x \xrightarrow{\mathbf{a}} x'$ or $y \xrightarrow{\mathbf{a}} x'$.
    - If $x \xrightarrow{\mathbf{a}} x'$, then$_7$ $x + (y + z) \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.
    - If $y \xrightarrow{\mathbf{a}} x'$, then$_7$ $y + z \xrightarrow{\mathbf{a}} x'$ and thus$_7$ $x + (y + z) \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.
  - if $z \xrightarrow{\mathbf{a}} x'$, then$_7$ $y + z \xrightarrow{\mathbf{a}} x'$ and thus$_7$ $x + (y + z) \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.

Proof for the right-hand side is analogous.

Axiom A3 $x + x \Leftrightarrow x$.

Left-hand side:

- Suppose $x + x \downarrow$, then$_6$ $x \downarrow$.
- Suppose $x + x \xrightarrow{\mathbf{a}} x'$, then$_7$ $x \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.

Right-hand side:

- Suppose $x \downarrow$, then$_6$ $x + x \downarrow$.
- Suppose $x \xrightarrow{\mathbf{a}} x'$, then$_7$ $x + x \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.

Axiom A4 $(x + y) \cdot z \Leftrightarrow x \cdot z + y \cdot z$.

Left-hand side:

- Suppose $(x + y) \cdot z \downarrow$, then$_3$, $x + y \downarrow$ and $z \downarrow$, and thus$_6$ $x \downarrow$ and $z \downarrow$ or $y \downarrow$ and $z \downarrow$.
  - If $x \downarrow$ and $z \downarrow$, then$_3$ $x \cdot z \downarrow$ and thus$_6$ $x \cdot z + y \cdot z \downarrow$.
  - If $y \downarrow$ and $z \downarrow$, then$_3$ $y \cdot z \downarrow$ and thus$_6$ $x \cdot z + y \cdot z \downarrow$.
- Suppose $(x + y) \cdot z \xrightarrow{\mathbf{a}} x'$, then, either$_4$ $x + y \xrightarrow{\mathbf{a}} x''$ and $x' = x'' \cdot z$, or$_5$ $x + y \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$.
  - if $x + y \xrightarrow{\mathbf{a}} x''$ and $x' = x'' \cdot z$, then$_7$ $x \xrightarrow{\mathbf{a}} x''$ or $y \xrightarrow{\mathbf{a}} x''$.
    - if $x \xrightarrow{\mathbf{a}} x''$, then$_4$ $x \cdot z \xrightarrow{\mathbf{a}} x'' \cdot z$ $(= x')$ and thus$_7$ $x \cdot z + y \cdot z \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.
    - if $y \xrightarrow{\mathbf{a}} x''$, then$_4$ $y \cdot z \xrightarrow{\mathbf{a}} x'' \cdot z$ $(= x')$ and thus$_7$ $x \cdot z + y \cdot z \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.
  - if $x + y \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$, then$_6$ $x \downarrow$ or $y \downarrow$.
    - if $x \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$, then$_5$ $x \cdot z \xrightarrow{\mathbf{a}} x'$ and thus$_7$ $x \cdot z + y \cdot z \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.
    - if $y \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$, then$_5$ $y \cdot z \xrightarrow{\mathbf{a}} x'$ and thus$_7$ $x \cdot z + y \cdot z \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.

Right-hand side:

- Suppose $x \cdot z + y \cdot z \downarrow$, then$_6$, $x \cdot z \downarrow$ or $y \cdot z \downarrow$.
  - If $x \cdot z \downarrow$ then$_3$ $x \downarrow$ and $z \downarrow$. Then$_6$ $x + y \downarrow$ and thus$_3$ $(x + y) \cdot z \downarrow$.
  - If $y \cdot z \downarrow$ then$_3$ $y \downarrow$ and $z \downarrow$. Then$_6$ $x + y \downarrow$ and thus$_3$ $(x + y) \cdot z \downarrow$.
- Suppose $x \cdot z + y \cdot z \xrightarrow{\mathbf{a}} x'$, then$_7$ $x \cdot z \xrightarrow{\mathbf{a}} x'$ or $y \cdot z \xrightarrow{\mathbf{a}} x'$.
  - if $x \cdot z \xrightarrow{\mathbf{a}} x'$, then either$_4$ $x \xrightarrow{\mathbf{a}} x''$ and $x' = x'' \cdot z$, or $_5$ $x \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$.
    - if $x \xrightarrow{\mathbf{a}} x''$ and $x' = x'' \cdot z$, then$_7$ $(x + y) \xrightarrow{\mathbf{a}} x''$ and thus$_4$ $(x + y) \cdot z \xrightarrow{\mathbf{a}} x'' \cdot z$ $(= x')$ and $R(x', x')$.
    - if $x \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$, then$_6$ $(x + y) \downarrow$ and thus$_5$ $(x + y) \cdot z \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.
  - proof for $y \cdot z \xrightarrow{\mathbf{a}} x'$ is analogous.

Axiom A5 $(x \cdot y) \cdot z \leftrightarrows x \cdot (y \cdot z)$.

Left-hand side:

- Suppose $(x \cdot y) \cdot z \downarrow$, then$_3$ $x \cdot y \downarrow$ and $z \downarrow$, and thus$_3$ $x \downarrow$, $y \downarrow$ and $z \downarrow$. But then also$_3$ $y \cdot z \downarrow$ and thus$_3$ $x \cdot (y \cdot z) \downarrow$.
- Suppose $(x \cdot y) \cdot z \xrightarrow{\mathbf{a}} x'$, then either$_4$ $x \cdot y \xrightarrow{\mathbf{a}} x''$ and $x' = x'' \cdot z$, or$_5$ $x \cdot y \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$.
  - if $x \cdot y \xrightarrow{\mathbf{a}} x''$ and $x' = x'' \cdot z$, then either$_4$ $x \xrightarrow{\mathbf{a}} x'''$ and $x'' = x''' \cdot y$, or$_5$ $x \downarrow$ and $y \xrightarrow{\mathbf{a}} x''$.
    - if $x \xrightarrow{\mathbf{a}} x'''$ and $x'' = x''' \cdot y$, then$_4$ $x \cdot (y \cdot z) \xrightarrow{a} x''' \cdot (y \cdot z)$ and $x' = (x''' \cdot y) \cdot z$. $R((x''' \cdot y) \cdot z,\ x''' \cdot (y \cdot z))$.
    - if $x \downarrow$ and $y \xrightarrow{\mathbf{a}} x''$, then$_4$ $y \cdot z \xrightarrow{\mathbf{a}} x'' \cdot z$ $(= x')$ and$_5$ $x \cdot (y \cdot z) \xrightarrow{\mathbf{a}} x'$. $R(x', x')$.
  - if $x \cdot y \downarrow$ and $z \xrightarrow{\mathbf{a}} x'$, then$_3$ $x \downarrow$ and $y \downarrow$. So$_5$ $y \cdot z \xrightarrow{\mathbf{a}} x'$ and $_5$ $x \cdot (y \cdot z) \xrightarrow{\mathbf{a}} x'$ and $R(x', x')$.

Proof for the right-hand side is analogous.

Axiom A6 $x + \delta \leftrightarrows x$.

Left-hand side:

- Suppose $x + \delta \downarrow$, then$_6$ $x \downarrow$ or $\delta \downarrow$. Since $\delta \downarrow$ does not hold, $x \downarrow$.
- Suppose $x + \delta \xrightarrow{\mathbf{a}} x'$, then$_7$ $x \xrightarrow{\mathbf{a}} x'$ or $\delta \xrightarrow{\mathbf{a}} x'$. Again, since $\delta \xrightarrow{\mathbf{a}} x'$ does not hold, $x \xrightarrow{\mathbf{a}} x'$. $R(x', x')$.

Proof for the right-hand side is trivial using deduction rules 6 and 7.

Axiom A7 $\delta \cdot x \leftrightarrows \delta$.

Neither $\delta \cdot x$ nor δ can terminate or do a transition.

Axiom A8 $x \cdot \varepsilon \leftrightarrows x$.

Left-hand side:

- Suppose $x \cdot \varepsilon \downarrow$, then$_3$ $x \downarrow$ (and$_1$ $\varepsilon \downarrow$).
- Suppose $x \cdot \varepsilon \xrightarrow{\mathbf{a}} x'$, then either$_4$ $x \xrightarrow{\mathbf{a}} x''$ and $x' = x'' \cdot \varepsilon$ or$_5$ $x \downarrow$ and $\varepsilon \xrightarrow{\mathbf{a}} x'$, which is not possible. So $x \xrightarrow{\mathbf{a}} x''$ $x' = x'' \cdot \varepsilon$. $R(x'' \cdot \varepsilon, x'')$.

Right-hand side:

- Suppose $x \downarrow$. Since$_1$ $\varepsilon \downarrow$ it holds that$_3$ $x \cdot \varepsilon \downarrow$.
- Suppose $x \xrightarrow{\mathbf{a}} x'$, then$_4$ $x \cdot \varepsilon \xrightarrow{\mathbf{a}} x' \cdot \varepsilon$ and $R(x' \cdot \varepsilon, x')$.

Axiom A9 $\varepsilon \cdot x \leftrightarrows x$

Proof is similar to the proof of Axiom A8.

Axiom L1 $[\delta]_b \leftrightarrows \delta$.

Neither $[\delta]_b$ nor δ can terminate or do a transition.

Axiom L2 $[\varepsilon]_b \leftrightarrows \varepsilon$.

Both$_{19}$ $[\varepsilon]_b \downarrow$ and$_1$ $\varepsilon \downarrow$ and neither side can do a transition.

Axiom L3 $[\mathcal{U}_B \cdot x]_b \leftrightarrows \mathcal{U}_B \cdot [x]_b$.

Since$_2$ $\mathcal{U}_B \xrightarrow{\mathcal{U}_B} \varepsilon$ and$_1$ $\varepsilon \downarrow$, it holds that$_{4,5}$ $\mathcal{U}_B \cdot x \xrightarrow{\mathcal{U}_B} x$. So$_{20}$ $[\mathcal{U}_U \cdot x]_b \xrightarrow{\mathcal{U}_B} [x]_b$. Furthermore$_{2,1,4,5}$ $\mathcal{U}_B \cdot [x]_b \xrightarrow{\mathcal{U}_B} [x]_b$. $R([x]_b, [x]_b)$.

Axiom L4 $[a_n \cdot x]_b \leftrightarrows a_{nalt1} \cdot [x]_b$ if $a = b$.

Similar to the proof of Axiom L3, using rule 22 instead of rule 20.

Axiom L5 $[a_n \cdot x]_b \leftrightarrows a_n \cdot [x]_b$ if $a \neq b$.

Similar to the proof of Axiom L3, using rule 21 instead of rule 20.

Axiom L6 $[a \cdot x]_b \leftrightarrows a \cdot [x]_b$.

Similar to the proof of Axiom L3, using rule 23 instead of rule 20.

Axiom L7 $[x + y]_b \leftrightarrows [x]_b + [y]_b$.

Left-hand side:

- Suppose $[x + y]_b \downarrow$, then$_{19}$ $x + y \downarrow$ and thus$_6$ $x \downarrow$ or $y \downarrow$.
  - if $x \downarrow$, then$_{19}$ $[x]_b \downarrow$ and thus$_6$ $[x]_b + [y]_b \downarrow$.
  - if $y \downarrow$, then$_{19}$ $[y]_b \downarrow$ and thus$_6$ $[x]_b + [y]_b \downarrow$.

- Suppose $[x+y]_b \xrightarrow{\mathcal{U}_A} x'$, then$_{20}$ $x+y \xrightarrow{\mathcal{U}_A} x''$ and $x' = [x'']_b$. If $x+y \xrightarrow{\mathcal{U}_A} x''$, then$_7$ $x \xrightarrow{\mathcal{U}_A} x''$ or $y \xrightarrow{\mathcal{U}_A} x''$.
  - if $x \xrightarrow{\mathcal{U}_A} x''$, then$_{20}$ $[x]_b \xrightarrow{\mathcal{U}_A} [x'']_b$ $(= x')$ and thus$_6$ $[x]_b + [y]_b \xrightarrow{\mathcal{U}_A} x'$. $R(x', x')$.
  - if $y \xrightarrow{\mathcal{U}_A} x''$, then$_{20}$ $[y]_b \xrightarrow{\mathcal{U}_A} [x'']_b$ $(= x')$ and thus$_6$ $[x]_b + [y]_b \xrightarrow{\mathcal{U}_A} x'$. $R(x', x')$.
- Suppose $[x+y]_b \xrightarrow{a_n} x'$ for $a \neq b$. Similar to the previous case, using rule 21 instead of rule 20.
- Suppose $[x+y]_b \xrightarrow{a_n} x'$ for $a = b$. Similar to the previous case, using rule 22 instead of rule 21.
- Suppose $[x+y]_b \xrightarrow{a} x'$. Similar to the previous case, using rule 23 instead of rule 22.

Right-hand side:

- Suppose $[x]_b + [y]_b \downarrow$, then$_7$ $[x]_b \downarrow$ or $[y]_b \downarrow$.
  - if $[x]_b \downarrow$, then$_{19}$ $x \downarrow$ and thus$_6$ $x+y \downarrow$, so$_{19}$ $[x+y]_b \downarrow$.
  - if $[y]_b \downarrow$, then$_{19}$ $y \downarrow$ and thus$_6$ $x+y \downarrow$, so$_{19}$ $[x+y]_b \downarrow$.
- Suppose $[x]_b + [y]_b \xrightarrow{\mathcal{U}_A} x'$, then$_7$ $[x]_b \xrightarrow{\mathcal{U}_A} x'$ or $[y]_b \xrightarrow{\mathcal{U}_A} x'$.
  - if $[x]_b \xrightarrow{\mathcal{U}_A} x'$, then$_{20}$ $x \xrightarrow{\mathcal{U}_A} x''$ and $x' = [x'']_b$. If $x \xrightarrow{\mathcal{U}_A} x''$, then$_7$ $x+y \xrightarrow{\mathcal{U}_A} x''$ and$_{20}$ $[x+y]_b \xrightarrow{\mathcal{U}_A} [x'']_b$ $(= x')$. $R(x', x')$.
  - if $[y]_b \xrightarrow{\mathcal{U}_A} x'$, then$_{20}$ $y \xrightarrow{\mathcal{U}_A} x''$ and $x' = [x'']_b$. If $y \xrightarrow{\mathcal{U}_A} x''$, then$_7$ $x+y \xrightarrow{\mathcal{U}_A} x''$ and$_{20}$ $[x+y]_b \xrightarrow{\mathcal{U}_A} [x'']_b$ $(= x')$. $R(x', x')$.
- Suppose $[x]_b + [y]_b \xrightarrow{a_n} x'$ for $a \neq b$. Similar to the previous case, using rule 21 instead of rule 20.
- Suppose $[x]_b + [y]_b \xrightarrow{a_n} x'$ for $a = b$. Similar to the previous case, using rule 22 instead of rule 21.
- Suppose $[x]_b + [y]_b \xrightarrow{a} x'$. Similar to the previous case, using rule 23 instead of rule 22.

Axiom UL1–7

Similar to the proofs of Axioms L1–7.

Axiom M1 $x \parallel y \Leftrightarrow x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$.

We look at the transisiton of both sides of the axiom at the same time, making a case distinction on termination of x and any of the actions x can execute. Note that the axiom is symmetric in x and y.

- Suppose $x \downarrow$.
  - if $y \downarrow$, then$_{29}$ $x \parallel y \downarrow$, but also$_{33}$ $x \lfloor\!\lfloor y \downarrow$ and $y \lfloor\!\lfloor x \downarrow$ and thus$_6$ $x \lfloor\!\lfloor y + y \lfloor\!\lfloor x \downarrow$.
  - if $y \xrightarrow{\mathbf{a}} y'$, this case is handles in the rest of the proof for this axiom, using symmetry in x and y.

- Suppose $x \xrightarrow{\mathcal{U}_A} x'$, then$_{30}$ $x \parallel y \xrightarrow{\mathcal{U}_A} x' \parallel \lfloor y \rfloor_A$ and$_{34}$ $x \mathbin{\lfloor\!\lfloor} y \xrightarrow{\mathcal{U}_A} x' \parallel \lfloor y \rfloor_A$.
  Thus$_7$ $x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x \xrightarrow{\mathcal{U}_A} x' \parallel \lfloor y \rfloor_A$. $R(x' \parallel \lfloor y \rfloor_A, x' \parallel \lfloor y \rfloor_A)$.
- Suppose $x \xrightarrow{a_0} x'$, then$_{31}$ $x \parallel y \xrightarrow{a_0} x' \parallel [y]_a$ and$_{35}$ $x \mathbin{\lfloor\!\lfloor} y \xrightarrow{a_0} x' \parallel [y]_a$.
  Thus$_7$ $x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x \xrightarrow{a_0} x' \parallel [y]_a$. $R(x' \parallel [y]_a, x' \parallel [y]_a)$.
  Note that a transition $x \xrightarrow{a_n} x'$ for $n > 0$ is not possible.
- Suppose $x \xrightarrow{a} x'$, then$_{32}$ $x \parallel y \xrightarrow{a} x' \parallel y$ and$_{36}$ $x \mathbin{\lfloor\!\lfloor} y \xrightarrow{a} x' \parallel y$.
  Thus again$_7$ $x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x \xrightarrow{a_0} x' \parallel y$. $R(x' \parallel y, x' \parallel y)$.

Axiom M2 $\delta \mathbin{\lfloor\!\lfloor} x \leftrightarrows \delta$.

Neither side can terminate, nor do a transition.

Axiom M3 $\varepsilon \mathbin{\lfloor\!\lfloor} \delta \leftrightarrows \delta$.

Again, neither side can terminate, nor do a transition.

Axiom M4 $\varepsilon \mathbin{\lfloor\!\lfloor} \varepsilon \leftrightarrows \varepsilon$.

Both$_1$ $\varepsilon \downarrow$ and$_{33}$ $\varepsilon \mathbin{\lfloor\!\lfloor} \varepsilon \downarrow$. No other transitions are possible.

Axiom M5 $\varepsilon \mathbin{\lfloor\!\lfloor} \mathbf{a} \cdot x \leftrightarrows \delta$.

No rules can be applied to both sides, since $\mathbf{a} \cdot x \downarrow$ does not hold. So neither side terminates, nor can do a transition.

Axiom M6 $\varepsilon \mathbin{\lfloor\!\lfloor} (x + y) \leftrightarrows \varepsilon \mathbin{\lfloor\!\lfloor} x + \mathbin{\lfloor\!\lfloor} y$.

Left-hand side:

- Suppose $x + y \downarrow$, then$_6$ $x \downarrow$ or $y \downarrow$. Furthermore$_1$, $\varepsilon \downarrow$.
  - if $x \downarrow$, then$_{33}$ $\varepsilon \mathbin{\lfloor\!\lfloor} x \downarrow$ and thus$_6$ $\varepsilon \mathbin{\lfloor\!\lfloor} x + \varepsilon \mathbin{\lfloor\!\lfloor} y \downarrow$.
  - if $y \downarrow$, then$_{33}$ $\varepsilon \mathbin{\lfloor\!\lfloor} y \downarrow$ and thus$_6$ $\varepsilon \mathbin{\lfloor\!\lfloor} x + \varepsilon \mathbin{\lfloor\!\lfloor} y \downarrow$.
- Suppose $x + y \xrightarrow{\mathbf{a}} x'$, then none of the rules can be applied.

Right-hand side: Since$_1$ $\varepsilon \downarrow$, $\varepsilon \mathbin{\lfloor\!\lfloor} x + \varepsilon \mathbin{\lfloor\!\lfloor} y \downarrow$ if and only if$_6$ $\varepsilon \mathbin{\lfloor\!\lfloor} x \downarrow$ or $\varepsilon \mathbin{\lfloor\!\lfloor} y \downarrow$.

- $\varepsilon \mathbin{\lfloor\!\lfloor} x \downarrow$ if and only if$_{29}$ $x \downarrow$. But then also$_6$ $x + y \downarrow$ and thus$_{29}$ $\varepsilon \mathbin{\lfloor\!\lfloor} (x + y) \downarrow$.
- $\varepsilon \mathbin{\lfloor\!\lfloor} y \downarrow$ if and only if$_{29}$ $y \downarrow$. But then also$_6$ $x + y \downarrow$ and thus$_{29}$ $\varepsilon \mathbin{\lfloor\!\lfloor} (x + y) \downarrow$.

No transitions are possible.

Axiom M7 $\mathcal{U}_A \cdot x \mathbin{\lfloor\!\lfloor} y \leftrightarrows \mathcal{U}_A \cdot (x \parallel \lfloor y \rfloor_A)$.

Since$_2$ $\mathcal{U}_A \xrightarrow{\mathcal{U}_A} \varepsilon$ and$_1$ $\varepsilon \downarrow$, it holds that$_{4,5}$ $\mathcal{U}_A \cdot x \xrightarrow{\mathcal{U}_A} x$. So$_{34}$ $\mathcal{U}_A \cdot x \mathbin{\lfloor\!\lfloor} y \xrightarrow{\mathcal{U}_A} x \parallel \lfloor y \rfloor_A$.

Furthermore$_{2,1,4,5}$ $\mathcal{U}_A \cdot (x \parallel \lfloor y \rfloor_A) \xrightarrow{\mathcal{U}_A} x \parallel \lfloor y \rfloor_A$. $R(x \parallel \lfloor y \rfloor_A, x \parallel \lfloor y \rfloor_A)$.

Axiom M8 $a_0 \cdot x \lfloor\!\lfloor y \Leftrightarrow a_0 \cdot (x \parallel [y]_a)$.

Similar to the proof of Axiom M7, using deduction rule 35 instead of rule 34.

Axiom M9 $a_n \cdot x \lfloor\!\lfloor y \Leftrightarrow \delta$ if $n > 0$.

Neither side can terminate, nor do a transition.

Axiom M10 $a \cdot x \lfloor\!\lfloor y \Leftrightarrow a \cdot (x \parallel y)$.

Similar to the proof of Axiom M7, using deduction rule 36 instead of rule 34.

Axiom M11 $(x + y) \lfloor\!\lfloor z \Leftrightarrow x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$.

Left-hand side:

- Suppose $(x + y) \lfloor\!\lfloor z \downarrow$, then$_{33}$, $x + y \downarrow$ and $z \downarrow$, and thus$_6$ $x \downarrow$ and $z \downarrow$ or $y \downarrow$ and $z \downarrow$.
  - If $x \downarrow$ and $z \downarrow$, then$_{33}$ $x \lfloor\!\lfloor z \downarrow$ and thus$_6$ $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \downarrow$.
  - If $y \downarrow$ and $z \downarrow$, then$_{33}$ $y \lfloor\!\lfloor z \downarrow$ and thus$_6$ $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \downarrow$.
- Suppose $(x + y) \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$, then$_{34}$, $x + y \xrightarrow{\mathcal{U}_A} x''$ and $x' = x'' \parallel \lfloor z \rfloor_A$. If $x + y \xrightarrow{\mathcal{U}_A} x''$, then$_7$ $x \xrightarrow{\mathcal{U}_A} x''$ or $y \xrightarrow{\mathcal{U}_A} x''$.
  - if $x \xrightarrow{\mathcal{U}_A} x''$, then$_{34}$ $x \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'' \lfloor\!\lfloor \lfloor z \rfloor_A (= x')$ and thus$_7$ $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$ and $R(x', x')$.
  - if $y \xrightarrow{\mathcal{U}_A} x''$, then$_{34}$ $y \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'' \lfloor\!\lfloor \lfloor z \rfloor_A (= x')$ and thus$_7$ $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$ and $R(x', x')$.
- Suppose $(x + y) \lfloor\!\lfloor z \xrightarrow{a_n} x'$. Similar to the previous case, using rule 35 instead of rule 34.
- Suppose $(x + y) \lfloor\!\lfloor z \xrightarrow{a} x'$. Similar to the previous case, using rule 36 instead of rule 35.

Right-hand side:

- Suppose $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \downarrow$, then$_6$, $x \lfloor\!\lfloor z \downarrow$ or $y \lfloor\!\lfloor z \downarrow$.
  - If $x \lfloor\!\lfloor z \downarrow$ then$_{33}$ $x \downarrow$ and $z \downarrow$. Then$_6$ $x + y \downarrow$ and thus$_{33}$ $(x + y) \lfloor\!\lfloor z \downarrow$.
  - If $y \lfloor\!\lfloor z \downarrow$ then$_{33}$ $y \downarrow$ and $z \downarrow$. Then$_6$ $x + y \downarrow$ and thus$_{33}$ $(x + y) \lfloor\!\lfloor z \downarrow$.
- Suppose $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$, then$_7$ $x \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$ or $y \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$.
  - if $x \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$, then$_{34}$ $x \xrightarrow{\mathcal{U}_A} x''$ and $x' = x'' \parallel \lfloor z \rfloor_A$. If $x \xrightarrow{\mathcal{U}_A} x''$, then$_7$ $(x + y) \xrightarrow{\mathcal{U}_A} x''$ and thus$_{34}$ $(x + y) \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'' \parallel \lfloor z \rfloor_A$ $(= x')$. $R(x', x')$.
  - proof for $y \lfloor\!\lfloor z \xrightarrow{\mathcal{U}_A} x'$ is analogous.
- Suppose $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \xrightarrow{a_n} x'$. Similar to the previous case, using rule 35 instead of rule 34.
- Suppose $x \lfloor\!\lfloor z + y \lfloor\!\lfloor z \xrightarrow{a} x'$. Similar to the previous case, using rule 36 instead of

rule 35.

Axiom TR1 $\langle\langle x\rangle\rangle \leftrightarrows \langle\langle x, \emptyset, x\rangle\rangle$.

We proof this by using case distinction on the possible transitions x can do.

- Suppose $x \downarrow$, then$_9$ $\langle\langle x\rangle\rangle \xrightarrow{\mathcal{C}_\emptyset} \varepsilon$ and$_{10}$ $\langle\langle x, \emptyset, x\rangle\rangle \xrightarrow{\mathcal{C}_\emptyset} \varepsilon$. $R(\varepsilon, \varepsilon)$.
- Suppose $x \xrightarrow{\mathcal{U}_A} x'$, then$_{11}$ $\langle\langle x\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, \emptyset, x'\rangle\rangle$ and$_{14}$ $\langle\langle x, \emptyset, x\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, \emptyset, x'\rangle\rangle$. $R(\langle\langle x, \emptyset, x'\rangle\rangle, \langle\langle x, \emptyset, x'\rangle\rangle)$.
- Suppose $x \xrightarrow{a_n} x'$, then$_{12}$ $\langle\langle x\rangle\rangle \xrightarrow{a_n} \langle\langle x, \{a\}, x'\rangle\rangle$ and$_{15}$, since $a \notin \emptyset$, $\langle\langle x, \emptyset, x\rangle\rangle \xrightarrow{a_n} \langle\langle x, \{a\}, x'\rangle\rangle$. $R(\langle\langle x, \{a\}, x'\rangle\rangle, \langle\langle x, \{a\}, x'\rangle\rangle)$.
- Suppose $x \xrightarrow{a} x'$, then$_{13}$ $\langle\langle x\rangle\rangle \xrightarrow{a_0} \langle\langle x, \{a\}, x'\rangle\rangle$ and$_{17}$, since $a \notin \emptyset$, $\langle\langle x, \emptyset, x\rangle\rangle \xrightarrow{a_0} \langle\langle x, \{a\}, x'\rangle\rangle$. $R(\langle\langle x, \{a\}, x'\rangle\rangle, \langle\langle x, \{a\}, x'\rangle\rangle)$.

Axiom TR2 $\langle\langle x, \emptyset, \delta\rangle\rangle \leftrightarrows \delta$.

Neither side can terminate, nor do a transition.

Axiom TR3 $\langle\langle x, A, \delta\rangle\rangle \leftrightarrows \mathcal{R}_A \cdot \langle\langle x\rangle\rangle$ if $A \neq \emptyset$.

Since $A \neq \emptyset$, rule 8 can be applied to $\langle\langle x, A, \delta\rangle\rangle$, so $\langle\langle x, A, \delta\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$. Furthermore$_2$ $\mathcal{R}_A \cdot \langle\langle x\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$. $R(\langle\langle x\rangle\rangle, \langle\langle x\rangle\rangle)$. No termination or other transitions are possible.

Axiom TR4 $\langle\langle x, \emptyset, \varepsilon\rangle\rangle \leftrightarrows \mathcal{C}_\emptyset$.

Since$_1$ $\varepsilon \downarrow$, it holds that$_{10}$ $\langle\langle x, \emptyset, \varepsilon\rangle\rangle \xrightarrow{\mathcal{C}_\emptyset} \varepsilon$. Also$_2$ $\mathcal{C}_\emptyset \xrightarrow{\mathcal{C}_\emptyset} \varepsilon$ and $R(\varepsilon, \varepsilon)$. This is the only action both sides of the axiom can execute.

Axiom TR5 $\langle\langle x, A, \varepsilon\rangle\rangle \leftrightarrows \mathcal{C}_A + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle$ if $a \neq \emptyset$.

Looking at the right-hand side, it can be concluded that two transitions can take place$_{7,2}$ $\mathcal{C}_A + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle \xrightarrow{\mathcal{C}_A} \varepsilon$ and$_{7,2,1,4,5}$ $\mathcal{C}_A + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$. The left-hand side can do exactly the same transition$_{10}$ $\langle\langle x, A, \varepsilon\rangle\rangle \xrightarrow{\mathcal{C}_A} \varepsilon$, $R(\varepsilon, \varepsilon)$, and$_8$, since $A \neq \emptyset$, $\langle\langle x, A, \varepsilon\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$, $R(\langle\langle x\rangle\rangle, \langle\langle x\rangle\rangle)$. No other transitions are possible and neither process terminates.

Axiom TR6 $\langle\langle x, \emptyset, \mathcal{U}_B \cdot y\rangle\rangle \leftrightarrows \mathcal{U}_\emptyset \cdot \langle\langle x, \emptyset, y\rangle\rangle$.

Since$_{2,1,4,5}$ $\mathcal{U}_B \cdot y \xrightarrow{\mathcal{U}_B} y$ we conclude$_{14}$ $\langle\langle x, \emptyset, \mathcal{U}_B \cdot y\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, \emptyset, y\rangle\rangle$. Rule 8 cannot be applied, so no other transitions are possible. The right-hand side can also do only one transition$_{2,1,4,5}$, $\mathcal{U}_\emptyset \cdot \langle\langle x, \emptyset, y\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle e\rangle\rangle x\emptyset y$ and $R(\langle\langle x, \emptyset, y\rangle\rangle, \langle\langle x, \emptyset, y\rangle\rangle)$.

Axiom TR7 $\langle\langle x, A, \mathcal{U}_B \cdot y\rangle\rangle \leftrightarrows \mathcal{U}_\emptyset \cdot \langle\langle x, A, y\rangle\rangle + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle$ if $A \neq \emptyset$.

Looking at the right-hand side, it can be concluded that two transitions can take place$_{7,2,1,4,5}$, $\mathcal{U}_\emptyset \cdot \langle\langle x, A, y\rangle\rangle + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, A, y\rangle\rangle$ and apart from that$_{7,2,1,4,5}$, $\mathcal{U}_\emptyset \cdot \langle\langle x, A, y\rangle\rangle + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$. The left-hand side can do exactly the same transition. Since$_{2,1,4,5}$ $\mathcal{U}_B \cdot y \xrightarrow{\mathcal{U}_B} y$, we conclude that$_{14}$ $\langle\langle x, A, \mathcal{U}_B \cdot y\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, A, y\rangle\rangle$, and $R(\langle\langle x, A, y\rangle\rangle, \langle\langle x, A, y\rangle\rangle)$. Furthermore$_8$, since $A \neq \emptyset$, we conclude $\langle\langle x, A, \varepsilon\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$, and $R(\langle\langle x\rangle\rangle, \langle\langle x\rangle\rangle)$. No other transitions are possible and neither process terminates.

Axiom TR8 $\langle\langle x, \emptyset, a_n \cdot y\rangle\rangle \leftrightarrows a_n \cdot \langle\langle x, \{a\}, y\rangle\rangle$.

Similar to the proof of Axiom TR6, using deduction rule 15 instead of rule 14.

Axiom TR9 $\langle\langle x, A, a_n \cdot y\rangle\rangle \leftrightarrows a_n \cdot \langle\langle x, A \cup \{a\}, y\rangle\rangle + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle$ if $a \notin A \wedge A \neq \emptyset$.

Similar to the proof of Axiom TR7, using deduction rule 15 instead of rule 14.

Axiom TR10 $\langle\langle x, A, a_n \cdot y\rangle\rangle \leftrightarrows a \cdot \langle\langle x, A, y\rangle\rangle + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle$ if $a \in A$.

Similar to the proof of Axiom TR7, using deduction rule 16 instead of rule 14.

Axiom TR11 $\langle\langle x, \emptyset, a \cdot y\rangle\rangle \leftrightarrows a_0 \cdot \langle\langle x, \{a\}, y\rangle\rangle$

Similar to the proof of Axiom TR6, using deduction rule 17 instead of rule 14.

Axiom TR12 $\langle\langle x, A, a \cdot y\rangle\rangle \leftrightarrows a_0 \cdot \langle\langle x, A \cup \{a\}, y\rangle\rangle + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle$ if $a \notin A \wedge A \neq \emptyset$.

Similar to the proof of Axiom TR7, using deduction rule 17 instead of rule 14.

Axiom TR13 $\langle\langle x, A, a \cdot y\rangle\rangle \leftrightarrows a \cdot \langle\langle x, A, y\rangle\rangle + \mathcal{R}_A \cdot \langle\langle x\rangle\rangle$ if $a \in A$.

Similar to the proof of Axiom TR7, using deduction rule 18 instead of rule 14.

Axiom TR14 $\langle\langle x, A, y + z\rangle\rangle \leftrightarrows \langle\langle x, A, y\rangle\rangle + \langle\langle x, A, z\rangle\rangle$.

First of all, it can be easily seen by looking at the deduction rules that neither side can terminate. Furthermore$_8$, if $A \neq \emptyset$, $\langle\langle x, A, y + z\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$, $\langle\langle x, A, y\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$ and thus$_7$ $\langle\langle x, A, y\rangle\rangle + \langle\langle x, A, z\rangle\rangle \xrightarrow{\mathcal{R}_A} \langle\langle x\rangle\rangle$. $R(\langle\langle x\rangle\rangle, \langle\langle x\rangle\rangle)$. We make a case distinction on the actions that the left-hand and right-hand side can execute.

Left-hand side:

- Suppose $\langle\langle x, A, y + z\rangle\rangle \xrightarrow{\mathcal{U}_B} x'$. If $A = B$ and $A \neq \emptyset$ this can be a rollback, handled above. Otherwise$_{14}$, if $B = \emptyset$, $y + z \xrightarrow{\mathcal{U}_C} x''$ and $x' = \langle\langle x, A, x''\rangle\rangle$. if $y + z \xrightarrow{\mathcal{U}_C} x''$, then$_7$ $y \xrightarrow{\mathcal{U}_C} x''$ or $z \xrightarrow{\mathcal{U}_C} x''$.
  - if $y \xrightarrow{\mathcal{U}_C} x''$, then$_{14}$ $\langle\langle x, A, y\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, A, x''\rangle\rangle$. And thus$_7$, using $B = \emptyset$, $\langle\langle x, A, y\rangle\rangle +$

$\langle\langle x, A, z\rangle\rangle \xrightarrow{\mathcal{U}_\emptyset} \langle\langle x, A, x''\rangle\rangle$ $(= x')$. $R(x', x')$.

- if $z \xrightarrow{\mathcal{U}_C} x''$, then$_{14}$ $\langle\langle x, A, z\rangle\rangle \xrightarrow{\mathcal{U}_B} \langle\langle x, A, x''\rangle\rangle$. And thus$_7$, using $B = \emptyset$, $\langle\langle x, A, y\rangle\rangle + \langle\langle x, A, z\rangle\rangle \xrightarrow{\mathcal{U}_B} \langle\langle x, A, x''\rangle\rangle$ $(= x')$. $R(x', x')$.

- Suppose $\langle\langle x, A, y + z\rangle\rangle \xrightarrow{a_n} x'$. This case is similar to the previous case, using rule 15 if $a \notin A$ and rule 16 if $a \in A$.
- Suppose $\langle\langle x, A, y + z\rangle\rangle \xrightarrow{a} x'$. This case is similar to the previous case, using rule 17 if $a \notin A$ and rule 18 if $a \in A$.

Right-hand side:

- Suppose $\langle\langle x, A, y\rangle\rangle + \langle\langle x, A, z\rangle\rangle \xrightarrow{\mathcal{U}_B} x'$. If $A = B$ and $A \neq \emptyset$ this can be a rollback, handled above. Otherwise$_7$, $\langle\langle x, A, y\rangle\rangle \xrightarrow{\mathcal{U}_B} x'$ or $\langle\langle x, A, z\rangle\rangle \xrightarrow{\mathcal{U}_B} x'$.
  - If $\langle\langle x, A, y\rangle\rangle \xrightarrow{\mathcal{U}_B} x'$, then$_{14}$ if $B = \emptyset$, $y \xrightarrow{\mathcal{U}_C} x''$ and $x' = \langle\langle x, A, x''\rangle\rangle$. If $y \xrightarrow{\mathcal{U}_C} x''$, then$_7$ $y + z \xrightarrow{\mathcal{U}_C} x''$ and thus$_{14}$, using $B = \emptyset$, $\langle\langle x, A, y + z\rangle\rangle \xrightarrow{\mathcal{U}_B} \langle\langle x, A, x''\rangle\rangle$ $(= x')$. $R(x', x')$.
  - proof for $\langle\langle x, A, z\rangle\rangle \xrightarrow{\mathcal{U}_B} x'$ is analogous.
- Suppose $\langle\langle x, A, y\rangle\rangle + \langle\langle x, A, z\rangle\rangle \xrightarrow{a_n} x'$. This case is similar to the previous case, using rule 15 if $a \notin A$ and rule 16 if $a \in A$.
- Suppose $\langle\langle x, A, y\rangle\rangle + \langle\langle x, A, z\rangle\rangle \xrightarrow{a} x'$. This case is similar to the previous case, using rule 17 if $a \notin A$ and rule 18 if $a \in A$.

□

B.2 Elimination of PAtrans to BPA$_{\delta\varepsilon}$rec

Theorem (Eliminiation to BPA$_{\delta\varepsilon}$rec) For every PAtrans term t there exists a guarded linear recursive specification E over BPA$_{\delta\varepsilon}$ such that t is a solution of E.

Proof This theorem is proved by induction on the general structure of t. In the proof we use the definition of *linear* as given in [BW90]. Then, by definition, all linear BPA$_{\delta\varepsilon}$ terms are guarded. Note that this definition slightly differs from the definition in [BK84a] where guardedness of linear BPA$_{\delta\varepsilon}$ terms is not demanded.

1. $t \equiv \varepsilon$. Then t is a linear BPA$_{\delta\varepsilon}$ term.

2. $t \equiv \delta$. Then t is a linear BPA$_{\delta\varepsilon}$ term.

3. $t \equiv \mathbf{a}$ for $\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$. Then t is a linear BPA$_{\delta\varepsilon}$ term.

4. $t \equiv t_1 + t_2$ for PAtrans terms t_1 and t_2. By induction, there are linear BPA$_{\delta\varepsilon}$ terms s_1 and s_2 such that $T(\text{PAtrans}) \models s_1 = t_1$ and $T(\text{PAtrans}) \models s_2 = t_2$. But then also $T(\text{PAtrans}) \models s_1 + s_2 = t_1 altt_2$ and $s_1 + s_2$ is a linear BPA$_{\delta\varepsilon}$ term.

5. $t = t_1 \cdot t_2$ for PAtrans terms t_1 and t_2. By induction, there are linear BPA$_{\delta\varepsilon}$ terms s_1 and s_2 such that $T(\text{PAtrans}) \models s_1 = t_1$ and $T(\text{PAtrans}) \models s_2 = t_2$. Since s_1 is linear, $s_1 \equiv \varepsilon$, $s_1 \equiv \delta$, $s_1 \equiv \mathbf{a}$, $s_1 \equiv \mathbf{a} \cdot s_3$ or $s_1 \equiv s_3 + s_4$ for $\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$ and linear BPA$_{\delta\varepsilon}$ terms s_3 and s_4.
   - $s_1 \equiv \varepsilon$. Then $s_1 \cdot s_2 = \varepsilon \cdot s_2 = s_2$, which is a linear BPA$_{\delta\varepsilon}$ term.
   - $s_1 \equiv \delta$. Then $s_1 \cdot s_2 = \delta \cdot s_2 = \delta$, which is a linear BPA$_{\delta\varepsilon}$ term.
   - $s_1 \equiv \mathbf{a} \cdot s_3$ for $\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$ and linear BPA$_{\delta\varepsilon}$ term s_3. Then $s_1 \cdot s_2 = (\mathbf{a} \cdot s_3) \cdot s_2 = \mathbf{a} \cdot (s_3 \cdot s_2)$. By induction, $s_3 \cdot s_2$ is linear, so $\mathbf{a} \cdot (s_3 \cdot s_2)$ is linear and thus $s_1 \cdot s_2$ is linear.
   - $s_1 \equiv s_3 + s_4$ for linear BPA$_{\delta\varepsilon}$ terms s_3 and s_4. Then, $s_1 \cdot s_2 = (s_3 + s_4) \cdot s_2 = (s_3 \cdot s_2) + (s_4 \cdot s_2)$. Since, by induction, $s_3 \cdot s_2$ and $s_4 \cdot s_2$ are linear BPA$_{\delta\varepsilon}$ terms, $s_1 \cdot s_2$ is linear.

6. $t \equiv [t_1]_b$ for PAtrans term t_1 and $b \in \mathbb{A}$. By induction, there exists a linear BPA$_{\delta\varepsilon}$ term s_1 such that $T(\text{PAtrans}) \models s_1 = t_1$. So, $s_1 \equiv \varepsilon$, $s_1 \equiv \delta$, $s_1 \equiv \mathbf{a}$, $s_1 \equiv \mathbf{a} \cdot s_3$ or $s_1 \equiv s_3 + s_4$ for $\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$ and linear BPA$_{\delta\varepsilon}$ terms s_3 and s_4.
   - $s_1 \equiv \varepsilon$. Then $[s_1]_b = [\varepsilon]_b = \varepsilon$, which is a linear BPA$_{\delta\varepsilon}$ term.
   - $s_1 \equiv \delta$. Then $[s_1]_b = [\delta]_b = \delta$, which is a linear BPA$_{\delta\varepsilon}$ term.
   - $s_1 \equiv \mathbf{a} \cdot s_3$ for $\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$ and linear BPA$_{\delta\varepsilon}$ term s_3. Then $[s_1]_b = [\mathbf{a} \cdot s_3]_b = [\mathbf{a}]_b \cdot [s_3]_b$. By induction, $[s_3]_b$ is linear. Furthermore, $[\mathbf{a}]_b \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$ so $[\mathbf{a}]_b \cdot [s_3]_b$ is linear and thus $[s_1]_b$ is linear.
   - $s_1 \equiv s_3 + s_4$ for linear BPA$_{\delta\varepsilon}$ terms s_3 and s_4. Then $[s_1]_b = [s_3 + s_4]_b = [s_3]_b + [s_4]_b$. By induction, $[s_3]_b$ and $[s_4]_b$ are linear and thus $[s_1]_b$ is linear.

7. $t \equiv \lfloor t_1 \rfloor_A$ for PAtrans term t_1 and $A \subseteq \mathbb{A}$. This case is treated analogous to case 6.

8. $t \equiv t_1 \lfloor\!\lfloor t_2$ for PAtrans terms t_1 and t_2. By induction, there exists a linear BPA$_{\delta\varepsilon}$ term s_1 such that $T(\text{PAtrans}) \models s_1 = t_1$. But since all recursive specifications in PAtrans are guarded, there is a linear BPA$_{\delta\varepsilon}$ term r_1 which is in head normal form, such that $T(\text{PAtrans}) \models r_1 = s_1 = t_1$. We prove this case by induction on the structure of r_1:
   - $r_1 \equiv \varepsilon$. Then $s_1 \lfloor\!\lfloor s_2 = \varepsilon \lfloor\!\lfloor s_2$, which equals δ if $s_2 \neq \varepsilon$ and ε otherwise. Both are linear BPA$_{\delta\varepsilon}$ term.
   - $r_1 \equiv \delta$. Then $s_1 \lfloor\!\lfloor s_2 = \delta \lfloor\!\lfloor s_2 = \delta$, which is a linear BPA$_{\delta\varepsilon}$ term.
   - $r_1 \equiv \mathbf{a} \cdot r_2$ for $\mathbf{a} \in \mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$ and linear BPA$_{\delta\varepsilon}$ term r_2.Then PAtrans $\vdash t = (\mathbf{a} \cdot r_2) \lfloor\!\lfloor t_2$. Depending on the structure of $\mathbf{a}$, PAtrans $\vdash t = \mathcal{U}_A(r_2 \parallel \lfloor t_2 \rfloor_A)$, PAtrans $\vdash t = a_n(r_2 \parallel [t_2]_a)$ or PAtrans $\vdash t = a(r_2 \parallel t_2)$. So there is a function f such that PAtrans $\vdash t = (\mathbf{a} \cdot r_2) \lfloor\!\lfloor t_2 = \mathbf{a} \cdot (r_2 \parallel f(t_2))$ where $f(x) \in \{\lfloor x \rfloor_A, [x]_a, x\}$. We proved (6,7) that $f(t_2)$ is a linear BPA$_{\delta\varepsilon}$ term.
     Furthermore, PAtrans $\vdash r_2 \parallel f(t_2) = r_2 \lfloor\!\lfloor f(t_2) + f(t_2) \lfloor\!\lfloor r_2$ and, by induction, both $r_2 \lfloor\!\lfloor f(t_2)$ and $f(t_2) \lfloor\!\lfloor r_2$ are linear BPA$_{\delta\varepsilon}$ terms and thus $r_2 \parallel f(t_2)$ is a linear BPA$_{\delta\varepsilon}$

term. Since **a** is in $\mathbb{A} \cup \mathbb{L} \cup \mathbb{UL}$, $\mathbf{a} \cdot (r_2 \parallel f(t_2))$ is linear and thus $(\mathbf{a} \cdot r_2) \mathbin{\lfloor\!\lfloor} t_2$ is a linear $\mathrm{BPA}_{\delta\varepsilon}$ term.

- $r_1 \equiv r_2 + r_3$ for r_2 and r_3 linear $\mathrm{BPA}_{\delta\varepsilon}$ terms. Then PAtrans $\vdash t = (r_2 + r_3) \mathbin{\lfloor\!\lfloor} t_2 = r_2 \mathbin{\lfloor\!\lfloor} t_2 + r_3 \mathbin{\lfloor\!\lfloor} t_2$. By induction there exist linear $\mathrm{BPA}_{\delta\varepsilon}$ terms s_2 and s_3 such that PAtrans $\vdash s_2 = r_2 \mathbin{\lfloor\!\lfloor} t_2$ and PAtrans $\vdash s_3 = r_3 \mathbin{\lfloor\!\lfloor} t_2$. Then also $T(\text{PAtrans}) \models t = (r_2 + r_3) \mathbin{\lfloor\!\lfloor} t_2 = r_2 \mathbin{\lfloor\!\lfloor} t_2 + r_3 \mathbin{\lfloor\!\lfloor} t_2 = s_2 + s_3$ and $s_2 + s_3$ is a linear $\mathrm{BPA}_{\delta\varepsilon}$ term.
- $r_1 \equiv X$ for some recursion variable X. This case is not possible since r_1 should be guarded.

9. $t \equiv t_1 \parallel t_2$ for PAtrans terms t_1 and t_2. PAtrans $\vdash t_1 \parallel t_2 = t_1 \parallel t_2 = t_1 \mathbin{\lfloor\!\lfloor} t_2 + t_2 \mathbin{\lfloor\!\lfloor} t_1$. We have proved (8) that there exist linear $\mathrm{BPA}_{\delta\varepsilon}$ terms s_1 and s_2 such that $T(\text{PAtrans}) \models s_1 = t_1 \mathbin{\lfloor\!\lfloor} t_2$ and $T(\text{PAtrans}) \models s_2 = t_2 \mathbin{\lfloor\!\lfloor} t_3$. But then, $T(\text{PAtrans}) \models t_1 \parallel t_2 = t_1 \mathbin{\lfloor\!\lfloor} t_2 + t_2 \mathbin{\lfloor\!\lfloor} t_1 = s_1 + s_2$ and $s_1 + s_2$ is a linear $\mathrm{BPA}_{\delta\varepsilon}$ term.

10. $t \equiv \langle\langle t_1 \rangle\rangle$ for closed PAtrans term t_1. PAtrans $\vdash \langle\langle t_1 \rangle\rangle = \langle\langle t_1, \emptyset, t_1 \rangle\rangle$. As is proved in 11, there exists a linear $\mathrm{BPA}_{\delta\varepsilon}$ term s_1 such that $T(\text{PAtrans}) \models s_1 = \langle\langle t_1, \emptyset, t_1 \rangle\rangle$ and thus $T(\text{PAtrans}) \models s_1 = \langle\langle t_1 \rangle\rangle$.

11. $t \equiv \langle\langle t_1, A, t_2 \rangle\rangle$ for PAtrans terms t_1 and t_2 and $A \subseteq \mathbb{A}$. We give a set of recursive equations, E, and prove by induction on the structure of t_2 that for all terms $\langle\langle t_1, A, t_2 \rangle\rangle$ there exists a linear $\mathrm{BPA}_{\delta\varepsilon}$ term s such that $T(\text{PAtrans}) \models s = \langle\langle t_1, A, t_2 \rangle\rangle$. Let E, the set of recursive equations, be defined as follows:

$$
\begin{array}{lllll}
E = \{ & X_{\delta}^{t,\emptyset} & = \delta, & X_{\delta}^{t,A} & = \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
 & X_{\varepsilon}^{t,\emptyset} & = \mathcal{C}_\emptyset, & X_{\varepsilon}^{t,A} & = \mathcal{C}_A + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
 & X_{\mathcal{U}_B \cdot t_1}^{t,\emptyset} & = \mathcal{U}_\emptyset \cdot X_{t_1}^{t,\emptyset}, & X_{\mathcal{U}_B \cdot t_1}^{t,A} & = \mathcal{U}_\emptyset \cdot X_{t_1}^{t,A} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
 & X_{c_n \cdot t_1}^{t,\emptyset} & = c_n \cdot X_t^{t,\{c\}}, & X_{b_n \cdot t_1}^{t,A} & = b_n \cdot X_{t_1}^{t,A \cup \{b\}} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
 & X_{a_n \cdot t_1}^{t,A} & = a \cdot X_{t_1}^{t,A} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, & X_{c \cdot t_1}^{t,\emptyset} & = c_0 \cdot X_{t_1}^{t,\{c\}}, \\
 & X_{b \cdot t_1}^{t,A} & = b_0 \cdot X_{t_1}^{t,A \cup \{b\}} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, & X_{a \cdot t_1}^{t,A} & = a \cdot X_{t_1}^{t,A} + \mathcal{R}_A \cdot X_t^{t,\emptyset}, \\
 & X_{t_1 + t_2}^{t,B} & = X_{t_1}^{t,B} + X_{t_2}^{t,B} & & \\
 & \mid A \subseteq \mathbb{A},\ A \neq \emptyset,\ B \subseteq \mathbb{A},\ n \in \mathbb{N},\ n > 0,\ a \in A,\ b \in \mathbb{A} \setminus A,\ c \in \mathbb{A} & & & \}
\end{array}
$$

As can be easily seen by comparing axioms TR2–20 with the recursive equations in E, $T(\text{PAtrans}) \models \langle X_{t_2}^{t_1,A} | E \rangle = \langle\langle t_1, A, t_2 \rangle\rangle$ holds for all t_1, A and t_2. So there exists a linear $\mathrm{BPA}_{\delta\varepsilon}$ term for every $\langle\langle t_1, A, t_2 \rangle\rangle$ in PAtrans, viz. $\langle X_{t_2}^{t_1,A} | E \rangle$. Furthermore, since PAtrans $\vdash \langle\langle t \rangle\rangle = \langle\langle t, \emptyset, t \rangle\rangle$, $T(\text{PAtrans}) \models \langle\langle t \rangle\rangle = \langle X_t^{t,\emptyset} | E \rangle$.

So for all PAtrans terms t there exists a linear $\mathrm{BPA}_{\delta\varepsilon}$ term s such that PAtrans $\models s = t$.

□

C

Proofs of Test Derivation Theory

In this appendix we give full proofs of some lemmas and theorems that are given in Chapter 10.

C.1 Proof of Lemma 10.6.5

Lemma Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ and $\varsigma_0, \varsigma_1 \in L^*$, $\varsigma_1 \neq \varepsilon$. Then

$$\varsigma_0\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \quad \Longleftrightarrow \quad \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0|}(s_0 \textbf{ after } \varsigma_0)})$$

where $|\varsigma|$ is the length of trace ς.

Proof We prove this lemma by using induction on the structure of ς_0.

$\varsigma_0 = \varepsilon$.
Induction hypothesis (IH): Let $\varsigma_0^m \in L^*$ such that $|\varsigma_0^m| = m$. Then,

$$\varsigma_0^m\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \quad \Longleftrightarrow \quad \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|}(s_0 \textbf{ after } \varsigma_0^m)}) \ .$$

$$\begin{array}{ll} & \varsigma_0\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\ \Longleftrightarrow & \{\ \varsigma_0 = \varepsilon,\ \varepsilon\varsigma_1 = \varsigma_1\ \} \\ & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\ \Longleftrightarrow & \{\ |\varsigma_0| = |\varepsilon| = 0,\ s_0 \textbf{ after } \varepsilon = \{s_0\}\ \} \\ & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0|}(s_0 \textbf{ after } \varsigma_0)}) \end{array}$$

We prove that

$$\begin{array}{ll} & \varsigma_0^{m+1}\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\ \Longleftrightarrow & \\ & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^{m+1}|}(s_0 \textbf{ after } \varsigma_0^{m+1})}) \ . \end{array}$$

We prove this by case distinction for $\varsigma_0^{m+1}{=}\varsigma_0^m a$ for $a \in \mathbb{L}_?$ and $\varsigma_0^{m+1}{=}\varsigma_0^m b$ for $b \in \mathbb{L}_!$. Note that $\mathbb{L}_? \cup \mathbb{L}_! = L$, so these cases cover all labels in L.

$\varsigma_0^{m+1} = \varsigma_0^m a$ for $a \in \mathbb{L}_?$:

$$\begin{array}{ll} & \varsigma_0^{m+1}\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\ \Longleftrightarrow & \left\{ \ \varsigma_0^{m+1} = \varsigma_0^m a \ \right\} \\ & \varsigma_0^m a\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\ \Longleftrightarrow & \left\{ \begin{array}{l} \text{(IH) } \varsigma_0^m a\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \ \equiv \\ \quad a\varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|}(s_0 \textbf{ after } \varsigma_0^m)}) \end{array} \right\} \\ & a\varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|}(s_0 \textbf{ after } \varsigma_0^m)}) \\ \Longleftrightarrow & \left\{ \begin{array}{l} \text{(gentest, second option) } n > 0, \ a \in \mathbb{L}_?, \\ (s_0 \textbf{ after } \varsigma_0^m) \textbf{ after } a \neq \emptyset \end{array} \right\} \\ & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|-1}((s_0 \textbf{ after } \varsigma_0^m) \textbf{ after } a)}) \\ \Longleftrightarrow & \left\{ \begin{array}{l} n - |\varsigma_0^m| - 1 = n - (|\varsigma_0^m| + 1) = n - |\varsigma_0^m a|, \\ (s_0 \textbf{ after } \varsigma_0^m) \textbf{ after } a = s_0 \textbf{ after } \varsigma_0^m a \end{array} \right\} \\ & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m a|}(s_0 \textbf{ after } \varsigma_0^m a)}) \\ \Longleftrightarrow & \left\{ \ \varsigma_0^{m+1} = \varsigma_0^m a \ \right\} \\ & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^{m+1}|}(s_0 \textbf{ after } \varsigma_0^{m+1})}) \end{array}$$

$\varsigma_0^{m+1} = \varsigma_0^m b$ for $b \in \mathbb{L}_!$:

$$\begin{array}{ll} & \varsigma_0^{m+1}\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\ \Longleftrightarrow & \left\{ \ \varsigma_0^{m+1} = \varsigma_0^m b \ \right\} \\ & \varsigma_0^m b\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\ \Longleftrightarrow & \left\{ \begin{array}{l} \text{(IH) } \varsigma_0^m b\varsigma_1 \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \ \equiv \\ \quad b\varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|}(s_0 \textbf{ after } \varsigma_0^m)}) \end{array} \right\} \\ & b\varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|}(s_0 \textbf{ after } \varsigma_0^m)}) \end{array}$$

$$
\begin{array}{ll}
 & b\varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|}(s_0 \textbf{ after } \varsigma_0^m)}) \\
\Longleftrightarrow & \{ \text{ (gentest, third option) } n > 0 \ \} \\
 & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m|-1}((s_0 \textbf{ after } \varsigma_0^m) \textbf{ after } b)}) \\
\Longleftrightarrow & \left\{ \begin{array}{l} n - |\varsigma_0^m| - 1 = n - (|\varsigma_0^m| + 1) = n - |\varsigma_0^m b|, \\ (s_0 \textbf{ after } \varsigma_0^m) \textbf{ after } b = s_0 \textbf{ after } \varsigma_0^m b \end{array} \right\} \\
 & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^m b|}(s_0 \textbf{ after } \varsigma_0^m b)}) \\
\Longleftrightarrow & \{ \ \varsigma_0^{m+1} = \varsigma_0^m b \ \} \\
 & \varsigma_1 \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma_0^{m+1}|}(s_0 \textbf{ after } \varsigma_0^{m+1})})
\end{array}
$$

□

C.2 Proof of Theorem 10.6.7

Theorem Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. Then

$$\text{test suite } \bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})} \text{ is complete.}$$

Proof Let s be $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$ and T be $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$. Then,

$$
\begin{array}{ll}
 & T \text{ is complete} \\
\equiv & \{ \text{ definition of complete test suites } \} \\
 & \forall i \quad i \textbf{ rrconf } s \ \Leftrightarrow \ i \textbf{ passes } T \\
\equiv & \{ \text{ definition of } \textbf{rrconf} \text{ and } \textbf{passes} \ \} \\
 & \forall i \quad \forall \varsigma \in \text{traces}(s) \ \exp(i \textbf{ after } \varsigma) \subseteq \exp(s \textbf{ after } \varsigma) \\
 & \qquad \Longleftrightarrow \\
 & \qquad \neg \exists t \in T \ \exists \varsigma \in \text{traces}(i) \ \varsigma; \textbf{fail} \in \text{traces}(t)
\end{array}
$$

We prove this by proving exhaustiveness ($\Leftarrow$) and soundness ($\Rightarrow$) separately.

- Exhaustiveness.

$$
\begin{array}{ll}
\forall i & \forall \varsigma \in \text{traces}(s) \ \exp(i \textbf{ after } \varsigma) \subseteq \exp(s \textbf{ after } \varsigma) \\
 & \Longleftarrow \\
 & \neg \exists t \in T \ \exists \varsigma \in \text{traces}(i) \ \varsigma; \textbf{fail} \in \text{traces}(t)
\end{array}
$$

  We prove exhaustiveness by contradiction:
  Let $\varsigma \in \text{traces}(s)$ and $b \in \exp(i \textbf{ after } \varsigma)$ such that $b \notin \exp(s \textbf{ after } \varsigma)$. Then, we

prove that $\exists t \in T\ \exists \varsigma' \in \text{traces}(i)\ \varsigma'; \textbf{fail} \in \text{traces}(t)$.

$$
\begin{array}{ll}
 & \exists t \in T\ \exists \varsigma' \in \text{traces}(i)\ \varsigma'; \textbf{fail} \in \text{traces}(t) \\
\Leftarrow & \{\ b \in \exp(i\ \textbf{after}\ \varsigma) \Rightarrow \varsigma b \in \text{traces}(i),\ \text{Let}\ \varsigma' = \varsigma; b\ \} \\
 & \exists t \in T\ \varsigma; b; \textbf{fail} \in \text{traces}(t) \\
\Leftarrow & \{\ \text{Definition of T}\ \} \\
 & \exists n > 0\ \varsigma; b; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})}) \\
\equiv & \{\ \text{Lemma 10.6.5}\ \} \\
 & \exists n > 0\ b; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^{n-|\varsigma|}(\{s_0\ \textbf{after}\ \varsigma\})}) \\
\Leftarrow & \left\{ \begin{array}{l} \text{gentest (third option), let}\ n > |\varsigma|, \\ b \notin \exp(s_0\ \textbf{after}\ \varsigma) \Rightarrow b \in \mathbb{L}_! \setminus \exp(s_0\ \textbf{after}\ \varsigma) \end{array} \right\} \\
 & \text{true}
\end{array}
$$

- Soundness.

$$
\begin{array}{ll}
\forall i & \forall \varsigma \in \text{traces}(s)\ \exp(i\ \textbf{after}\ \varsigma) \subseteq \exp(s\ \textbf{after}\ \varsigma) \\
 & \Longrightarrow \\
 & \neg \exists t \in T\ \exists \varsigma \in \text{traces}(i)\ \varsigma; \textbf{fail} \in \text{traces}(t)
\end{array}
$$

Soundness is also proved by contradiction:
Let $t \in T$ and $\varsigma \in \text{traces}(i)$ such that $\varsigma; \textbf{fail} \in \text{traces}(t)$. Then, by definition of T, $\exists n > 0\ \varsigma; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^n(\{s_0\})})$.
Let $m > 0$ such that $\varsigma; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})})$.
We prove that $\exists \varsigma' \in \text{traces}(s) \exists b \in \exp(i\ \textbf{after}\ \varsigma')\ b \notin \exp(s\ \textbf{after}\ \varsigma')$.
Let $\varsigma'' \in \text{traces}(s)$ and $b'' \in \exp(i\ \textbf{after}\ \varsigma'')$.
Then, we prove that $b'' \notin \exp(s\ \textbf{after}\ \varsigma'')$. Since $\varsigma; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})})$, using Lemma 10.6.6, $\exists \varsigma' \in \text{traces}(s) \exists b \in \mathbb{L}_!\ \varsigma = \varsigma'; b$. Let $\varsigma = \varsigma''; b''$. Then,

$$
\begin{array}{ll}
 & \varsigma; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})}) \\
\equiv & \{\ \varsigma = \varsigma''; b''\ \} \\
 & \varsigma''; b''; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^m(\{s_0\})}) \\
\equiv & \{\ \text{Lemma 10.6.5}\ \} \\
 & b''; \textbf{fail} \in \text{traces}(\overline{\text{gentest}^{m-|\varsigma''|}(s_0\ \textbf{after}\ \varsigma'')}) \\
\Rightarrow & \{\ \text{Definition of algorithm gentest (third option)}\ \} \\
 & b'' \in \mathbb{L}_! \setminus \exp(s_0\ \textbf{after}\ \varsigma'') \\
\Rightarrow & \{\ \text{Set theory}\ \} \\
 & b'' \notin \exp(s_0\ \textbf{after}\ \varsigma'')
\end{array}
$$

□

C.3 Proof of Lemma 10.6.9

Lemma Let s be an MRRTS $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$, L be the set of labels in s, ς be a trace in s and E_ς be the set of expected responses after execution of ς in s:

$$E_\varsigma \quad = \quad \{b \in \mathbb{L}_! \mid \exists a \in \mathrm{req}(b)\ \exists \varsigma_0, \varsigma_1 \in L^* \ (\varsigma = \varsigma_0 a \varsigma_1 \wedge \neg \exists b' \in \mathrm{resp}(a)\ b' \in \varsigma_1)\} \ .$$

Then,

1. $\forall a \in \mathbb{L}_?\ E_\varsigma \cup \mathrm{resp}(a) = E_{\varsigma a}$
2. $\forall b \in \mathbb{L}_!\ E_\varsigma \setminus \mathrm{resp}(b) = E_{\varsigma b}$

Proof

1. $\forall a \in \mathbb{L}_?\ E_\varsigma \cup \mathrm{resp}(a) = E_{\varsigma a}$. Let $a \in \mathbb{L}_?$. Then,

$$\begin{array}{ll}
 & E_{\varsigma a} \\
= & \{ \text{ definition of } E \ \} \\
 & \left\{ b \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a' \in \mathrm{req}(b)\ \exists \varsigma_0, \varsigma_1 \in L^* \\ \quad (\varsigma a = \varsigma_0 a' \varsigma_1 \wedge \neg \exists b' \in \mathrm{resp}(a')\ b' \in \varsigma_1) \end{array} \right\} \\
= & \{ \text{ case distinction: } a' = a \wedge \varsigma_1 = \varepsilon \text{ and } \varsigma_1 = \varsigma_1' a \ \} \\
 & \{b \in \mathbb{L}_! \mid a \in \mathrm{req}(b),\ \neg \exists b' \in \mathrm{resp}(a)\ b' \in \varepsilon)\} \\
 & \cup \left\{ b \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a' \in \mathrm{req}(b)\ \exists \varsigma_0, \varsigma_1' \in L^* \\ \quad (\varsigma a = \varsigma_0 a' \varsigma_1' a \wedge \neg \exists b' \in \mathrm{resp}(a')\ b' \in \varsigma_1' a) \end{array} \right\} \\
= & \{ \ \varsigma_1 = \emptyset \ \Rightarrow \ \neg \exists b' \in \mathrm{resp}(a)\ b' \in \varsigma_1 \ \} \\
 & \{b \in \mathbb{L}_! \mid a \in \mathrm{req}(b)\} \\
 & \cup \left\{ b \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a' \in \mathrm{req}(b)\ \exists \varsigma_0, \varsigma_1' \in L^* \\ \quad (\varsigma a = \varsigma_0 a' \varsigma_1' a \wedge \neg \exists b' \in \mathrm{resp}(a')\ b' \in \varsigma_1' a) \end{array} \right\} \\
= & \{ \text{ (definition of req and resp) } \{b \in \mathbb{L}_! \mid a \in \mathrm{req}(b)\} = \mathrm{resp}(a) \ \} \\
 & \mathrm{resp}(a) \\
 & \cup \left\{ b \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a' \in \mathrm{req}(b)\ \exists \varsigma_0, \varsigma_1' \in L^* \\ \quad (\varsigma a = \varsigma_0 a' \varsigma_1' a \wedge \neg \exists b' \in \mathrm{resp}(a')\ b' \in \varsigma_1' a) \end{array} \right\} \\
= & \left\{ \begin{array}{l} \varsigma a = \varsigma_0 a' \varsigma_1' a \equiv \varsigma = \varsigma_0 a' \varsigma_1', \\ a \in \mathbb{L}_? \Rightarrow (\forall b' \in \mathrm{resp}(a')\ b' \in \varsigma_1' a \equiv b' \in \varsigma_1') \end{array} \right\} \\
 & \mathrm{resp}(a) \\
 & \cup \left\{ b \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a' \in \mathrm{req}(b)\ \exists \varsigma_0, \varsigma_1' \in L^* \\ \quad (\varsigma = \varsigma_0 a' \varsigma_1' \wedge \neg \exists b' \in \mathrm{resp}(a')\ b' \in \varsigma_1') \end{array} \right\} \\
= & \{ \text{ definition of } E \ \} \\
 & \mathrm{resp}(a) \cup E_\varsigma
\end{array}$$

2. $\forall b \in \mathbb{L}_! \; E_\varsigma \setminus \text{resp}(b) = E_{\varsigma b}$. Let $b \in \mathbb{L}_!$. Then,

$$
\begin{array}{ll}
 & E_{\varsigma b} \\
= & \{ \text{ definition of } E \ \} \\
 & \left\{ b' \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a \in \text{req}(b') \; \exists \varsigma_0, \varsigma_1 \in L^* \\ \quad (\varsigma b = \varsigma_0 a \varsigma_1 \wedge \neg \exists b'' \in \text{resp}(a) \; b'' \in \varsigma_1) \end{array} \right\} \\
= & \{ \; b \in \mathbb{L}_! \wedge a \in \mathbb{L}_? \;\Rightarrow\; \exists \varsigma_1' \in L^* \; \varsigma_1 = \varsigma_1' b \; \} \\
 & \left\{ b' \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a \in \text{req}(b') \; \exists \varsigma_0, \varsigma_1' \in L^* \\ \quad (\varsigma b = \varsigma_0 a \varsigma_1' b \wedge \neg \exists b'' \in \text{resp}(a) \; b'' \in \varsigma_1' b) \end{array} \right\} \\
= & \{ \text{ case distinction: } b' \in \varsigma_1' \text{ and } b'' = b \ \} \\
 & \left\{ b' \in \mathbb{L}_! \;\middle|\; \begin{array}{l} \exists a \in \text{req}(b') \; \exists \varsigma_0, \varsigma_1' \in L^* \\ \quad (\varsigma b = \varsigma_0 a \varsigma_1' b \wedge \neg \exists b'' \in \text{resp}(a) \; b'' \in \varsigma_1' \vee b'' = b) \end{array} \right\} \\
= & \{ \text{ set theory, } \varsigma b = \varsigma_0 a \varsigma_1' b \equiv \varsigma = \varsigma_0 a \varsigma_1' \ \} \\
 & \{ b' \in \mathbb{L}_! \mid \exists a \in \text{req}(b') \; \exists \varsigma_0, \varsigma_1' \in L^* \; (\varsigma = \varsigma_0 a \varsigma_1' \wedge \neg \exists b'' \in \text{resp}(a) \; b'' \in \varsigma_1') \} \\
 & \setminus \{ b' \in \mathbb{L}_! \mid \exists a \in \text{req}(b') \; \exists b'' \in \text{resp}(a) \; b'' = b \} \\
= & \{ \text{ definition of } E, \text{req}, \text{resp} \ \} \\
 & E_\varsigma \setminus \{ b' \in \mathbb{L}_! \mid \exists b'' \in \text{resp}(b') \; b'' = b \} \\
= & \{ \text{ definition of resp } \} \\
 & E_\varsigma \setminus \text{resp}(b)
\end{array}
$$

□

C.4 Proof of Lemma 10.6.12

Lemma Let MRRTS i be $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. Then,

$$\forall b \in \mathbb{L}_! \quad \varsigma b \in \text{traces}(i) \;\Rightarrow\; b \in E_\varsigma$$

where E_ς is as defined in Lemma 10.6.9.

Proof Let $b \in \mathbb{L}_!$. Then,

$$\begin{array}{ll}
 & \varsigma b \in \text{traces}(i) \\
\Rightarrow & \{\ i \text{ is an MRRTS, definition of req and resp}\ \} \\
 & \varsigma b\,|_{\text{req}(b)\cup\text{resp}(b)} \in \text{alt}(\text{req}(b), \text{resp}(b)) \\
\Rightarrow & \{\ \text{Definition of alt, } b \in \text{resp}(b)\ \} \\
 & \exists a \in \text{req}(b) \exists \varsigma' \in L^* \; \varsigma b = \varsigma' a b \\
\Rightarrow & \{\ \text{For } \varsigma_0 = \varsigma' \text{ and } \varsigma_1 = \varepsilon, \varsigma = \varsigma_0 a \varsigma_1\ \} \\
 & \exists a \in \text{req}(b) \exists \varsigma_0, \varsigma_1 \in L^* \; (\varsigma = \varsigma_0 a \varsigma_1 \;\wedge\; \neg\exists b' \in \text{resp}(a)\; b' \in \varsigma_1) \\
\Rightarrow & \{\ \text{Definition of } E_\varsigma\ \} \\
 & b \in E_\varsigma
\end{array}$$

□

C.5 Proof of Theorem 10.6.13

Theorem Let s be a specification $\langle S, \mathbb{L}, \rightarrow, s_0 \rangle$. Then test suite $\bigcup_{n>0} \overline{\text{gentest}^n_\emptyset(\{s_0\})}$ is complete.

Proof Let T be $\bigcup_{n>0} \overline{\text{gentest}^n(\{s_0\})}$ and $T_\emptyset$ be $\bigcup_{n>0} \overline{\text{gentest}^n_\emptyset(\{s_0\})}$. In Theorem 10.6.7 we proved that test suite T is complete. To prove that $T_\emptyset$ is complete as well, we prove that

$$\begin{array}{ll}
 & T_\emptyset \text{ is complete} \iff T \text{ is complete.} \\
\equiv & \{\ \text{definition of complete test suites}\ \} \\
 & \forall i \quad i \textbf{ rrconf } s \;\Leftrightarrow\; i \textbf{ passes } T_\emptyset \iff \forall i \quad i \textbf{ rrconf } s \;\Leftrightarrow\; i \textbf{ passes } T \\
\equiv & \{\quad\} \\
 & \forall i \quad i \textbf{ rrconf } s \;\Leftrightarrow\; i \textbf{ passes } T_\emptyset \Leftrightarrow i \textbf{ passes } T \\
\equiv & \{\ \text{Theorem 10.6.7}\ \} \\
 & \forall i \quad i \textbf{ passes } T_\emptyset \Leftrightarrow i \textbf{ passes } T
\end{array}$$

- $\forall i \quad i$ **passes** $T_\emptyset \Leftarrow i$ **passes** T

  Let MRRTS i be an implementation, $t \in T_\emptyset$ be a test case and $\varsigma \in \text{traces}(i)$ be a trace such that $\varsigma;\mathbf{fail} \in \text{traces}(t)$. Then we prove that i does not pass T.

  Lemma 10.6.10 proves that all traces in all test cases in $T_\emptyset$ are traces in some test case in T, so there exists a test case t' in T such that $\varsigma;\mathbf{fail} \in \text{traces}(t')$ and thus i does not pass T.

- $\forall i \quad i$ **passes** $T_\emptyset \Rightarrow i$ **passes** T.

  We prove this by contradiction. Let MRRTS i be an implementation, $t \in T$ be a test case $b \in \mathbb{L}_!$ be a response label and ςb be a trace in i such that $\varsigma b; fail \in \text{traces}(t)$. Then we prove that there exists a test case $t' \in T_\emptyset$ such that $\varsigma b;\mathbf{fail} \in \text{traces}(t')$.

  By inspecting Algorithm 10.6.8 and using Lemma 10.6.11, it can be concluded that this trace can only occur in $T_\emptyset$ if

$$b;\mathbf{fail} \;\in\; \text{traces}(\bigcup_{n>0} \overline{\text{gentest}_{E_\varsigma}^{n-|\varsigma_0|}(s_0 \;\mathbf{after}\; \varsigma)}) \;.$$

  This can only be the case if $b \in E_\varsigma \setminus \exp(s_0 \;\mathbf{after}\; \varsigma)$. Since ςb is a trace in i, Lemma 10.6.12 proves that $b \in E_\varsigma$. From Lemma 10.6.5 and $\varsigma b; fail \in \text{traces}(t)$ it follows that $b \notin \exp(s_0 \;\mathbf{after}\; \varsigma)$. So $b \in E_\varsigma \setminus \exp(s_0 \;\mathbf{after}\; \varsigma)$.

□

Summary

Nowadays, more and more activity takes place via the Internet. This means that Internet applications become more important. Since these applications also grow in size and complexity, specifying and implementing such applications becomes harder. Many issues arise, like session management and the correct implementation of parallel use by multiple clients. In our opinion, formal methods can help in tackling these problems.

In this thesis, a new specification language, *DiCons*, is introduced, which can be used for specifying a specific group of Internet applications, the so-called distributed consensus applications. These are applications which can be used by a group of clients to reach a common goal. Users do not have to physically meet and all communication takes place asynchronously. Examples of such applications are the drawing of *Sinterklaaslootjes*, a vote, an auction, but also the scheduling of a meeting.

We first inspect the differences between Web-based and "normal" window-based applications. Next, we have a look at the differences and similarities between some example applications and we determine the risks that are involved by using Web applications. From that, we abstract concepts that are of importance for all applications in the domain we focus on.

With these concepts in mind, we develop a formal specification language based on process algebra. Apart from the default operators for sequential and alternative composition, the language contains three conditional operators: conditional branching, conditional repetition and conditional disrupts. We explain how states and time are added to the language and which communication primitives we make use of for modelling the communication via the Internet between users and the application itself. Apart from that, we add the possibility for specifying transactional behaviour. We do this by first developing a detailed formalism which describes this transactional behaviour. Next, this formalism is adapted such that it fits in the *DiCons* specification language.

To show the usefulness of the language, we prove some properties of specifications. Furthermore, a methodology is given for the testing of Internet applications. This methodology is based on transition systems that for instance could be generated by

DiCons specifications. Using an implementation of the algorithm for the testing of Internet applications, some applications are tested. It is also explained how Internet applications can be implemented using current technology, and how a specification can be turned into executable code. The compiler that is developed for the first version of *DiCons* is shortly discussed and the extensions that should be added to be able to turn specifications described in this thesis into executable code are mentioned.

Samenvatting

Tegenwoordig vinden er meer en meer activiteiten plaats via het internet. Dit betekent dat internet-applicaties steeds belangrijker worden. Omdat deze applicaties ook in omvang en complexiteit groter worden, wordt het specificeren en implementeren van zulke applicaties steeds lastiger. Er komen veel kwesties de hoek om kijken, zoals sessie-beheer en het correct implementeren van het parallel gebruik door meerdere cliënten. Onze mening is dat formele methoden kunnen helpen deze problemen aan te pakken.

In dit proefschrift wordt een nieuwe specificatietaal, *DiCons*, geïntroduceerd, die gebruikt kan worden voor het specificeren van een speciale groep van internet-applicaties, de zogenaamde gedistribueerde consensus applicaties. Dit zijn applicaties die door een groep van cliënten gebruikt kan worden om een gemeenschappelijk doel te bereiken. Hierbij hoeven gebruikers elkaar niet fysiek te ontmoeten en vindt alle communicatie asynchroon plaats. Voorbeelden van zulke applicaties zijn het trekken van Sinterklaaslootjes, een stemming, een veiling, maar ook het plannen van een vergadering.

We bekijken eerst de verschillen tussen op het web gebaseerde en "gewone" window-gebaseerde applicaties. Vervolgens bekijken we de verschillen en overeenkomsten tussen enkele voorbeeldapplicaties en we kijken welke risico's het gebruik van web-applicaties met zich meebrengt. Daaruit abstraheren we concepten die voor alle applicaties in het voor ons interessante domein van belang zijn.

Met deze concepten in ons achterhoofd ontwikkelen we een op proces-algebra gebaseerde formele specificatietaal. De taal bevat naast de standaardoperatoren voor sequentiële en alternatieve compositie drie conditionele operatoren: de conditionele keuze, de conditionele herhaling en de conditionele onderbreking. We leggen uit hoe toestanden en tijd aan de taal toegevoegd worden en welke communicatieprimitieven we gebruiken voor het modelleren van communicatie via het internet tussen gebruikers en de applicatie zelf. Daarnaast voegen we de mogelijkheid voor het specifieren van transactioneel gedrag toe. Dit doen we door eerst een formalisme in detail uit te werken dat dit transactioneel gedrag beschrijft. Dit formalisme wordt vervolgens aangepast zodat het in de *DiCons* specificatietaal past.

Om de toepasbaarheid van de taal aan te tonen worden enkele eigenschappen van specificaties bewezen. Daarnaast wordt een methodiek gegeven voor het testen van internet-applicaties. Deze methodiek is gebaseerd op transitiesystemen die onder andere met behulp van *DiCons* specificaties beschreven kunnen worden. Met behulp van een implementatie van het algoritme voor het testen van internet-applicaties worden enkele applicaties getest. Ook wordt uitgelegd hoe een internet-applicatie geïmplementeerd kan worden met behulp van de momenteel gangbare technieken, en hoe een specificatie omgezet kan worden in uitvoerbare programmacode. Er wordt kort ingegaan op de compiler die voor een eerste versie van *DiCons* geïmplementeerd is en er wordt aangegeven welke uitbreidingen nodig zijn om specificaties van de in dit proefschrift beschreven taal om te zetten in programmacode.

Curriculum Vitae

Harm van Beek was born on the 7th of November 1975 in Westerhoven, Noord-Brabant, The Netherlands.

In 1995 he received his atheneum diploma from the Hertog-Jan College in Valkenswaard.

From 1995 till 2000 he studied computer science at the Department of Mathematics and Computer Science of the Technische Universiteit Eindhoven in Eindhoven. He received his master's degree in August 2000 (cum laude) after finishing his master's thesis, titled 'Internet Protocols for Distributed Consensus – the *DiCons* Language'. This work served as a basis for starting a Ph.D. program.

Subsequently, he got a part-time position as a Ph.D. student at the Formal Methods group of the Department of Mathematics and Computer Science of the Technische Universiteit Eindhoven. His work, which focused on the development of a formalism for specifying Internet applications, took place at the (Eindhoven) Embedded Systems Institute. The research led to several publications and to this Ph.D. thesis.

Next to his work on this thesis, he continued running ISAAC, a company started in 1998 which focuses on the development of Internet applications with complex server-side behaviour.

Bibliography

[ABBC99] D.L. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: a domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, May 1999. Special Section: Domain-Specific Languages (DSL).

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services, version 1.1. BEA, IBM, Microsoft, SAP AG and Siebel Systems, available at `http://www.siebel.com/bpel`, May 2003.

[AD76] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *2nd International Conference on Software Engineering*, pages 562–570, San Francisco, CA, USA, 1976.

[AH96] V. Atluri and W. Huang. An authorization model for workflows. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 25–47, Rome, Italy, September 1996.

[AP91] G. Arango and R. Prieto-Díaz. Domain analysis concepts and research directions. In R. Prieto-Díaz and G. Arango, editors, *Domain Analysis and Software Systems Modeling*, pages 9–32, Los Angeles, CA, USA, 1991. IEEE Computer Society Press.

[Bad79] D.Z. Badal. Correctness of concurrency control and implications for distributed databases. In *Proceedings of the IEEE COMPSAC 79*, Chicago, USA, November 1979.

[BAL+90] E. Brinksma, R. Alderden, J. Langerak, R. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. In *Second International Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990.

[BB88] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78:205–245, 1988.

[BBM01a] J.C.M. Baeten, H.M.A. van Beek, and S. Mauw. An MSC based representation of *DiCons*. In *Proceedings of the 10th SDL Forum*, volume 2078 of *Lecture Notes in Computer Science*, pages 328–347, Copenhagen, Denmark, June 2001. Springer-Verlag.

[BBM01b] J.C.M. Baeten, H.M.A. van Beek, and S. Mauw. Operational semantics of *DiCons*, a formal language for developing Internet applications. CS-Report 01/12, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven, October 2001.

[BBM01c] J.C.M. Baeten, H.M.A. van Beek, and S. Mauw. Specifying Internet applications with *DiCons*. In *Proceedings of the 16th ACM Symposium on Applied Computing (SAC 2001)*, pages 576–584, Las Vegas, Nevada, USA, March 2001.

[BC95] K. Bharat and L. Cardelli. Distributed applications in a multimedia setting. In *Proceedings of the First International Workshop on Hypermedia Design*, pages 185–192, Montpellier, France, 1995.

[Bee00] H.M.A. van Beek. Internet protocols for distributed consensus – the *DiCons* language. Master's thesis, Technische Universiteit Eindhoven, August 2000.

[Bee02] H.M.A. van Beek. An algebraic approach to transactional processes. CS-Report 02/18, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven, December 2002.

[Ber91] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer-Verlag, 1991.

[BFG02] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *Proceedings of the 11th international world wide web conference (WWW2002)*, Honolulu, Hawaii, USA, May 2002.

[BFP01] J.A. Bergstra, W.J. Fokkink, and A. Ponse. Process algebra with recursive operations. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 333–389. Elsevier, 2001.

[BFV+99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12th *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[BG81] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[BHM+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture. W3C Working Group Note, `http://www.w3.org/TR/ws-arch`, February 2004.

[BK82] J.A. Bergstra and J.W. Klop. Fixed point semantics in process algebras. Report IW 206, Mathematisch Centrum, Amsterdam, 1982.

[BK84a] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Automata, Languages and Programming, 11th Colloquium*, volume 172 of *Lecture Notes in Computer Science*, pages 82–94, Antwerp, Belgium, July 1984. Springer-Verlag.

[BK84b] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[BK02] V. Bos and J.J.T. Kleijn. *Formal specification and analysis of industrial systems*. PhD thesis, Technische Universiteit Eindhoven, 2002.

[BKP92] F.S. de Boer, J.W. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In A. Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 137–147, Santa Cruz, CA, June 1992. IEEE Computer Society Press.

[BKT85] J.A. Bergstra, J.W. Klop, and J.V. Tucker. Process algebra with asynchronous communication mechanisms. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Proceedings of the Seminar on Concurrency*, number 197 in Lecture Notes in Computer Science, pages 76–95. Springer-Verlag, 1985.

[Bla05] Blauw Research. Online thuiswinkelen naar recordomzet van 1,7 miljard euro. Press release, available at `http://www.blauw.nl/`, March 2005.

[BLM02] R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in join calculus. In L. Brim, P. Jancar, M. Kretinsky, and A. Kucera, editors, *Proceedings of CONCUR 2002*, volume 2421 of *Lecture Notes in Computer*

Science, pages 321–335, Brno, Czech Republic, August 2002. Springer-Verlag.

[BM01] R. Bruni and U. Montanari. Zero-safe net models for transactions in Linda. In U. Montanari and V. Sassone, editors, *Proceedings of ConCoord 2001, International Workshop on Concurrency and Coordination*, volume 54 of *Electronic Notes in Theoretical Computer Science*, Lipari Island, Italy, August 2001. Elsevier Science.

[BM03] H.M.A. van Beek and S. Mauw. Automatic conformance testing of Internet applications. In A. Petrenko and A. Ulrich, editors, *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, volume 2931 of *Lecture Notes in Computer Science*, pages 205–222, Montreal, Canada, October 2003. Springer-Verlag.

[BM04] P.V. Biron and A. Malhotra. XML schema part 2: Datatypes second edition. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0/`, October 2004.

[BMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). Internet RFC 1738, December 1994.

[BMS02] C. Brabrand, A. Møller, and M.I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology (TOIT)*, 2(2):79–114, May 2002.

[BPW94] J.A. Bergstra, A. Ponse, and J.J. van Wamel. Process algebra with backtracking. In J. W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *REX Workshop*, number 803 in Lecture Notes in Computer Science, pages 46–91, Noordwijkerhout, The Netherlands, 1994. Springer-Verlag.

[Bra89] R.T. Braden. Requirements for Internet hosts – communication layers. Internet RFC 1122, October 1989.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. *Protocol Specification, Testing and Verification VI, IFIP 1987*, pages 349–360, 1987.

[BT00] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, pages 44–50, Nantes, July 2000.

[BV93] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *Proceedings of the International Conference on Concurrency Theory – CONCUR'93*, number

715 in Lecture Notes in Computer Science, pages 477–492, Hildesheim, Germany, 1993. Springer-Verlag.

[BV95] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 149–268. Oxford University Press, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[Car94] L. Cardelli. Obliq: a language with distributed scope. SRC Research Report 122, Digital Equipment, June 1994.

[CD99] L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, May 1999.

[Che99] S. Cheung. *Java Transaction Service (JTS)*. Sun Microsystems, Inc., December 1999.

[CLM+02] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C Recommendation, `http://www.w3.org/TR/P3P`, April 2002.

[CM96] M. Cortes and P. Mishra. DCWPL: a programming language for describing collaborative work. In *Proceedings of ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, Language Support for Groupware, pages 21–29, 1996.

[CMS03] A.S. Christensen, A. Møller, and M.I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.

[Col02] Coldbeans Software. Vote servlet. `http://www.servletsuite.com/servlets/vote.htm`, 2002.

[Col03] CollabNet, Inc. MaxQ. `http://maxq.tigris.org/`, 1999–2003.

[DEW97] R.B. Doorenbos, O. Etzioni, and D.S. Weld. A scalable comparison-shopping agent for the world-wide web. In W. Lewis Johnson and B. Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 39–48, Marina del Rey, CA, USA, February 1997. ACM Press.

[Die01] Dieselpoint, Inc. dieseltest. `http://www.dieseltest.com/`, 2001.

[DKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[DP92] P. Degano and C. Priami. Proved trees. In W. Kuich, editor, *Proceedings ICALP'92*, number 623 in Lecture Notes in Computer Science, pages 629–640, Vienna, 1992. Springer-Verlag.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. On the notions of consistency and predicate locks in a data base system. *Communications of the ACM*, 19(11), November 1976. Also published in/as: IBM, Res.R. RJ1487, San Jose, CA, December 1974.

[Eur99] European Computer Manufacturers Association. ECMAScript Language Specification. Standard ECMA–262, third edition, ISO/IEC 16262, December 1999.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet RFC 2616, June 1999.

[FGR04] W.F. Fokkink, J.F. Groote, and M.A. Reniers. Process algebra needs proof methodology. *Bulletin of the EATCS*, 82:108–125, February 2004. Also appeared as Computer Science Report 04/04, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2004.

[FKK96] A.O. Freier, P. Kariton, and P.C. Kocher. The SSL protocol: Version 3.0. Internet draft, Netscape Communications, 1996. Available at `http://www.netscape.com/eng/ssl3/`.

[Fla98] D. Flanagan. *JavaScript Definitive Guide*. O'Reilly, July 1998.

[Fok94] W.J. Fokkink. The tyft/tyxt format reduces to tree rules. In M. Hagiya and J.C. Mitchell, editors, *Proc. 2nd Symposium on Theoretical Aspects of Computer Software – TACS'94*, number 789 in Lecture Notes in Computer Science, pages 440–453, Sendai, April 1994. Springer-Verlag.

[Ful02] J. Fulmer. Siege. `http://www.joedog.org/siege/`, 2002.

[FW04] D.C. Fallside and P. Walmsley. XML schema part 0: Primer second edition. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0/`, October 2004.

[Gar95] S. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, Inc., Newton, MA, USA, 1995.

[GHM+03a] M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, and H.F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, `http://www.w3.org/TR/soap12-part1`, June 2003.

[GHM+03b] M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, and H.F. Nielsen. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation, `http://www.w3.org/TR/soap12-part2`, June 2003.

[GLW02] P. Gardner, C. Laneve, and L. Wischik. The fusion machine (extended abstract). In L. Brim, P. Jančar, M. Křetinský, and A. Kučera, editors, *CONCUR 2002: Concurrency Theory (13th International Conference)*, volume 2421 of *LNCS*, pages 418–433, Brno, Czech Republic, August 2002. Springer.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra81] J. Gray. The transaction concept: Virtues and limitations. In *International Conference On Very Large Data Bases (VLDB '81)*, pages 144–154, Los Angeles, CA, USA, September 1981. IEEE Computer Society Press.

[Hee98] L. Heerink. *Ins and outs in refusal testing*. PhD thesis, University of Twente, The Netherlands, 1998.

[HHJ+87] C.A.R. Hoare, I.J. Hayes, H. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Surfin. Laws of programming. In *Communications of the ACM, 30(8)*, pages 672–686, 1987.

[HHK+03] H. Haas, O. Hurley, A. Karmarkar, J. Mischkinsky, M. Jones, L. Thompson, and R. Martin. SOAP Version 1.2 Specification Assertions and Test Collection. W3C Recommendation, `http://www.w3.org/TR/soap12-testcollection`, June 2003.

[HL02] M. Howard and D. LeBlanc. *Writing Secure Code, Second Edition*. MicroSoft Press, 2002.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[Int86] International Organization for Standardization. Information processing – text and office systems – Standard Generalized Markup Language (SGML). ISO 8879, December 1986.

[ISO96] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. Information retrieval, transfer and management for OSI; framework: Formal methods in conformance testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500, ISO – ITU-T, Geneva, 1996.

[IT00] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC2000)*. ITU-TS, Geneva, 2000.

[JL02] X. Jia and H. Liu. Rigorous and automatic testing of web applications. In *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 280–285, Cambridge, MA, USA, November 2002.

[Jon93] D. Jonscher. Extending access control with duties – realized by active mechanisms. In B. Thuraisingham and C.E. Landwehr, editors, *Database Security VI: Status and Prospects*, pages 91–111, North-Holland, 1993.

[KB94] Bharat K. and M.H. Brown. Building distributed, multi-user applications by direct manipulation. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Groupware and 3D Tools, pages 71–81, 1994.

[KBR+04] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web services choreography description language (WS-CDL), version 1.0. W3C Working Draft, `http://www.w3.org/TR/ws-cdl-10`, December 2004.

[KM98] T. Kistler and H. Marais. WebL – a programming language for the Web. *Computer Networks and ISDN Systems*, 30(1–7):259–270, April 1998.

[LCP03] P. Lomax, M. Childs, and R. Petrusha. *VBScript in a Nutshell, 2nd Edition*. O'Reilly, April 2003.

[Ley01] F. Leymann. Web Services Flow Language (WSFL 1.0). IBM, available at `http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, May 2001.

[LLW+04] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (DOM) level 3 core specification. W3C Recommendation, `http://www.w3.org/TR/DOM-Level-3-Core`, April 2004.

[LM98] D. Li and R.R. Muntz. COCA: Collaborative objects coordination architecture. In *Proceedings of ACM CSCW'98 Conference on Computer-Supported Cooperative Work*, Infrastructures for Collaboration, pages 179–188, 1998.

[LR95] D.A. Ladd and J.C. Ramming. Programming the web: An application-oriented language for hypermedia service programming. In *Proceedings of the 4th WWW Conference, WWW Consortium*, pages 567–586, 1995.

[LWM98] D. Li, Z. Wang, and R. Muntz. Building online auctions from the perspective of COCA. Technical report, UCLA Department of Computer Science, September 1998.

[Mac05] Macromedia, Inc. ColdFusion MX. `http://www.macromedia.com/software/coldfusion/`, 1995–2005.

[MB01] S. Mauw and V. Bos. Drawing message sequence charts with LaTeX. *TUGboat*, 22(1/2):87–92, March 2001.

[Mea55] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, September 1955.

[Mic05] Microsoft Corporation. ASP.NET Web: The official Microsoft ASP.NET site. Available at `http://www.asp.net/`, 2003–2005.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[Min05] Miniwatts International, Inc. Internet world stats – usage and population statistics. `http://www.internetworldstats.com/`, 2001–2005.

[Mit03] N. Mitra. SOAP Version 1.2 Part 0: Primer. W3C Recommendation, `http://www.w3.org/TR/soap12-part0`, June 2003.

[Mos81] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Computing*. PhD thesis, MIT, 1981.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992.

[MRW01] S. Mauw, M.A. Reniers, and T.A.C.. Willemse. Message sequence charts in the software engineering process. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, pages 437–463. World Scientific Publishing Co., 2001.

[MWW04] S. Mauw, W.T. Wiersma, and T.A.C. Willemse. Language-driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14(6):1–39, 2004.

[Neu05] P.G. Neumann. The risk digest, forum on risks to the public in computers and related systems. `http://catless.ncl.ac.uk/Risks/`, 1985–2005. ACM Committee on Computers and Public Policy.

[NMS02] O. Niese, T. Margaria, and B. Steffen. Automated functional testing of web-based applications. In *Proceedings of the 5th Int. Conference On Software and Internet Quality Week Europe (QWE2002)*, Brussels, Belgium, March 2002.

[O'B99] L. O'Brien. Vox populi. *Java Pro Magazine*, June 1999.

[Obj05] Object Mentor, Inc. JUnit, testing resources for extreme programming. `http://www.junit.org/`, 2001–2005.

[OSGL03] The Original Software Group Ltd. TestWEB. `http://www.testweb.com/`, 2003.

[Pet80] C.A. Petri. Introduction to general net theory. In W. Brauer, editor, *Net theory and applications: Proceedings of the advanced course on general net theory, processes and systems*, volume 84 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1980.

[PHP05] The PHP Group. PHP: Hypertext Preprocessor. `http://www.php.net/`, 2001–2005.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Denmark, 1981. Published in 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 1977. IEEE Computer Society Press.

[RC04] D. Robinson and K. Coar. The Common Gateway Interface (CGI) version 1.1. Internet RFC 2875, October 2004.

[RFPG96] J. Rice, A. Farquhar, P. Piernot, and T. Gruber. Using the web instead of a window system. In *Human Factors in Computing Systems, CHI'96 Conference Proceedings*, pages 103–110, Vancouver, B.C, Canada, 1996.

[RLHJ99] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 specification. W3C Recommendation, `http://www.w3.org/TR/html401`, December 1999.

[Rog01] J. Rogers. *Microsoft Jscript.Net Programming*. Macmillan Computer Pub, December 2001.

[RT01] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 25–34, Toronto, Ontario, Canada, May 2001. IEEE Computer Society.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[SGMS94] K. Salem, H. García-Molina, and J. Shands. Altruistic locking. *ACM Transactions on Database Systems*, 19(1):117–165, March 1994.

[Sit05] SiteOption. Auction Engine. `http://www.siteoption.com/AuctionEngine.cfm`, 1995–2005.

[SM77] R.M. Shapiro and R.E. Millstein. Reliability and fault recovery in distributed processing. In *OCEANS'77, Conference Record*, volume II, pages 31D.1–31D.5, Los Angeles, CA, USA, October 1977.

[SUN05a] SUN Microsystems, Inc. Java Servlet Technology. `http://java.sun.com/products/servlet/`, 1994–2005.

[SUN05b] SUN Microsystems, Inc. Java Transaction API (JTA). `http://java.sun.com/products/jta/`, 1994–2005.

[SUN05c] SUN Microsystems, Inc. JavaServer Pages Technology. `http://java.sun.com/products/jsp/`, 1994–2005.

[TBMM04] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures second edition. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0/`, October 2004.

[Tha01] S. Thatte. XLANG, web services for business process design. Microsoft Corporation, 2001.

[Thi02] P. Thiemann. WASH/CGI: Server-side Web scripting with sessions and typed, compositional forms. In S. Krishnamurthi and C.R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages: 4th International Symposium*, volume 2257 of *Lecture Notes in Computer Science*, pages 192–208, Portland, OR, USA, January 2002. Springer-Verlag.

[Tho97] R. Thomas. Team-based access control (TMAC): A primitive for applying role-based access controls in collaborative environments. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, pages 13–19, Fairfax, Virginia, USA, November 1997.

[Tre94] J. Tretmans. A formal approach to conformance testing. In O. Rafiq, editor, *International Workshop on Protocol Test Systems VI*, volume C-19 of *IFIP Transactions*, pages 257–276. North-Holland, 1994.

[Tre95] J. Tretmans. Testing labelled transition systems with inputs and outputs. In A. Cavalli and S. Budkowski, editors, *Participants Proceedings of the Int. Workshop on Protocol Test Systems VIII – COST 247 Session*, pages 461–476, Evry, France, September 1995.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[TS97] R. Thomas and R. Sandhu. Task-based authorization controls (TBAC): Models for active and enterprise-oriented authorization management. In *Proceedings of the 11th IFIP Working Conference on Database Security*, pages 136–151, Lake Tahoe, California, USA, August 1997.

[Ude99] J. Udell. *Practical Internet Groupware*. O'Reilly & Associates, Inc., October 1999.

[Ver97] J.J. Vereijken. *Discrete-Time Process Algebra*. PhD thesis, Technische Universiteit Eindhoven, 1997.

[Vra97] J.L.M. Vrancken. The algebra of communicating processes with empty process. *Theoretical Computer Science*, 177(2):187–328, May 1997.

[VS+03] S. Viswanadha, S. Sankar, et al. Java Compiler Compiler [tm] (JavaCC [tm]) – The Java Parser Generator. `http://javacc.dev.java.net/`, 1999–2003.

[Wal98] N. Walsh. A technical introduction to XML. In *World Wide Web Journal*, October 1998.

[Wei86] G. Weikum. A theoretical foundation of multilevel concurrency control. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 31–42, Cambridge, Massachusetts, March 1986.

[WLB99] H. Wium Lie and B. Bos. Cascading style sheets, level 1. W3C Recommendation, `http://www.w3.org/TR/CSS1`, January 1999.

[WO02] Y. Wu and J. Offutt. Modeling and testing web-based applications. ISE Technical ISE-TR-02-08, GMU, November 2002.

[WS92] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Transaction Models for Advanced Database Applications*. Morgan Kaufmann, February 1992.

[YBP+04] F. Yergeau, T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (third edition). W3C Recommendation, `http://www.w3.org/TR/2004/REC-xml`, February 2004.

Index

A

access control 30
  DiCons 44
  formalisation of 72
  role-based 30
  team-based 31
ACID properties *37*, 80
action
  internal 117
  lockable 85
  locking 112, *118*
  unlocking 85, *118*
action function 61
Active Server Pages 3, *189*
adapter 151
alphabet 49
alternating structure 155
alternative composition . . . . . *see* composition, alternative
API . . . . . *see* application programming interface
application programming interface 190
ASP . . . . . *see* Active Server Pages
associativity
  of alternative composition 51
  of sequential composition 51
atomicity *37*, 80
authorisation control
  task-based 32
axioms
  for conditional branching 54
  for conditional disrupt 56
  for conditional repetition 55
  for deadlock 52
  for locking 89
  for parallel composition 90
  for the empty process 53
  for transactional composition 86
  for unlocking 89
  of BPA 51

B

Basic Process Algebra 49
blocking 138
  of a process 139
  of a user 140
BPA . . . . . *see* Basic Process Algebra
browse 111

C

Cascading Style Sheets 3, *183*
CGI . . . . . *see* Common Gateway Interface
chaos 111
client identification . . . . . *see* identification, client
ColdFusion 189
commit 81
Common Gateway Interface 189
communication primitives . . . . . *see* interaction primitives
commutativity
  of alternative composition 51
composition
  alternative 50
  sequential 50
  transactional 83
concept 33
  fixed 11
  irrelevant 11
  variable 11
conditional branching 54
conditional disrupt 56
conditional repetition 54
conformance 152
  relation 158
congruence 100
consistency *37*, 80
CSS . . . . . *see* Cascading Style Sheets
cursor stability 111

D

deadlock 52
  absence of 99
deduction rules
  for $T(\mathrm{BPA}_{\delta\varepsilon})$ 53, 65
  for alternative composition 53, 65, *125*
  for anonymous replication 74, *132*
  for conditional branching 55, 65, *126*
  for conditional disrupt 56, 65, *126*
  for conditional repetition 55, 65, *126*
  for extended generalised parallel composition 77, *135*
  for extended replication 77, *133*
  for generalised parallel composition 76, *135*
  for interaction primitives 71, *124*
  for internal actions *123*
  for locking 90, *130*
  for parallel composition 73, 91, *131*
  for replication 75, *133*
  for scope operator 62, 64, *127*
  for sequential composition 53, 65, *125*
  for the empty process 53, 65, *123*
  for the time step 64, *123*
  for transactional composition 88, *128*
  for unlocking 90, *130*
denial of service 29
derivably equal 52
DiCons *4*, 115
dirty data 81
distributed consensus 4
Document Object Model 204
Document Type Definition 203
DOM . . . . . *see* Document Object Model
domain analysis 9
domain identification 11
domain-specific language 9
driver 151
DSL . . . . . *see* domain-specific language
DTD . . . . . *see* Document Type Definition
durability *38*, 80

E

ECMAScript 187
effect function 61, 105
elevation of privilege 29
empty process 52
evaluation 59
evaluation function 59
Extensible Markup Language 201
Extensible Style sheet Language 203

F

fail 152
formal methods 3

G

group 72
  registered 76

H

HTML . . . . . *see* Hypertext Markup Language
HTTP . . . . . *see* Hypertext Transfer Protocol
hyperlink 182
hypertext 182
Hypertext Markup Language 3, 15, 38, *183*
  tags 183
Hypertext Transfer Protocol *14*, 17, 38, 182, 192
  GET request 193
  POST request 194

I

idempotency
  of alternative composition 51
identification 34
  client 42
identifier intersection 103
if-then-else-fi 54
implementation relation 152
inconclusive 152
independent responsiveness 141
information disclosure 28
instance 35
interaction
  active server push *40*, 67
  anonymous 74
  reactive server pull *41*, 68
  reactive server push *40*, 67
  session-oriented server pull *41*, 68
  session-oriented server push *41*, 68
interaction primitives *40*, 118
IOTS . . . . . *see* transition system, input-output
isolated 112
isolation *37*, 80
  degrees of 111

J

Java Compiler Compiler 200
Java Server Pages 3, *190*
JavaCC . . . . . *see* Java Compiler Compiler

JavaScript 187
JScript 187
JSP . . . . . *see* Java Server Pages

L

level of abstraction 38
lock
  exclusive 112
  read 81
  shared 112
  write 81
lock counter 85
lost updates 81
LTS . . . . . *see* transition system, labelled

M

Mealy machine 156
message sequence chart 18, 43
MIOTS . . . *see* transition system, multi input-output
MRRTS . . . . . *see* transition system, multi request-response
MSC . . . . . *see* message sequence chart

O

operator 50
  alternative composition *50*, 119
  anonymous replication *74*, 120
  bang . . . . . *see* operator, replication
  choice . . . . . *see* operator, alternative composition
  conditional branching *54*, 119
  conditional disrupt *56*, 120
  conditional repetition *54*, 120
  extended bang*see* operator, extended replication
  extended generalised merge . . . . . *see* operator, extended generalised parallel composition
  extended generalised parallel composition . . *76*, 120
  extended replication *76*, 120
  extended transactional composition 121
  generalised merge . . . . . *see* operator, generalised parallel composition
  generalised parallel composition *75*, 120
  left-merge 89
  locking *88*, 121
  merge . . . . . 89, *see* operator, parallel composition
  parallel composition *73*, 120
  precedence scheme 51
  replication *75*, 120
  scope *60*, 120
  sequential composition *50*, 119
  transactional composition *83*, 121
  unlocking *88*, 121

P

partitioning 155
pass 152
path format 100
PGP . . . . . *see* Pretty Good Privacy
PHP . . . . . *see* PHP Hypertext Preprocessor
PHP Hypertext Preprocessor 3
pragmatics 12
Pretty Good Privacy 29
primer 151
problem domain 11
problem instance 36
problem space 11
process 35
  empty . . . . . *see* empty process
  regular 101
process algebra 35, *49*
process term *49*, 117
program counter *36*, 57
projection 155
protocol
  connection-less 14
  stateless 14

R

RBAC . . . . . *see* access control, role-based
registered group . . . . . *see* group, registered
registration 42, *76*
regular process . . . . . *see* process, regular
repeatable reads . . . . . *see* isolated
repudiation 28
request label 156
response
  enabled 163
  expected 157
response label 156
right distributivity
  of sequential composition 51
rollback 81
RRTS . . . . . *see* transition system, request-response

S

Secure Socket Layers 29
sequential composition *see* composition, sequential
serialisable . . . . . *see* isolated
servlet 190
session 17, 35, *40*, 155

implementation of . . . 194
label . . . 73
session validity . . . 143
SGML *see* Standard Generalized Markup Language
SOS . . . *see* Structural Operational Semantics
specification . . . 151
spoofing identity . . . 27
SQL . . . *see* Structured Query Language
SSL . . . *see* Secure Socket Layers
Standard Generalized Markup Language . . . 203
state . . . 57, *59*
initial . . . 154
space . . . 34, 36
STRIDE model . . . 27
Structural Operational Semantics . . . 53
Structured Query Language . . . 189
substitution . . . 60, 104

T

tampering with data . . . 27
TBAC . . . *see* authorisation control, task-based
termination
successful . . . *52*, 117
unsuccessful . . . 117
test
case . . . 152, 158, *158*
derivation . . . 158, *160*, 162
algorithm . . . *160*, 162
execution . . . *151*, 158
hypothesis . . . 152
suite . . . 152, *158*
complete . . . 159
exhaustive . . . 159
sound . . . 159
testing
batch-wise . . . 164
black-box . . . *150*, 153
conformance . . . 150, *152*
dynamic . . . 150
on-the-fly . . . 164
thin client . . . *16*, 150
time . . . 45, *63*, 116
TMAC . . . *see* access control, team-based
trace . . . 138
transaction . . . 36, *80*
nesting of . . . 80
transactional composition . . . *see* composition, transactional
transfer condition . . . 100
transition . . . 121
label . . . 121
relation . . . 154
transition system
input/output . . . 156
labelled . . . *154*, 169
without rollback transitions . . . 170
multi input/output . . . 156
multi request/response . . . *156*, 171
request/response . . . 156
two-phase . . . 111
type . . . 116
type determination . . . 60

U

Uniform Resource Locator . . . 192
universe
of groups . . . *72*, 116
of identifiers . . . 58
of input parameters . . . *68*, 116
of messages . . . *68*, 116
of output parameters . . . *68*, 116
of session labels . . . 116
of states . . . 116
of types . . . 58
of users . . . *68*, 116
of valuations . . . 116
unlocking set . . . 85
unwrapping . . . 171
updates function . . . 106
URL . . . *see* Uniform Resource Locator
user . . . 42
anonymous . . . 34, *42*

V

valuation . . . 58
uninitialised . . . 59
valuation set . . . 103
valuation stack . . . 59
VBScript . . . 187
verdict . . . 152

W

WAM . . . *see* workflow authorisation model
well-formed . . . 111
while-do-od . . . 54
workflow authorisation model . . . 32

X

XML . . . *see* Extensible Markup Language
XML Schemas . . . 203
XSL . . . *see* Extensible Style sheet Language

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into*

Acute Effects of Air Pollution Episodes. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-*. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05