Formal specification and analysis of industrial systems

Formal specification and analysis of industrial systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 7 maart 2002 om 15.00 uur

door

Victor Bos
geboren te Apeldoorn

en

Jeroen Johannes Theodorus Kleijn
geboren te Goirle

Dit proefschrift is goedgekeurd door de promotoren van drs. V. Bos:
prof.dr. J.C.M. Baeten
en
prof.dr.ir. J.E. Rooda

Copromotor:
dr. S. Mauw

Dit proefschrift is goedgekeurd door de promotoren van ir. J.J.T. Kleijn:
prof.dr.ir. J.E. Rooda
en
prof.dr. J.C.M. Baeten

Copromotor:
dr.ir. J.M. van de Mortel-Fronczak

Print: Universiteitsdrukkerij, Technische Universiteit Eindhoven

IPA Dissertation Series 2002-02.

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

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Bos, Victor and Kleijn, Jeroen J.T.

Formal specification and analysis of industrial systems / by Victor Bos and Jeroen J.T. Kleijn. - Eindhoven : Technische Universiteit Eindhoven, 2002.
Proefschrift. - ISBN 90-386-2743-2
NUGI 857
Subject headings: systems engineering; discrete event systems / industrial systems; formal methods / process algebra / industrial systems; specification / industrial systems; verification / industrial systems; simulation / modelling language

To understand the basic issues in the behaviour of concurrent systems, it is helpful to have a simple language "*with as few operators or combinators as possible, each of which embodies some distinct and intuitive idea, and which together give completely general expressive power*" (R. Milner [142, p. 264]). The pictograms on the cover show to what extent we succeeded in defining language constructs with a distinct and intuitive idea.

Contents

Preface

This thesis is the result of four years of intensive cooperation at the Eindhoven University of Technology: cooperation between the Mechanical Engineering Department and the Mathematics and Computer Science Department; cooperation between the Systems Engineering group and the Formal Methods group; and most of all, cooperation between the authors. Cooperation between the two groups existed for years already, and by the start of the research project called *Algebraic methods for analysing industrial production processes* it really took shape. This project offered us Ph.D. positions and aimed at specification and analysis of industrial systems using formal methods. This kills two birds with one stone. On the one hand, it can improve specification and analysis of industrial systems. On the other hand, it can provide feedback on the suitability of formal methods and broaden their application domain.

We would like to thank the following persons. Firstly, we thank our supervisors, prof.dr. J.C.M. Baeten and prof.dr.ir. J.E. Rooda, and our co-supervisors, dr. S. Mauw and dr.ir. J.M. van de Mortel-Fronczak, for the opportunities given and the support offered. Secondly, we thank the members of our committee, prof.dr. W.J. Fokkink, prof.dr.ir. C.A. Middelburg, prof.dr. H. Nijmeijer, and prof.dr. M. Rem, for reading this thesis. They provided many constructive remarks, for which we are very grateful. Thirdly, we thank some persons in particular for being a sparring-partner or providing valuable comments: dr. S.C.C. Blom, dr. J.H. Geuvers, prof.dr.ir. J.F. Groote, dr.ir. A.T. Hofkamp, and dr.ir. M.A. Reniers. Fourthly, we thank our colleagues of the Systems Engineering group and the Formal Methods group for contributing to a pleasant work atmosphere. In this respect, we also thank EESI. Finally, we thank our families and friends for their support.

All research presented in this thesis is joint work. In particular, we want to remark that all presented lemmas are results of equal efforts of both authors. Together, we wrote Chapters 1, 2, 4, 5, and 9 and Appendix B. Victor Bos wrote Chapters 6 and 7 and Appendix C. Jeroen Kleijn wrote Chapters 3 and 8 and Appendix A.

Summary

The complexity of modern industrial systems increases, the amount of money involved increases, and competition on the market gets stronger. Consequently, the impact of design errors increases. Modelling industrial systems before they are built enables engineers to reduce the number of design errors. In addition, products change faster, new products are developed faster and so must the systems needed to produce them. Consequently, (new) industrial systems have to be realised within shorter time frames. Therefore, industry makes high demands on methods used for modelling industrial systems.

Typical system properties that determine the success or failure of industrial systems are *throughput*, *cycle time*, (absence of) *deadlock*, and *livelock*. Modelling techniques should enable analysis of such properties. The class of system properties can be divided into two subclasses: *performance* properties and *functional* properties. Throughput and cycle time belong to the first class, deadlock and livelock belong to the second class.

Simulation is a powerful technique for performance analysis. By simulating models of industrial systems, it is possible to calculate statistically significant approximations of, for instance, the throughput and cycle time. To that end, the Systems Engineering Group of the Eindhoven University of Technology has developed a specification language. The language is called χ and together with its simulators it has been used in many case studies.

For functional analysis, however, simulation is less suitable. Simulation can be used to show that a (model of a) system has deadlock, but it is, in general, not possible to show that a system is deadlock-free. Furthermore, simulation cannot be used to detect that a (model of a) system has livelock or not.

Formal methods, on the other hand, do provide opportunities for functional analysis. Formal methods are mathematical notations and techniques that can be used to prove correctness of a system by mathematical proof. Usually, a formal

method consists of a formal specification language and several techniques (some of which may be automatic) to prove properties about specifications written in that language. Using formal methods, we are able to prove a system is, or is not, deadlock-free and livelock-free.

Now, consider the following observation: from a computer science point of view, models defined in a specification language like χ are just programs and formal methods have been developed mainly to analyse such programs. Hence, by proving properties of programs, we prove properties of models, and, provided that those models are accurate, properties of real life systems.

Following this observation, first we provided a formal *syntax* and *semantics* for χ. A formal syntax explicitly defines the syntactic structure of a language and a formal semantics gives mathematical meaning to the defined language constructs. By giving a formal syntax and semantics, we obtain a mathematical framework that enables calculation.

Formalisation of χ resulted in a language where artificial restrictions have been abandoned and language constructs have become mathematical operators. Furthermore, a notion of equivalence (*bisimulation*) has been defined. Using this notion, we are able to derive general equalities for χ processes and operators. Together with an introduced notion of *abstraction*, we can also verify whether an *implementation* satisfies its *specification*.

Second, we developed *tools*. A formal framework as described above enables *manual* verification of χ models. Unfortunately, manual verification is quite laborious and it requires a solid background in logic and formal reasoning. In practise, especially for models of production systems, many systems will soon be too large to verify by hand. Tool support is then indispensable.

With respect to performance analysis, a simulator was built that works exactly according to the defined semantics (whereas the existing simulator implements an intuitive interpretation of the semantics).

With respect to functional analysis, newly developed tools have been combined with existing tools from the formal methods community in order to enable *model checking*. Model checking can be described as *exhaustive simulation*. Instead of simulating an certain subset of the behaviours of a model, all behaviours of the model are simulated. Consequently, if such an exhaustive simulation does not find violations of a particular property, we can conclude with mathematical certainty

that the model *satisfies* the property. We are able to do model checking on a reasonably large subset of χ models.

Third, case studies have been conducted in order to test the developed mathematical framework and tools. They ranged from small toy examples to performance and functional analysis of real life systems. They show that a combination of simulation and verification improves substantially the analysing power of χ.

Samenvatting

De complexiteit van moderne industriële systemen neemt toe, de hoeveelheid geld die daarmee gemoeid gaat neemt toe, en de concurrentie op de markt wordt sterker. Dientengevolge hebben ontwerpfouten grotere consequenties. Het modelleren van industriële systemen alvorens zij worden gebouwd biedt ingenieurs de mogelijkheid het aantal ontwerpfouten te reduceren. Daarbij komt dat producten sneller veranderen, en dat nieuwe producten sneller worden ontwikkeld. Dit geldt tevens voor de systemen die deze producten produceren. Zodoende moeten ook (nieuwe) fabrieken en machines in een korter tijdsbestek kunnen worden gerealiseerd. Dientengevolge stelt de industrie hoge eisen aan de methoden voor het modelleren van industriële systemen.

Typische systeemeigenschappen die het succes of falen van een industriëel systeem bepalen zijn *doorzet*, *doorlooptijd*, '*deadlock*', en '*livelock*'. Modelleertechnieken zouden analyse van deze eigenschappen mogelijk moeten maken. De klasse van systeemeigenschappen kan worden onderverdeeld in twee subklassen: *prestatie-eigenschappen* en *functionaliteitseigenschappen*. Doorzet en doorlooptijd behoren tot de eerste klasse, 'deadlock' en 'livelock' behoren tot de tweede klasse.

Simulatie is een krachtige techniek voor het uitvoeren van een prestatie-analyse. Door modellen van industriële systemen te simuleren, is het mogelijk statistisch significante benaderingen van bijvoorbeeld doorzet en doorlooptijd te berekenen. Daartoe heeft de sectie Systems Engineering van de Technische Universiteit Eindhoven een specificatietaal ontwikkeld. Deze taal heet χ en samen met haar simulatoren is ze al veelvuldig gebruikt in casestudies.

Voor functionaliteitsanalyse daarentegen, is simulatie minder geschikt. Simulatie kan gebruikt worden om aan te tonen dat een model 'deadlock' bevat, maar het is in het algemeen niet mogelijk om aan te tonen dat een model 'deadlock'-vrij is. Tevens kan simulatie niet worden gebruikt om te detecteren of een (model van een) systeem 'livelock' heeft of niet.

Formele methoden, daarentegen, voorzien wel in mogelijkheden voor functionaliteitsanalyse. Formele methoden zijn wiskundige notaties en technieken die gebruikt kunnen worden om correctheid van systemen aan te tonen door middel van wiskundige bewijsvoering. Doorgaans bestaat een formele methode uit een formele specificatietaal en verschillende technieken (waarvan sommige geautomatiseerd kunnen zijn) om eigenschappen te bewijzen van specificaties opgesteld in die taal. Gebruikmaking van formele methoden stelt ons in staat te bewijzen dat een systeem wel of niet 'deadlock'- en 'livelock'-vrij is.

Beschouw nu de volgende observatie: bezien vanuit een informatica oogpunt zijn modellen, gedefinieerd in een specificatietaal als χ, gewoon programma's, en formele methoden zijn ontwikkeld om dergelijke programma's te analyseren. Dientengevolge, door eigenschappen van programma's te bewijzen, bewijzen we eigenschappen van modellen, en, vooropgesteld dat deze modellen accuraat zijn, eigenschappen van 'real-life' systemen.

In navolging van deze observatie, hebben we ten eerste χ van een formele *syntax* en *semantiek* voorzien. Een formele syntax definieert expliciet de syntactische structuur van een taal, en een formele semantiek geeft een wiskundige betekenis aan de gedefinieerde taalconstructies. Door het geven van een formele syntax en semantiek wordt een wiskundig raamwerk verkregen waarmee gerekend kan worden.

De formalisering van χ resulteerde in een taal waaruit kunstmatige restricties zijn verwijderd en taalconstructies wiskundige operatoren zijn geworden. Tevens werd een notie van gelijkheid (*bisimulatie*) gedefinieerd. Hiermee kunnen we algemene gelijkheden afleiden voor χ processen en operatoren. Samen met een geïntroduceerde notie van *abstractie* kunnen we tevens verifiëren of een *implementatie* voldoet aan zijn *specificatie*.

Ten tweede hebben we *gereedschappen* ontwikkeld. Een formeel raamwerk zoals hierboven omschreven maakt *handmatige* verificatie van χ modellen mogelijk. Helaas is handmatige verificatie nogal bewerkelijk en vereist het een solide achtergrond in logica en formeel redeneren. In de praktijk, en dan in het bijzonder voor modellen van productiesystemen, zullen veel systemen al snel te groot zijn om handmatig te verifiëren. Ondersteuning in de vorm van gereedschappen is dan onmisbaar.

Wat betreft prestatie-analyse, werd een simulator gebouwd die exact volgens de gedefinieerde semantiek werkt (waar de bestaande simulator een intuïtieve interpretatie van de semantiek implementeert).

Wat betreft functionaliteitsanalyse, werden nieuw ontwikkelde gereedschappen gecombineerd met bestaande gereedschappen uit de formele methoden gemeenschap om zodoende '*model checking*' mogelijk te maken. 'Model checking' kan worden omschreven als uitputtende simulatie. In plaats van het simuleren van een zekere subset van de gedragingen van een model, worden alle gedragingen van het model gesimuleerd. Dientengevolge, als een dergelijke uitputtende simulatie geen schendingen van een bepaalde eigenschap vindt, kunnen we met wiskundige zekerheid concluderen dat het model voldoet aan de eigenschap. Het is mogelijk een redelijk grote subset van χ modellen te 'model checken'.

Ten derde zijn casestudies uitgevoerd om het ontwikkelde wiskundige raamwerk en de ontwikkelde gereedschappen te testen. Deze casestudies variëren van kleine voorbeeldjes tot prestatie- en functionaliteitsanalyses van 'real-life' systemen. Zij laten zien dat een combinatie van simulatie en verificatie de analysekracht van χ substantieel verbetert.

Introduction · 1

Industrial systems produce and/or process products. Examples of such systems are factories, machines, and warehousing systems. Designing modern industrial systems is an increasingly complicated task. Several causes can be given for this increase in complexity. For instance, higher demands are made on production processes due to increasing product diversity and/or product innovation. Also, competition on the market gets stronger due to globalization. Besides increasing the complexity of industrial systems, these causes also require (new) industrial systems to be realised within shorter time frames.

Therefore, industry makes high demands on methods and techniques used for modelling industrial systems. The goal of these methods and techniques is to reduce the number of design errors by showing the design has desirable system properties. Typical system properties that determine the success or failure of industrial systems are *throughput* (the number of products per hour), *cycle time* (the time a product spends in a system), *deadlock* (the inability to proceed at all), and *livelock* (the inability to proceed sensibly).

1.1 · Modelling industrial systems

By Systems Engineering refer to the research field that investigates and develops methods, techniques, and tools to design advanced industrial systems. Due to the high demands on these methods and techniques, as mentioned above, a shift from qualitative approaches to quantitative approaches can be observed. This shift is particularly visible in the way industrial systems are modelled. Three kinds of models can be distinguished [118]: physical (for example scale models), graphical (for example engineering drawings), and symbolic (for example formal specifications). The shift from qualitative approaches to quantitative approaches

is confirmed by an increase of symbolic modelling techniques, because they are well suited for quantitative analysis.

Often, symbolic models are written in a specification language. Clearly, it is important to choose a suitable specification language. Whether a language is suitable depends on the application at hand. Hence, languages come in different flavours.

Specification languages can be subdivided into three categories: *continuous* languages, *discrete event* languages, and combined, also called *hybrid*, languages. Continuous languages are used for modelling continuous (physical) systems and processes. Usually, continuous behaviour is defined by differential equations. An example of a continuous language is ACSL (Advanced Continuous Simulation Language) [145]. Discrete event languages are used to describe discrete event behaviour of (physical) systems and processes. Examples of such languages are SIMAN (Simulation Analysis) [160], the DEVS (Discrete Event System) formalism [196, 195], and μDemos [34, 35] (a sugared version of Demos (Discrete Event Modelling On Simula) [33]). Finally, languages exist that combine continuous and discrete event features into one formalism. Examples of such languages are COSMOS [117, 116], Dymola (Dynamic Modelling Laboratory) [67, 52], gPROMS (general Process Modelling System) [21], Modelica [135], Omola (Object-oriented Modelling Language) [6], Personal Prosim [186], and χ [26, 175, 8].

The main purpose of the languages mentioned above, and of languages for industrial systems modelling in general, is to understand the dynamic behaviour of systems by means of modelling and simulation. Simulation is a powerful tool when it comes to analysing system properties. We distinguish *performance* analysis and *functional* analysis. Typically, performance analysis concerns models of complete production facilities, and usually focuses on properties like throughput and cycle time. Functional analysis typically concerns models of single machines and their control systems. It usually focuses on specification-implementation checks and deadlock and livelock detection. Traditionally, performance analysis and functional analysis were research topics studied by different communities. However, nowadays people acknowledge that these two forms of analysis should be studied together [46].

Although simulation turned out to be a successful approach for performance analysis, with respect to functional analysis the approach has some disadvantages. Firstly, if the language is nondeterministic, simulation does not provide information about all possible behaviours of a system. That is, simulation can show the

presence of errors, but cannot show the *absence* of errors. As a consequence, the absence of deadlock cannot be guaranteed. Secondly, simulation requires assignment of concrete values to system parameters such as production rates, buffer sizes, operating times of machines, and the amount of work in process. As a consequence, simulations have to be repeated when there is a change in the system parameters.

In the field of Formal Methods, specification and analysis of functional properties of programs is often done using mathematical techniques called *formal methods*. These techniques, as opposed to simulation techniques, enable analysis of all possible behaviours of the system. Since our objective is to improve specification and analysis techniques for industrial systems, it is interesting to consider the following observation. Models of industrial systems defined in a simulation language are just programs. Hence, we can use formal techniques to specify and analyse functional properties of models of industrial systems. Provided those models are accurate, functional properties of real-life industrial systems can be analysed. This leads to the following research topic.

Research topic 1 ▪ *Is it possible to improve specification and analysis techniques for industrial systems by means of formal methods?*

1.2 ▪ Formal methods

Formal methods are mathematical notations and techniques designed to establish correctness of a system by mathematical proof. Application of formal methods eliminates much ambiguity and enables designers to have a consistent and objective understanding of a system. On the other hand, application of formal methods is laborious and requires thorough knowledge of mathematics. Therefore, formal methods are not suitable for all types of application.

A particular class of systems for which formal methods are beneficial is called *industrial critical systems* [81, 80, 180] (sometimes called *mission* critical). Examples of industrial critical systems are medical systems, traffic regulation systems, electronic payment systems, and wafer steppers. For these systems, failures can have serious consequences. By applying formal methods, the number of failures due to design errors can be reduced.

A formal method usually consists of a formal specification language, a mathematical framework, and tool support. A formal specification language has a formal

syntax and *semantics*. A formal syntax explicitly defines the syntactic structure of a language and a formal semantics gives mathematical meaning to all language constructs. The mathematical framework enables calculation similar to calculation in ordinary arithmetic. In theory, such a framework suffices to perform verification. That is, one can now perform *manual* verification. Unfortunately, manual verification is quite laborious and it requires a solid background in logic and formal reasoning. In practice, especially for models of production systems, the system under consideration will soon be too large to be verified by hand. Tool support is then indispensable. Tool support ranges from interactive administrative tools, which check proof steps and keep track of proof obligations, to completely automated tools, which establish proofs themselves. The latter provide facilities for *automatic* verification.

Research in Formal Methods resulted in methods like ACP (Algebra of Communicating Processes) [19, 18, 71], Algebraic Theory of Processes [101], CCS (Calculus of Communicating Systems) [143, 141], CSP (Communicating Sequential Processes) [106, 47, 105], automata theory [131], and Petri Nets [166, 162]. In addition, examples of methods based on ACP are LOTOS (Language of Temporal Ordering Specification) [190, 66], μCRL (micro Common Representation Language) [90, 93, 92], and (the formal semantics of) MSC (Message Sequence Charts) [111, 167]. An example based on CCS is the Edinburgh CWB (Concurrency Work Bench) [147]. An example based on CSP is SPIN [109, 108] with input language PROMELA (Process Meta Language) [31, 193, 154]. Examples based on automata theory are Kronos [194, 62], HyTech (the Hybrid Technology tool) [103, 5], and UPPAAL (from Uppsala, Sweden and Aalborg, Denmark) [126]. An example based on Petri Nets is ExSpect (Executable Specification Tool) [100]. Finally, there are many methods based on general formal logics. Examples are Esterel [30, 29], SDL (Specification and Description Language) [146], VDM (Vienna Development Method) [113], Z [60], PVS (Prototype Verification System) [181, 158, 185], STeP (Stanford Temporal Prover) [134, 133], NuSMV [53], a re-implementation and extension of SMV (Symbolic Model Verifier), and HOL (Higher Order Logic) [86].

As explained above, two strands in verification can be distinguished: manual verification and automatic verification. The latter is based on logics for which decision procedures exist. For these procedures to exist, one has to compromise on the expressive power of the logics. Therefore, the logics of automatic verification techniques usually have less expressive power than the logics of manual verification techniques.

Usually, manual verification techniques are, from a theoretical point of view, more powerful than automatic verification techniques. This does not mean, however, that manual verification techniques are more suited to analyse real-life industrial systems. The problem with manual verification is that, due to its formal nature, it requires the user to pay attention to a lot of details, which entails a substantial administrative problem. Furthermore, one needs a solid background in logic and formal reasoning in order to establish correct formal proofs. Therefore, applying manual techniques in large projects is a complicated task that needs a team of highly educated engineers.

Automatic verification tries to tackle this problem by using less expressive logics for which decision procedures can be implemented. The most widely used logics are *temporal logics* [164]. In temporal logics one can describe properties of the behaviour of a system over time. Usually, decidability results are established for such temporal logics, which means that effective algorithms (algorithms that always produce an answer in finite time) can be developed to check if a certain property, expressed as a temporal logic formula, holds in a particular state, in some states, or in all states. The idea is to have decision procedures deal with the details of the verification automatically, while still maintaining a logic with reasonable expressive power to describe properties of the model. Ideally, automatic verification should be a 'push-button' technique for which no special background in formal logic and mathematics is required. The problem remains, however, that informal descriptions of properties have to be translated into formal descriptions. This requires understanding of the mathematical framework of the automatic verification technique and the skill to validate whether a temporal logic formula is a correct translation of the informal property to be checked.

Of course, the distinction between manual and automatic verification is not as clear cut as depicted here. Moreover, attempts to integrate manual techniques with automatic techniques [165], have resulted in interesting combinations. For instance, there are manual verification techniques with a lot of tool support to systematically deal with most of the details of a verification task.

An automatic verification technique that found its way into many industrial applications is *model checking* [55]. Model checking can be described as *exhaustive simulation*. That is, instead of simulating an arbitrary subset of the behaviours of a model, *all* behaviours of the model are simulated. Consequently, if such an exhaustive simulation does not find violations of a particular property, we can conclude, with mathematical certainty, that the model *satisfies* the property. Ef-

fectively, an exhaustive simulation is an exploration of the complete *state space* of a model. Since the state space of a parallel system grows exponentially in the number of parallel processes, such an exploration is only feasible for relatively small models. This is called the *state space explosion* problem and it renders automatic verification of large real-life industrial systems infeasible.

Although the state space explosion problem seems a fundamental problem of model checking, several techniques, like partial order reduction, symbolic model checking, and symmetry reduction [44, 23, 50, 161, 83, 82, 48], have been developed to tackle the state space explosion problem. Using these techniques, it is possible to analyse real-life systems of considerable size.

The previous section explained that system properties can be divided into performance properties and functional properties. Simulation is a powerful technique to analyse performance properties, but it is much less powerful with respect to functional properties. Above, we explained that formal methods are useful for functional analysis. Integration of simulation techniques with formal methods techniques will enable us to analyse both performance properties and functional properties. This leads to the following research topic.

Research topic 2 · *Is it possible to integrate formal methods with existing simulation techniques?*

1.3 · Formal specification and analysis of industrial systems

As described above, both Systems Engineering and Formal Methods make use of specification languages. For the sake of discussion, we will call the languages developed and used in the first field *engineering languages* and languages developed and used by the second field *formal languages*. We can conceive of the following alternatives to integrate simulation techniques with formal methods:

1. translate an engineering language into a formal language,
2. use a formal language to analyse industrial systems, or
3. formalise an engineering language.

The first alternative has the advantage that after the translation, all theory and tools of the concerning formal language are readily available. This makes it an alternative with which results can be achieved rapidly. Another advantage is

that for many engineering languages, formal languages exist with more or less the same constructs. Therefore, the translation is usually not very complicated. In [119, 120], a χ specification of an industrial system is translated into μCRL and verified using the *Focus Points and Convergent Process Operators* proof technique [95, 94]. In [188], μDemos is translated into CCS, thereby enabling formal verification with the Edinburgh CWB. In [40], we defined a translation scheme from χ into PROMELA. We applied this translation scheme to verify a χ specification of a production system with the SPIN model checker. However, there are several disadvantages to this alternative. First of all, the language to specify a system is different from the language to analyse that system. Therefore, the systems engineer has to master both languages and he should be able to switch between these languages repeatedly, since designing a system is an iterative process. Another problem is that the translation of the engineering language into the formal language is generally not complete. That is, only a subset of the engineering language can be translated. The reason is that some constructs of the engineering language do not have equivalent constructs in the formal language. Such constructs have to be encoded and that complicates the translation substantially.

The second alternative also has the advantage of making the theory and tools of the concerning formal language readily available for specification and analysis of industrial systems. Another advantage of this alternative is that the system engineer specifies and analyses his systems in the same language, as opposed to the first alternative. In some engineering areas, like electronic circuit design [49, 65, 144] and protocol design [171, 170, 128, 112, 54, 108], this approach has resulted in numerous successes. However, there is a disadvantage to this alternative; the systems engineer has to stop using his 'familiar' engineering language and to start using a 'strange' formal language. The success of this alternative largely depends on the willingness of the systems engineer to switch.

The third alternative tries to circumvent the disadvantage of the second alternative by focusing from the start on the application domain: the field of systems engineering. By formalising an existing engineering language, techniques from the field of Formal Methods become available to systems engineers using their own notations and techniques. The advantages are that new (formal) techniques can be merged with existing (simulation) techniques. The disadvantage is that more work needs to be done before formal techniques can be used. First of all, a particular systems engineering language needs to be formalised. After that, theory should be developed for the new formal language. Finally, tools should be developed, based on this theory, that support formal analysis of systems described in this formal

language. However, this disadvantage is probably not as big as it seems. Firstly, the formalisation is a one time job. Secondly, techniques to describe existing formal languages (for instance, algebraic specification techniques and operational semantics techniques) are likely to be useful to formalise engineering languages as well. Thirdly, if such particular techniques can be used, they provide indications to build a mathematical framework for the new formal language. Namely, this framework should address the same issues as the frameworks of the existing formal languages that were described with these techniques in the first place. Finally, if the mathematical framework of the new formal language is similar to that of existing formal languages, support tools do not have to be designed from scratch. Instead, their design and implementation can be based on the same (proven) principles as the designs and implementations of existing tools. Furthermore, these tools can be integrated with language independent tools. Examples of such tools are FCTOOLS using the *fc2* format [168] for labelled transition systems, the CADP tools (Cæsar Aldébaran Development Package) [75] using the BCG (Binary Coded Graphs) format [197] for labelled transition systems, the μCRL tools [91] using the SVC format (named after the Systems Validation Centre Project) for labelled transition systems [125], and the Graphviz tools (graph visualisation) [74] using the *dot* format [124] to represent graphs.

Considering the advantages and the disadvantages of the three alternatives, we think that in the long run, the third alternative is the most promising to bridge the gap between Systems Engineering and Formal Methods. Therefore, we decided to formalise an engineering language, to develop a mathematical framework for this language, and to implement tools to support formal analysis with this language.

As mentioned in Section 1.1, there are many engineering languages to specify and analyse industrial systems. In order to give the work presented in this thesis more practical relevance, it is necessary to select a good representative. Looking at the languages actually used by engineers, we see that they have more or less the same expressive power. This is not surprising since they have the same application domain. In Section 1.1, we subdivided engineering languages into three categories: continuous languages, discrete event languages, and hybrid languages. A good representative is likely to be found in the category of hybrid languages because they feature most aspects of engineering languages [26]. We mentioned the languages COSMOS, Dymola, gPROMS, Modelica, Omola, Personal Prosim, and χ. Of these languages, we chose the χ language for the following reasons. First of all, χ is a language that resulted from years of experience with other simulation languages and libraries [157, 176, 177, 173, 174, 172]. The developers of χ felt

that neither of these languages and libraries provided the right notation and tools needed to model industrial systems efficiently. Therefore, in the early nineties, they decided to develop their own modeling language. This resulted in the first version of χ [8, 150]. A recent positioning of χ with respect to other engineering languages is given in [26]. Second, from its earliest development on, developers of χ have recognized the importance of mathematical reasoning; something which is uncommon for many engineering languages. Third, χ is inspired by theoretical languages and notations, like CSP and the guarded command language [63], and χ has symbols whereas other languages usually have keywords. Finally, χ has been applied successfully in numerous case studies, some of which are listed below. For a global overview see `http://se.wtb.tue.nl/posters/`.

- *Discrete (re-entrant) flow lines*: design of a multi-process multi-product wafer fab [51, 97, 179] (Philips), balancing of a car assembly line [88] (Volvo/Mitsubishi)
- *Hybrid (jumbled) flow* [25]: design of a fruit juice blending and packaging plant [69] (Riedel)
- *Control architectures*: agent-based control of systems [151, 153], flexible manufacturing systems [152]
- *Machinery*: scheduling algorithms for medical equipment [149] (TNO)

The basis of χ is called *discrete* χ. This is a discrete event simulation language with probabilistic constructs. In [39], we formalised a small subset of discrete χ and in [2], a simulator for this language is discussed. Extensions of discrete χ in different directions exist. For example, in [68], a simulator for *hybrid* χ is discussed. This is a hybrid simulation language in which both discrete event processes and continuous processes can be specified and analysed. In addition, in [107], discrete χ and its tool support is extended to enable real-time control.

Summarizing this section, we explained three alternatives to integrate simulation techniques with formal methods. We selected the third alternative: formalise an engineering language. In particular, we chose the engineering language χ. The goal of this formalisation is not just to provide a formal semantics of the language, but to develop a mathematical framework that enables calculation with χ models and to develop tools that support this calculation. In addition, the resulting framework should be powerful enough to analyse real-life systems. This essentially means

turning χ into a practical formal method. This leads to the following research topic.

Research topic 3 · *Is it possible to convert χ into a formal method?*

1.4 · Approach

In order to convert χ into a practical formal method, we need to define a formal semantics of χ, develop a mathematical framework, develop tools, and perform case studies.

The formalisation of χ resulted in a new version of χ together with a corresponding mathematical framework (see Chapters 2, 3 and 4). This new version, we call χ_σ. The subscript σ refers to the Greek word $\sigma\eta\mu\alpha\sigma\iota o\lambda o\gamma\acute{\iota}\alpha$ which in Latin characters is spelled sèmasiologia and translates to the word 'semantics'. The language χ_σ resembles χ very closely (see Chapter 5) and we define a translation scheme to translate a χ specification into a χ_σ specification (see Chapter 6). Eventually, χ_σ should replace χ.

The tools we developed enable simulation and model checking of χ_σ specifications (see Chapter 7). They are integrated with existing tools to manipulate and visualize state spaces.

We analysed real-life industrial systems with the formal method χ_σ (see Chapter 8). These case studies show that industrial systems can be specified in χ_σ. Furthermore, they show that χ_σ enables performance analysis, similar to χ, as well as functional analysis.

As mentioned in the previous section, extensions of χ exist for hybrid and real-time systems. Currently, these extensions are the subject of active research and they are likely to undergo substantial changes. Therefore, this thesis focuses on discrete χ. The most up-to-date reference of the discrete χ version formalised in this thesis is the on-line document [121]. In Chapter 3, a brief description of this version of χ is given. In the remainder of this thesis, we use the name χ to denote discrete χ, unless explicitly stated otherwise.

We restrict the formalisation of χ by disregarding the probabilistic language constructs. Incorporating such constructs in a (formal) language is a research project of its own. Future research should address this topic.

1.5 ▪ Overview

Chapter 2 describes the formal semantics of the data types of χ and χ_σ. In Chapter 3, we introduce the specification language χ, and Chapter 4 defines the specification language χ_σ. The relation between χ and χ_σ is described in Chapter 5. In order to translate a χ specification into a χ_σ specification, we define a translation scheme in Chapter 6. In Chapter 7, we describe the tools we developed for χ_σ, and in Chapter 8, several examples and case studies are discussed. Finally, in Chapter 9, we draw conclusions and discuss opportunities for further research.

Data types · 2

This chapter discusses the data types of χ and χ_σ. We introduce operators and functions on these data types that are used frequently in models of industrial systems (see Chapter 8). Most of the data types are defined by means of *Algebraic Specification* (AS) [9, 27, 136, 156]. In particular, we use *Membership Equational Logic* (MEL) [138, 43]. The syntax and semantics of MEL as we used it, is described in Appendix C. The decision to use MEL is based on the fact that it is among the most powerful AS formalisms and allows natural specification. We do remark, though, that other AS formalisms with the concept of subsorts, such as for example, *order sorted algebra* [84, 85], would probably be equally useful.

Some χ data types are not defined in MEL. They are introduced at the end of this chapter. The semantics of χ_σ as presented in Chapter 4, however, only uses data types defined by means of AS.

This chapter is organised as follows. Section 2.1 gives a brief introduction to MEL, and the concepts of MEL are illustrated by an example. The basic data types of χ_σ are defined by MEL *specifications* of booleans (Section 2.2), natural numbers (Section 2.3), integer numbers (Section 2.4), rational numbers (Section 2.5), and by a specification *scheme* of channels (Section 2.6). Section 2.7 introduces MEL *theories*. The element theory given in this section, defines requirements on elements of generic data types. The relation between specifications and theories is defined by MEL *views*. A view defines a mapping between a theory and a specification or another theory. In Section 2.8, we define views from the element theory to the basic data types. The generic data types of χ_σ are sets (Section 2.9), lists (Section 2.10), and tuples (Section 2.11). Generic data types can be instantiated with other (basic or generic) data types using views. In Section 2.12, we define views from the element theory to the generic data types. The χ data types that are not formalised are discussed in Section 2.13. This chapter is concluded by a discussion in Section 2.14.

2.1 · Introduction to MEL

Consider the following specification of the booleans. Note that this is an example; the χ_σ booleans are defined in Section 2.2.

spec BOOL-EXAMPLE
sort bool.
constructors
  $true :\rightarrow$ bool,
  $false :\rightarrow$ bool.
operator
  $\_ \Rightarrow \_ :$ bool bool $\rightarrow$ bool.
var b : bool.
equations
[BX1] $b \Rightarrow true = true$,
[BX2] $false \Rightarrow b = true$,
[BX3] $true \Rightarrow false = false$.
end

The specification starts with the keyword **spec** and ends with the keyword **end**. Keywords are printed in bold type. This specification is called BOOL-EXAMPLE. The specification is divided in *sections*, each starting with a keyword. In the **sort** section, a sort name, bool, is defined. Sort names correspond to types in programming languages. In the **constructors** section, two *constructor* operators are defined: *true* and *false*. Note that section keywords can appear in singular and plural form. The **operator** section defines the boolean implication operator '⇒' with type bool bool → bool. The underscores in the specification indicate the argument positions relative to the operator. The type defines the sorts of the arguments and the sort of the result. That is, '⇒' is an infix operator that takes two terms (see Appendix C, Definition C.6) of sort bool as arguments and the result is also a term of sort bool. There is no semantical difference between constructors and other operators in MEL, see Definition C.3, page 280, and Section C.1, page 279. Below, we will explain why we call certain operators constructors. For now, it suffices to know that the specifications are written in such a way that all terms containing normal operators should be reducible to terms containing constructors only. For the boolean example this means every closed boolean term should be reducible to *true* or *false*. A closed (boolean) term is a term without variables. Reducing a term means rewriting it according to the *derivation* rules of MEL,

see Definition C.11 on page 283. Equations are defined in the **equations** section. Each equation has a label, for example, the first equation in BOOL-EXAMPLE has label [BX1]. Following the label is an equality between two terms, for example, $b \Rightarrow true = true$ states that $b \Rightarrow true$ is equal to *true*. The terms are built up from constructors, operators, and variables. The variables are defined in the **var** section. In the example, there is one variable, b, of sort bool.

In a signature all constants and operator symbols (including their argument and result types) are defined, see Definition C.4 on page 280. The *signature* of a MEL specification is defined by its **sort**, **constructor**, and **operator** sections. For example, the signature of BOOL-EXAMPLE consists of the sort bool, the constructors *true* and *false* and the operator '⇒' (including their argument and result types).

The MEL specifications we present in this chapter have been validated by extensive testing. That is, we used an AS tool, called Maude [57, 56, 138, 58], to implement and test the MEL specifications. The fact that MEL is supported, is the main reason we chose Maude instead of similar systems, like, Elan [37], ASF+SDF [45], and CAFE [73]. Maude reduces terms by rewriting them according to the equations of a specification. Furthermore, Maude applies equations from left to right only; it interprets equations as *rewrite rules*. Thus, a specification induces a *Term Rewrite System* (TRS) [122]. If, at a certain point in a computation, no rewrite rule can be applied, the computation has *terminated* and the concerning term is, by definition, a *normal form*. A computation can be *terminating* or *nonterminating*. If every computation in a TRS is terminating, the TRS is terminating. Vice versa, if there exist nonterminating computations, the TRS is nonterminating. Notice that if Maude applied equations from right to left, too, every computation would be nonterminating.

Interpreting equations as rewrite rules does not guarantee a terminating TRS. Since Maude produces results only for terminating computations, we have to show that nonterminating computations cannot occur. In order to guarantee a specification induces a *terminating* TRS, we distinguish between constructors and (normal) operators. By calling certain operators constructors and by defining the remaining operators in terms of these constructors, it is clear that every computation will result in a term consisting of constructors only. Such terms are called *constructor terms*. Consequently, if there exist nonterminating computations, there exist nonterminating computations on constructor terms only. So, to prove the TRSs induced by the MEL specifications are terminating, there are two proof obliga-

tions. Firstly, operators have to be defined in terms of constructors. Secondly, computations on constructor terms have to be terminating.

In addition to terminating computations, a desirable property of TRSs is *confluence*. A TRS is confluent if whenever two (different) computations start from the same term, it is always possible to continue the computations such that they end up with the same term. This means choices between applicable rewrite rules are irrelevant. This property is sometimes called the *Church-Rosser* property. If a terminating TRS is not confluent, there is at least one term from which computations start leading to different normal forms. On the other hand, if a terminating TRS is confluent, every term has a unique normal form. Many AS tools, like Maude, are built on the assumption that the TRSs are confluent and terminating. This assumption enables an effective decision procedure to determine whether two terms are equal. Namely, reduce both terms to normal form and compare them.

We do not give a formal proof that shows the TRSs induced by the MEL specifications in this chapter are terminating and confluent. The benefit of such a proof would be that the TRSs can be used to perform computations on the data types. However, our goal is to specify the data types and not to provide implementations. Therefore, we confined ourselves to checking the termination obligations mentioned above and by testing the MEL specifications in Maude.

The semantics, or interpretation, of a specification is an *algebra*, see Definition C.12 on page 284. The interpretation of a sort name is a set and the interpretation of an operator is a function. In an algebra, terms and equations have an *interpretation*, see Definition C.14 on page 285. If (the interpretation of) all equations hold(s) in a particular algebra, the algebra is called a *model* of the specification, see Definition C.15 on page 285. For example, an algebra for BOOL-EXAMPLE is $A = (\{S\}, \{S\}, \{\bot :\rightarrow S, \top :\rightarrow S, imp : S \times S \rightarrow S\})$, where $S = \{0, 1\}$, $\bot() = 0$, $\top() = 1$, and imp is defined by

$$imp(x, y) = \begin{cases} 0 & \text{if } x = 1 \text{ and } y = 0 \\ 1 & \text{otherwise.} \end{cases}$$

The algebra has a set S containing 0 and 1, and three functions: $\bot$, $\top$, and imp. Note that $\bot$ and $\top$ are nullary functions; application of these functions is denoted by $\bot()$ and $\top()$, respectively. In order to see that A is an algebra for BOOL-EXAMPLE, we define the following mapping (interpretation) of syntactic entities to semantical entities:

$$\text{bool} \mapsto S,\ \mathit{true} \mapsto \top,\ \mathit{false} \mapsto \bot,\ \Rightarrow \mapsto \mathit{imp}.$$

It is clear that under this mapping, the equations of the specification hold. Therefore, A is a model for BOOL-EXAMPLE.

Note that A is not the only algebra for BOOL-EXAMPLE. For example, if we take $S' = \{2, 3\}$ and replace 0 by 3 and 1 by 2 in the definition of $\bot$, $\top$, and *imp*, we get an algebra A' and $A \neq A'$. Of course, the difference between A and A' is cosmetic, since the structure of A is similar to that of A'. Formally, A and A' are *isomorphic* (see the discussion below Definition C.16 on page 286).

However, there are algebras for BOOL-EXAMPLE that are not isomorphic to A. For example, we could take $S'' = \{0, 1, 2\}$ and define an algebra A'' with the same definitions for $\bot$, $\top$, and *imp* as for A. Note that the equations hold in A''. Algebra A'' is not isomorphic to A, since the set S'' has three elements whereas S has only two elements. Therefore, there can be no homomorphism between A and A'' that is both surjective and injective. Element 2 of S'' is superfluous; it is not used as a function result. In a similar way, we could devise algebras for the boolean specification by adding superfluous functions to A. Such algebras are said to have *junk*; they have elements or functions for which no syntactic term exists in the specification.

Maybe more interesting is the algebra $A''' = (\{S'''\}, \{S'''\}, \{\bot :\rightarrow S, \top :\rightarrow S, imp : S \times S \rightarrow S\})$, where $S''' = \{0\}$. Note that in this case *imp* has to be defined by $imp(x, y) = 0$, since there is only one element in S'''. Similarly, both $\bot()$ and $\top()$ should yield 0. Using these definitions of $\bot$, $\top$, and *imp*, the same mapping we used for A can be used for A'''. Note that the equations of BOOL-EXAMPLE hold in A''' and therefore it is an algebra for the specification. In the interpretation A''' of BOOL-EXAMPLE, the constructors *true* and *false* are mapped onto the same semantical object 0. Therefore, in A''', the meaning of *true* is equal to the meaning of *false*. However, the specification does not tell us that *true* = *false*, since there is no way to derive this equality by applying the equations of BOOL-EXAMPLE. Algebras such as A''' are said to have *confusion*; they identify elements that cannot be identified by the specification.

Usually, we are only interested in algebras without junk and confusion. Therefore, we restrict the class of possible algebras for a specification to those that do not have superfluous elements or functions and that do not identify elements that cannot be identified in the specification. Algebras without junk and confusion are called *initial* algebras. Furthermore, there is only one initial algebra (up to isomorphism) for each specification. By definition, MEL specifications have an initial (algebra) semantics.

2.2 · Booleans

The MEL specification BOOL defines the booleans and the boolean operators. The **sort** section defines one sort name called bool. The **operators** section defines the boolean operators. First, the two boolean constants *true* and *false* are defined. In addition, the conventional boolean operators '¬' (negation), '∧' (conjunction), '∨' (disjunction), and '⇒' (implication) are defined. The constants are the only constructors of the booleans; closed terms with a boolean operator can be rewritten either to *true* or to *false*. Associativity and commutativity of the conjunction and disjunction operator are expressed by the attributes 'assoc' and 'comm', respectively.

Commutativity and associativity could be formulated by equations, but that would make the term rewrite system non-terminating. Therefore, MEL constructors and operators can have special attributes like 'comm' and 'assoc'. In Appendix C, the formal status of operator attributes is explained. For now, it suffices to say that an equational specification in MEL has two sets of equations. One set is defined implicitly by operator attributes and the other set is defined explicitly by equations. We sometimes call the first set 'attribute equations'.

The **var** section declares three boolean variables, b, b_0, and b_1. The variables are used to define equations. The **equations** section defines equations between boolean terms. It is evident that these equations ensure that any closed boolean term containing an operator ('¬', '∧', '∨', or '⇒'), can be rewritten to one of the forms *true* or *false*. Therefore, BOOL defines a terminating rewrite system.

spec BOOL
sort bool.
constructors
  $true :\rightarrow$ bool,
  $false :\rightarrow$ bool.
operators
  $\neg\_ :$ bool $\rightarrow$ bool,
  $\_ \wedge \_ :$ bool bool $\rightarrow$ bool [comm, assoc],
  $\_ \vee \_ :$ bool bool $\rightarrow$ bool [comm, assoc],
  $\_ \Rightarrow \_ :$ bool bool $\rightarrow$ bool.
var $b, b_0, b_1 :$ bool.
equations
[B1] $\neg true = false$,

[B2] $\neg false = true$,
[B3] $true \wedge b = b$,
[B4] $false \wedge b = false$,
[B5] $b_0 \vee b_1 = \neg(\neg b_0 \wedge \neg b_1)$,
[B6] $b_0 \Rightarrow b_1 = \neg b_0 \vee b_1$.
end

The specification of the booleans is quite simple. Therefore, it is suitable to demonstrate how the equations are used to formally prove the equivalence of two (boolean) terms. For example, consider the terms $(true \Rightarrow false) \Rightarrow true$ and $(true \Rightarrow false) \vee (false \Rightarrow true)$. By applying the equations, both terms can be rewritten to $true$. Therefore, the terms are equal, which is denoted by $\text{BOOL} \vdash (true \Rightarrow false) \Rightarrow true = (true \Rightarrow false) \vee (false \Rightarrow true)$. The formal derivations are:

$$
\begin{aligned}
& (true \Rightarrow false) \Rightarrow true \\
\overset{[B6]}{=}\ & \neg(\neg true \vee false) \vee true \\
\overset{[B1]}{=}\ & \neg(false \vee false) \vee true \\
\overset{[B5]}{=}\ & \neg(\neg(\neg false \wedge \neg false)) \vee true \\
\overset{[B2]}{=}\ & \neg(\neg(true \wedge true)) \vee true \\
\overset{[B3]}{=}\ & \neg(\neg true) \vee true \\
\overset{[B1]}{=}\ & \neg false \vee true \\
\overset{[B2]}{=}\ & true \vee true \\
\overset{[B5]}{=}\ & \neg(\neg true \wedge \neg true) \\
\overset{[B1]}{=}\ & \neg(false \wedge false) \\
\overset{[B4]}{=}\ & \neg false \\
\overset{[B2]}{=}\ & true,
\end{aligned}
$$

$$
\begin{aligned}
& (true \Rightarrow false) \vee (false \Rightarrow true) \\
\overset{[B6]}{=}\ & (\neg true \vee false) \vee (\neg false \vee true) \\
\overset{[B1]}{=}\ & (false \vee false) \vee (true \vee true) \\
\overset{[B5]}{=}\ & \neg(\neg false \wedge \neg false) \vee \neg(\neg true \wedge \neg true) \\
\overset{[B2]}{=}\ & \neg(true \wedge true) \vee \neg(\neg true \wedge \neg true) \\
\overset{[B1]}{=}\ & \neg(true \wedge true) \vee \neg(false \wedge false) \\
\overset{[B3]}{=}\ & \neg true \vee \neg(false \wedge false)
\end{aligned}
$$

$$\begin{aligned}
&\overset{[B4]}{=} \neg true \vee \neg false \\
&\overset{[B1]}{=} false \vee \neg false \\
&\overset{[B2]}{=} false \vee true \\
&\overset{[B5]}{=} \neg(\neg false \wedge \neg true) \\
&\overset{[B2]}{=} \neg(true \wedge \neg true) \\
&\overset{[B1]}{=} \neg(true \wedge false) \\
&\overset{[B3]}{=} \neg false \\
&\overset{[B2]}{=} true.
\end{aligned}$$

The generalised property, $(b \Rightarrow b') \Rightarrow b = (b \Rightarrow b') \vee (b' \Rightarrow b)$, for b and b' arbitrary (possibly open) boolean terms, cannot be proven using the equations given above.

2.3 · Natural numbers

The number specifications are set up hierarchically starting with the natural numbers (this section) and extending these via integer numbers (Section 2.4) to the rational numbers (Section 2.5). For each level of the number hierarchy, a MEL specification is given which defines both constructors and additional operations on numbers. Notice that the hierarchy is not extended to the real numbers, because a MEL specification of the real numbers does not exist (see also 5.4).

Specification NAT below defines the sorts posnat (positive natural numbers) and nat (natural numbers). The positive natural numbers are a *subsort* of the natural numbers. This is defined by the subsort relation posnat < nat.

The constructors of the natural numbers are 0 and s. Every natural number is represented by a term of the form $s\ s\ \ldots\ s\ 0$ where the number of s's is zero or more. Of course, in practice we will use the standard decimal notation for natural numbers.

In the **include** section, other MEL specifications can be included in the current specification. As can be seen, NAT includes specification BOOL of Section 2.2. Consequently, we are allowed to use the sorts, constructors, and operators of the booleans in the specification of the natural numbers, as if they were defined here. The formal status of included specifications is discussed in Section C.3.

First, the predecessor operator $p\_$ and the standard relational operators are defined. Note the use of subsorts in equations. For example, in Equation [N2] $(0 < pn =$

true), *pn* is a variable of sort posnat. That is, *pn* is a positive natural. In order to apply Equation [N2] on a term $n_0 < n_1$, for some natural number terms n_0 and n_1, the normal form of n_0 should be equal to the normal form of 0 and n_1 : posnat should hold. The advantage of using variables of specific (sub)sorts is that less conditions are required in the equations. For example, Equation [N2] reads $0 < pn = true$. If we did not have variables of subsorts, like pn : posnat, a condition should be added to the equation, as in $0 < n = true \Leftarrow n$: posnat.

Note that in some equations, for example, Equation [N5], we use an equality operator on natural numbers that is not defined explicitly in the **operators** section. In fact, we use the equality relation of MEL as if it were a normal binary operation. The soundness of this way of dealing with the equality relation of MEL is explained in [43]. Therefore, we can use an equality operator as if it were defined in MEL. Once we have such an equality operator, it is straightforward to define the corresponding inequality operator '$\neq$'.

Equations [N10] and [N11] define the *difference* operator on natural numbers. It computes the difference between two natural numbers and is a sort of replacement for the subtraction operator. We chose not to define a subtraction, $n_0 - n_1$, on natural numbers, since it requires some awkward definitions for the case where the second argument is greater than the first argument.

Some equations are conditional, for example, Equation [N14]. Only if the condition is true, does the left-hand side of a conditional equation hold. So, according to Equation [N14], if $n < m$ is true (to be determined by applying other equations), then $n \mathsf{div} pn = 0$ is true.

Equations [N17] and [N18] implement Eulers well known algorithm to compute the *gcd* (greatest common divisor) of two positive natural numbers. The *gcd* operation is needed to define unique normal forms of rational numbers in Section 2.5.

Finally, exponentiation is defined by Equations [N19] and [N20]. The first equation says, in conventional notation, $n^0 = 1$. The second equation says, in conventional notation, $n_0^{1+n_1} = n_0 \times n_0^{n_1}$. Using these equations, every term of the form $exp(n_0, n_1)$, where n_0 and n_1 are closed terms, can be rewritten to a (finite) term of the form $n_0 \times (n_0 \times \ldots n_0)$ (with n_1 occurrences of n_0).

spec NAT
include BOOL.
sorts posnat, nat.
subsort posnat < nat.

constructors

$0 :\rightarrow \text{nat}$,

$s\_ : \text{nat} \rightarrow \text{posnat}$.

operators

$p\_ : \text{posnat} \rightarrow \text{nat}$,

$\_ < \_ : \text{nat nat} \rightarrow \text{bool}$,

$\_ \leq \_ : \text{nat nat} \rightarrow \text{bool}$,

$\_ > \_ : \text{nat nat} \rightarrow \text{bool}$,

$\_ \geq \_ : \text{nat nat} \rightarrow \text{bool}$,

$\_ + \_ : \text{nat nat} \rightarrow \text{nat}$ [comm, assoc],

$d : \text{nat nat} \rightarrow \text{nat}$ [comm],

$\_ \times \_ : \text{nat nat} \rightarrow \text{nat}$ [comm, assoc],

$\_ \,\mathsf{div}\, \_ : \text{nat posnat} \rightarrow \text{nat}$,

$\_ \,\mathsf{mod}\, \_ : \text{nat posnat} \rightarrow \text{nat}$,

$gcd : \text{posnat posnat} \rightarrow \text{posnat}$ [comm],

$exp : \text{nat nat} \rightarrow \text{nat}$.

var

$n, n_0, n_1 : \text{nat}$,

$pn, pn_0, pn_1 : \text{posnat}$.

equations

[N1] $p\ s\ n = n$,

[N2] $0 < pn = true$,

[N3] $n < 0 = false$,

[N4] $s\ n_0 < s\ n_1 = n_0 < n_1$,

[N5] $n_0 \leq n_1 = n_0 < n_1 \vee n_0 = n_1$,

[N6] $n_0 > n_1 = n_1 < n_0$,

[N7] $n_0 \geq n_1 = n_1 \leq n_0$,

[N8] $0 + n = n$,

[N9] $(s\ n_0) + n_1 = s\ (n_0 + n_1)$,

[N10] $d(0, n) = n$,

[N11] $d(s\ n_0, s\ n_1) = d(n_0, n_1)$,

[N12] $0 \times n = 0$,

[N13] $(s\ n_0) \times n_1 = n_1 + (n_0 \times n_1)$,

[N14] $n \,\mathsf{div}\, pn = 0 \quad \Leftarrow n < pn$,

[N15] $n \,\mathsf{div}\, pn = s(d(n, pn) \,\mathsf{div}\, pn) \quad \Leftarrow pn \leq n$,

[N16] $n \,\mathsf{mod}\, pn = d(n, (n \,\mathsf{div}\, pn) \times pn)$,

[N17] $gcd(pn_0, pn_1) = pn_0 \quad \Leftarrow pn_0 = pn_1$,

[N18] $gcd(pn_0, pn_1) = gcd(d(pn_0, pn_1), pn_1) \quad \Leftarrow pn_1 < pn_0,$
[N19] $exp(n, 0) = s\ 0,$
[N20] $exp(n_0, s\ n_1) = n_0 \times exp(n_0, n_1).$
end

2.4 • Integer numbers

The integer numbers are an extension of the natural numbers. In the specification INT below, this is reflected by the fact that the sort nat is a subsort of the sort int. Consequently, the nonnegative integer numbers, that is, the natural numbers, are defined already (Section 2.3) and here we only have to add the negative integer numbers.

Specification INT defines two sorts: int (integer numbers) and nzint (non-zero integer numbers). The sort nzint of non-zero integer numbers is introduced in order to define division operators.

We chose to construct negative integer numbers by placing a minus sign '−' before positive natural numbers. So, the constructors of the integer numbers are the constructors of the natural numbers and the new constructor '−' of type posnat → nzint, which is defined below. Note that -0 is not a normal form. This does not mean that -0 is not a valid term, since the unary operator '−' is also a normal operator on integer numbers. It only means that -0 should be rewritable to a normal form. Fortunately, using equation [I1] the term -0 can be rewritten to the normal form 0 of sort nat.

As mentioned above, the sort nzint is defined in order to define the division operators _ div _, and _ mod _ on integer numbers. These operators are not defined if the second argument is 0. Here, the advantage of an AS formalism with subsorts, like MEL, becomes clear; these formalisms support partial functions.

Extending natural number operations to integer number operations merely means taking care of the sign of integer expressions. In addition to the extended operations, two new operations are defined: _ − _ (subtraction) and *abs* (absolute value). Note that by defining these operators on integer numbers, they can be applied on natural numbers, too.

spec INT
include NAT, BOOL.

sorts nzint, int.
subsorts
posnat < nzint < int,
nat < int.
constructor
$-\_ : \text{posnat} \to \text{nzint}$.
operators
$-\_ : \text{int} \to \text{int}$,
$s\_ : \text{int} \to \text{int}$,
$p\_ : \text{int} \to \text{int}$,
$abs : \text{int} \to \text{nat}$,
$\_ < \_ : \text{int int} \to \text{bool}$,
$\_ + \_ : \text{int int} \to \text{int}$ [comm, assoc],
$\_ - \_ : \text{int int} \to \text{int}$,
$d : \text{int int} \to \text{nat}$ [comm],
$\_ \times \_ : \text{int int} \to \text{int}$ [comm, assoc],
$\_ \,\mathsf{div}\, \_ : \text{int nzint} \to \text{int}$,
$\_ \,\mathsf{mod}\, \_ : \text{int nzint} \to \text{int}$,
$gcd : \text{nzint nzint} \to \text{posnat}$ [comm],
$exp : \text{int nat} \to \text{int}$.
var
$i, i_0, i_1 : \text{int}$,
$nzi : \text{nzint}$,
$n, n_0, n_1 : \text{nat}$,
$pn, pn_0, pn_1 : \text{posnat}$.
equations
[I1] $-0 = 0$,
[I2] $- - i = i$,
[I3] $p\ 0 = -s\ 0$,
[I4] $p(-pn) = -(s\ pn)$,
[I5] $s(-pn) = -(p\ pn)$,
[I6] $abs(n) = n$,
[I7] $abs(-pn) = pn$,
[I8] $-pn < n = true$,
[I9] $n < -pn = false$,
[I10] $-pn_0 < -pn_1 = pn_1 < pn_0$,
[I11] $-pn + n = d(pn, n) \quad \Leftarrow pn \leq n$,

[I12] $-pn + n = -d(pn, n) \Leftarrow n < pn$,
[I13] $-pn_0 + -pn_1 = -(pn_0 + pn_1)$,
[I14] $i_0 - i_1 = i_0 + -i_1$,
[I15] $d(-pn, n) = pn + n$,
[I16] $d(-pn_0, -pn_1) = d(pn_0, pn_1)$,
[I17] $-pn \times n = -(pn \times n)$,
[I18] $-pn_0 \times -pn_1 = pn_0 \times pn_1$,
[I19] $-pn_0 \mathsf{div}\ pn_1 = -(pn_0 \mathsf{div}\ pn_1)$,
[I20] $n \mathsf{div} -pn = -(n \mathsf{div}\ pn)$,
[I21] $-pn_0 \mathsf{div} -pn_1 = pn_0 \mathsf{div}\ pn_1$,
[I22] $i \mathsf{mod}\ nzi = i - ((i \mathsf{div}\ nzi) \times nzi)$,
[I23] $gcd(-pn_0, pn_1) = gcd(pn_0, pn_1)$,
[I24] $gcd(-pn_0, -pn_1) = gcd(pn_0, pn_1)$,
[I25] $exp(i, 0) = s\ 0$,
[I26] $exp(i, s\ n) = i \times exp(i, n)$.
end

2.5 • Rational numbers

In the same way as integer numbers are an extension of the natural numbers, rational numbers are an extension of the integer numbers. In specification RAT below, two sort names are defined: nzrat (non-zero rational numbers) and rat (rational numbers). The sort nzrat of non-zero rational numbers is introduced in order to define division operators on rational numbers.

There is one constructor operator for rational numbers, which is denoted by $\frac{i}{pn}$, where i : int and pn : posnat. In the operator definition $\frac{-}{-}$: nzint posnat $\rightarrow$ nzrat, it is unclear which argument position corresponds to the sort nzint and which corresponds to the sort posnat. We adopt the convention that the order of positions is determined in a left-right to top-bottom fashion, which means the top most position corresponds to the sort nzint and the bottom most position to the sort posnat. So, the sign of a rational number is stored in the numerator. This does not yield unique normal forms, since, for example, $\frac{1}{2}$ is equal to $\frac{4}{8}$ and $\frac{4}{2}$ is equal to 2. Therefore, two equations, Equations [R2] and [R3], are added that allow simplification of rational numbers to unique normal forms. A rational number $\frac{i}{pn}$ is in normal form if $i \neq pn$ and $1 \neq pn$ and $gcd(i, pn) = 1$. If the gcd is greater than 1, the normal form is computed by factoring out $gcd(i, pn)$.

Some, operations on integer numbers are extended to rational numbers. In addition, a division operator '/' on rational numbers and two conversion operations, *round* and *floor*, from the rational numbers to the integer numbers are given. The operation $round(r)$ computes the maximum of the integer numbers closest to the rational number r. Note that for any rational number, there are at most two closest integer numbers. For example, integer numbers 0 and 1 are equally close to $\frac{1}{2}$. Since $1 > 0$, the normal form of the term $round(\frac{1}{2})$ is 1. The operation $floor(r)$ computes the greatest integer less than or equal to the rational number r. For example, the normal form of $floor(\frac{1}{2})$ is 0.

spec RAT
include INT, BOOL.
sorts nzrat, rat.
subsorts
nzint < nzrat < rat,
int < rat.
constructor
$\frac{-}{\_}$: nzint posnat → nzrat.
operators
$\frac{-}{\_}$: int nzint → rat,
$-\_$: rat → rat,
$\_ + \_$: rat rat → rat [comm, assoc],
$\_ - \_$: rat rat → rat,
$\_ \times \_$: rat rat → rat [comm, assoc],
$\_/\_$: rat nzrat → rat,
abs : rat rat → rat,
round : rat → int,
floor : rat → int.
var
i, i_0, i_1 : int,
nzi, nzi_0, nzi_1 : nzint,
n : nat,
pn, pn_0, pn_1 : posnat,
r : rat.
equations
[R1] $\frac{0}{pn} = 0$,
[R2] $\frac{nzi}{s\ 0} = nzi$,
[R3] $\frac{nzi}{pn} = \frac{nzi \operatorname{div} gcd(nzi,pn)}{pn \operatorname{div} gcd(nzi,pn)}$ $\Leftarrow s\ 0 < gcd(nzi, pn)$,

[R4] $\frac{nzi}{-pn} = \frac{-nzi}{pn}$,
[R5] $-\frac{nzi}{pn} = \frac{-nzi}{pn}$,
[R6] $i_0 + \frac{i_1}{nzi} = \frac{(i_0 \times nzi)+i_1}{nzi}$,
[R7] $\frac{i_0}{nzi_0} + \frac{i_1}{nzi_1} = \frac{(i_0 \times nzi_1)+(i_1 \times nzi_0)}{nzi_0 \times nzi_1}$,
[R8] $i_0 - \frac{i_1}{nzi} = \frac{(i_0 \times nzi)-i_1}{nzi}$,
[R9] $\frac{i_0}{nzi} - i_1 = \frac{i_0-(nzi \times i_1)}{nzi}$,
[R10] $\frac{i_0}{nzi_0} - \frac{i_1}{nzi_1} = \frac{(i_0 \times nzi_1)-(i_1 \times nzi_0)}{nzi_0 \times nzi_1}$,
[R11] $i_0 \times \frac{i_1}{nzi} = \frac{i_0 \times i_1}{nzi}$,
[R12] $\frac{i_0}{nzi_0} \times \frac{i_1}{nzi_1} = \frac{i_0 \times i_1}{nzi_0 \times nzi_1}$,
[R13] $i/nzi = \frac{i}{nzi}$,
[R14] $\frac{nzi_0}{pn}/nzi_1 = \frac{nzi_0}{pn \times nzi_1}$,
[R15] $r/\frac{nzi_0}{nzi_1} = r \times \frac{nzi_1}{nzi_0}$,
[R16] $abs(\frac{pn_0}{pn_1}) = \frac{pn_0}{pn_1}$,
[R17] $abs(\frac{-pn_0}{pn_1}) = \frac{pn_0}{pn_1}$,
[R18] $floor(i) = i$,
[R19] $floor(\frac{i}{pn}) = i \operatorname{div} pn \quad \Leftarrow 0 < i$,
[R20] $floor(\frac{i}{pn}) = (i \operatorname{div} pn) - s\ 0 \quad \Leftarrow i < 0$,
[R21] $round(i) = floor(i + \frac{s\ 0}{s\ s\ 0})$.

end

2.6 · Channels

The channel data type defines communication channels. We use the convention to denote concrete channels by identifiers prefixed by the '∼' symbol. For instance, $\sim in$ and $\sim out$ are typical examples of concrete channels. Since the exact number of channels is application dependent, it is impossible to give a general MEL specification of channels. Therefore, we confine ourselves to a specification *scheme* CHANNEL. This scheme is parameterised by a set I of identifiers i, i', ...

spec CHANNEL
sort chan.
constructors
i : → chan,
i' : → chan,
⋮
end

2.7 · Elements

In this section we define a *theory* of elements. Theories are used to define general properties of data types, like existence of a particular sort. Such properties can be used to define data types that depend on these properties. To put it differently, a formal specification of properties by means of MEL theories enables us to define generic data types, like sets and lists (see Sections 2.9 and 2.10). The formal status of MEL theories is explained in Section C.3.

The difference between MEL specifications and MEL theories is their semantics. As mentioned in Section 2.1, MEL specifications have an initial algebra semantics. Using the same semantics for MEL theories is too restrictive. In fact, we want any possible algebra that has at least one set to be a valid interpretation of ELEMENT. Therefore, the semantics of MEL theories are allowed to have junk and confusion. The only requirements are that there is a set for every sort name and there are functions for every constructor and operator, such that the equations of the theory hold. Since the theory ELEMENT does not have equations, they are vacuously satisfied by all algebras having at least one set. So, any set, including the empty set, may serve as the interpretation of the sort elt. For instance, we could interpret sort elt as the set of booleans (the carrier set of the initial algebra of specification BOOLEAN). In this way, we can use the booleans as the ELEMENT parameter of the generic list data type (Section 2.10).

theory ELEMENT
sort elt.
end

Interpreting elements one way or another, means specifying a fixed interpretation of the sort elt. In MEL, this is formally done by so-called *views* from theories to specifications (or other theories). In the next section, we present some of these views from the theory ELEMENT to the specifications of the booleans, natural numbers, integer numbers, and rational numbers.

2.8 · Element views of basic data types

As mentioned above, instantiation of parameterised specifications or parameterised theories is done by views. A view is a mapping from a theory onto a specification or another theory. It is a generalisation of instantiating actual parameters for

formal parameters; the sorts, constructors, and operators of the specification are the actual parameters and the sorts, constructors, and operators of the theory are the formal parameters. We define four views from the theory ELEMENT to the specifications BOOL, NAT, INT, and RAT, respectively. Since there is only one sort defined in specification BOOL (Section 2.2), the view BOOL has to map the sort elt onto the sort bool. For the view NAT from ELEMENT to NAT, there are more choices, since there are two sorts (posnat and nat). Of course, we chose for nat, since that is the sort of all natural numbers. For the views from ELEMENT to INT and RAT, we chose for the sorts int and rat, respectively.

view BOOL
from ELEMENT **to** BOOL
sort elt **to** bool.
end

view NAT
from ELEMENT **to** NAT
sort elt **to** nat.
end

view INT
from ELEMENT **to** INT
sort elt **to** int.
end

view RAT
from ELEMENT **to** RAT
sort elt **to** rat.
end

2.9 · Sets

In this section, we give a specification of finite sets of elements. The set data type given in SET[X :: ELEMENT] below is generic; it is parameterised by a theory describing the elements of the set. Every specification for which there is an ELEMENT view (Section 2.7), can be used to instantiate the SET[X :: ELEMENT] specification. For example, the view BOOL (Section 2.8) maps the actual sort bool of BOOL

(Section 2.2) onto the formal sort elt of ELEMENT. So, we can build boolean sets by parameterising SET[X :: ELEMENT] by the view BOOL, as in SET[BOOL].

In the specification SET[X :: ELEMENT], three sort names are defined: elem (comma separated sequences of elements), neset (non-empty sets) and set (sets). The sort names are implicitly qualified with the parameter X. That is, the fully qualified names of the sort names are elem[X], neset[X], and set[X], respectively. After instantiation with BOOL, this becomes neset[BOOL], which is the sort of non-empty sets of booleans. The explicit qualification of elt.X indicates that the sort elt is defined in the theory ELEMENT.

There are two set constructors: '$\emptyset$' and '$\{\ldots\}$'. The first one denotes the empty set and the second one constructs a (non-empty) set from a sequence of elements. There is one constructor defined to build sequences of elements: ','. This constructor is an associative and commutative operator.

Equations [S1] and [S2] eliminate multiple occurrences of the same element in a sequence of elements. These equations are based on the commutativity and associativity of the operator ','. Using commutativity and associativity, occurrences of the same elements can be put next to each other after which the equations mentioned can be used to eliminate one of the occurrences.

We define the conventional set operators '$\in$' (membership test), '$\cap$' (set intersection), '$\cup$' (set union), '$\setminus$' (set difference), '$\subset$' (strict subset), and '$\subseteq$' (subset). In addition, we define a function *size* that computes the number of elements in a set.

spec SET[X :: ELEMENT]
include BOOL, NAT.
sorts elem, neset, set.
subsorts
  elt.X < elem,
  neset < set.
constructors
  _, _ : elem elem → elem [comm, assoc],
  $\emptyset$:→ set,
  {_} : elem → neset.
operators
  _ ∈ _ : elt.X set → bool,
  _ ∪ _ : set set → set [comm, assoc],
  _ ∩ _ : set set → set [comm, assoc],

$\_ \setminus \_ : \text{set set} \to \text{set}$,
$\_ \subset \_ : \text{set set} \to \text{bool}$,
$\_ \subseteq \_ : \text{set set} \to \text{bool}$,
$size : \text{set} \to \text{nat}$.

var

$e, e_0, e_1 : \text{elt.X}$,
$se, se_0, se_1 : \text{elem}$,
$s, s_0, s_1 : \text{set}$,
$ns : \text{neset}$.

equations

[S1] $e, e = e$,
[S2] $e, e, se = e, se$,
[S3] $e \in \emptyset = \mathit{false}$,
[S4] $e_0 \in \{e_1\} = (e_0 = e_1)$,
[S5] $e_0 \in \{e_1, se\} = (e_0 = e_1) \vee e_0 \in \{se\}$,
[S6] $\emptyset \cup s = s$,
[S7] $\{se_0\} \cup \{se_1\} = \{se_0, se_1\}$,
[S8] $\emptyset \cap s = \emptyset$,
[S9] $\{e\} \cap ns = \{e\} \quad \Leftarrow e \in ns$,
[S10] $\{e\} \cap ns = \emptyset \quad \Leftarrow \neg(e \in ns)$,
[S11] $\{e, se\} \cap ns = \{e\} \cup (\{se\} \cap ns) \quad \Leftarrow e \in ns$,
[S12] $\{e, se\} \cap ns = \{se\} \cap ns \quad \Leftarrow \neg(e \in ns)$,
[S13] $\emptyset \setminus s = \emptyset$,
[S14] $s \setminus \emptyset = s$,
[S15] $\{e\} \setminus ns = \emptyset \quad \Leftarrow e \in ns$,
[S16] $\{e\} \setminus ns = \{e\} \quad \Leftarrow \neg(e \in ns)$,
[S17] $\{e, se\} \setminus ns = \{se\} \setminus ns \quad \Leftarrow e \in ns$,
[S18] $\{e, se\} \setminus ns = \{e\} \cup (\{se\} \setminus ns) \quad \Leftarrow \neg(e \in ns)$,
[S19] $s_0 \subseteq s_1 = (s_0 \setminus s_1 = \emptyset)$,
[S20] $s_0 \subset s_1 = s_0 \subseteq s_1 \wedge s_0 \neq s_1$,
[S21] $size(\emptyset) = 0$,
[S22] $size(\{e\}) = s\ 0$,
[S23] $size(\{e, se\}) = s\ (size(\{se\} \setminus \{e\}))$.

end

2.10 · Lists

This section defines the list data type. As SET[X :: ELEMENT], specification LIST[X :: ELEMENT] is parameterised by the ELEMENT theory (Section 2.7). Therefore, to build boolean lists, the specification LIST[X :: ELEMENT] should be instantiated with the view BOOL of Section 2.8, as in LIST[BOOL].

Lists are comma separated sequences of elements enclosed in brackets. The order of the elements and the number of occurrences of an element is relevant. For instance, using decimal notation for integer numbers, the list $[0, 1, 2, 3]$ is different from the list $[0, 1, 3, 2]$ and $[0]$ is different from $[0, 0]$. List concatenation is written with the operator '$+\!\!+$', as in $[0, 1] +\!\!+ [2, 3]$. Membership test is denoted by the $\in$ operator, as in $3 \in [0, 1, 2, 3]$.

Specification LIST[X :: ELEMENT] defines three sorts: elem (comma separated sequences of elements), nelist (non-empty lists), and list (lists). As with the sorts of specification SET[X :: ELEMENT] (see Section 2.9) the sorts of LIST[X :: ELEMENT] are implicitly qualified with the parameter X. That is, the fully qualified names of the sorts are elem$[X]$, nelist$[X]$, and list$[X]$, respectively.

List subtraction is defined by Equations [L7]–[L12]. For each i-th occurence ($i = 0, 1, 2, \ldots$) of an element e in the second argument, if the i-th occurrence of e exists in the first argument, it is removed from this argument. For instance, $[2, 1, 1, 4, 1]$ - - $[4, 1, 2, 1] = [1]$.

The list operations *hd* and *tl* compute the head and the tail of a list, respectively. The head of a (nonempty) list is its first element. The tail of a (nonempty) list is the list except its first element. Both the head and the tail operation have a reversed variant, called *hr* (head right) and *tr* (tail right), respectively. They compute the head and the tail starting from the end (the right side) of the list. So, $hr(l)$ computes the last element in the list l and $tr(l)$ computes the list l except for its last element.

The *take* and *drop* functions [32] compute a list consisting of a given number of elements of a given list. The expression $take(l, n)$ computes the list consisting of the first n elements of list l and $drop(l, n)$ function computes the list consisting of the elements following the first n elements of l.

spec LIST[X :: ELEMENT]
include BOOL, NAT.
sorts elem, nelist, list.

subsorts

elt.X < elem,

nelist < list.

constructors

_, _ : elem elem → elem [assoc],

[ ] :→ list,

[_] : elem → nelist.

operators

_ ++ _ : list list → list [assoc],

_ ∈ _ : elt.X list → bool,

_ -- _ : list list → list,

hd(_) : nelist → elt.X,

hr(_) : nelist → elt.X,

tl(_) : nelist → list,

tr(_) : nelist → list,

len(_) : list → nat,

take(_, _) : list nat → list,

drop(_, _) : list nat → list.

var

e, e_0, e_1 : elt.X,

l, l_0, l_1 : list,

le, le_0, le_1 : elem,

nl : nelist,

n : nat.

equations

[L1] $l +\!\!+ [\,] = l$,

[L2] $[\,] +\!\!+ l = l$,

[L3] $[le_0] +\!\!+ [le_1] = [le_0, le_1]$,

[L4] $e \in [\,] = \mathit{false}$,

[L5] $e_0 \in [e_1] = (e_0 = e_1)$,

[L6] $e_0 \in [e_1, le] = (e_0 = e_1) \vee e_0 \in [le]$,

[L7] $[\,] \text{ -- } l = [\,]$,

[L8] $l \text{ -- } [\,] = l$,

[L9] $[e] \text{ -- } l = [\,] \quad \Leftarrow e \in l$,

[L10] $[e] \text{ -- } l = [e] \quad \Leftarrow \neg(e \in l)$,

[L11] $[e, le] \text{ -- } l = [le] \text{ -- } (l \text{ -- } [e]) \quad \Leftarrow e \in l$,

[L12] $[e, le] \text{ -- } l = [e] +\!\!+ ([le] \text{ -- } l) \quad \Leftarrow \neg(e \in l)$,

[L13] $len([\,]) = 0$,
[L14] $len([e]) = s\ 0$,
[L15] $len([e, le]) = s\ (len([le]))$,
[L16] $hd([e]) = e$,
[L17] $hd([e, le]) = e$,
[L18] $hr([e]) = e$,
[L19] $hr([le, e]) = e$,
[L20] $tl([e]) = [\,]$,
[L21] $tl([e, le]) = [le]$,
[L22] $tr([e]) = [\,]$,
[L23] $tr([le, e]) = [le]$,
[L24] $take(l, 0) = [\,]$,
[L25] $take([\,], n) = [\,]$,
[L26] $take([e], n) = [e] \quad \Leftarrow n > 0$,
[L27] $take([e, le], n) = [e] +\!\!+ take([le], p\ n) \quad \Leftarrow n > 0$,
[L28] $drop(l, 0) = l$,
[L29] $drop([\,], n) = [\,]$,
[L30] $drop([e], n) = [\,] \quad \Leftarrow n > 0$,
[L31] $drop([e, le], n) = drop([le], p\ n) \quad \Leftarrow n > 0$.
end

2.11 · Tuples

In this section we define the tuple data type. In fact, the specification is not a pure MEL specification, but rather a specification *scheme* to generate MEL specifications. We are aware of the fact that specification schemes are not defined formally, but we think it can be understood unambiguously. Once the *reflection* mechanism of MEL is explained, a formal treatment of specification schemes in MEL is possible. For a treatment of reflection in MEL, or actually, reflection in the more general logic called *rewriting logic*, we refer to [58]. A formal treatment of specification schemes in MEL is outside the scope of this thesis. In the TUPLE[X_0, ..., X_{n-1} :: ELEMENT] specification scheme, we use indices from a set I, where I is a set of n successive natural numbers starting from zero. Note that if I is the empty set, there are no X_i parameters for $i \in I$. For each set I, the scheme generates a pure MEL specification of n-tuples, that is, it generates a MEL specification with n parameters, each of which is described by the theory ELEMENT (see Section 2.7).

The specification scheme TUPLE[X_0, ..., X_{n-1} :: ELEMENT] defines one sort: tuple (n-tuples). As with the sorts of SET[X :: ELEMENT] (see Section 2.9) and the sorts of LIST[X :: ELEMENT] (see Section 2.10) the sort name tuple is implicitly qualified. The fully qualified sort name is tuple[X_0, ..., X_{n-1}].

There is one constructor operators for n-tuples, denoted by '$\langle \ldots \rangle$'. In addition, there are n postfix operators, denoted by '$.i$' ($0 \leq i < n$), to access the elements in an n-tuple. The operator '$.i$' returns the i^{th} element of a tuple.

spec TUPLE[X_0, ..., X_{n-1} :: ELEMENT]
sort tuple.
constructor
$\langle \_, \ldots, \_ \rangle$: elt.X_0 ... elt.X_{n-1} → tuple.
operators
$\_.0$: tuple → elt.X_0.
⋮
$\_.(n-1)$: tuple → elt.X_{n-1}.
var
e_0 : elt.X_0,
⋮
e_{n-1} : elt.X_{n-1}.
equation
[T1] $\langle e_0, \ldots, e_{n-1} \rangle.0 = e_0$,
⋮
[Tn] $\langle e_0, \ldots, e_{n-1} \rangle.(n-1) = e_{n-1}$.
end

To illustrate how the scheme is used, we present the specification of TUPLE[X_0, X_1 :: ELEMENT] of two-tuples. This specification is the result of taking the index set $I = \{0, 1\}$.

spec TUPLE[X_0, X_1 :: ELEMENT]
sort tuple.
constructor
$\langle \_, \_ \rangle$: elt.X_0 elt.X_1 → tuple.
operators
$\_.0$: tuple → elt.X_0,

_.1 : tuple → elt.X$_1$.
var e_0 : elt.X$_0$,
e_1 : elt.X$_1$.
equation
[T1] $\langle e_0, e_1 \rangle.0 = e_0$,
[T2] $\langle e_0, e_1 \rangle.1 = e_1$.
end

The specification TUPLE[X$_0$, X$_1$] can be instantiated by specifications and theories for which there is an ELEMENT view. For example, a specification of two tuples containing a boolean and an integer is defined by TUPLE[BOOL, INT], where BOOL and INT are names of ELEMENT views for the booleans and integer numbers, respectively (see Section 2.8).

The specification of empty tuples, see TUPLE[] below, is the degenerated instantiation resulting from $I = \emptyset$. The qualified sort name tuple.TUPLE[] contains exactly one element, the empty tuple, which is denoted by '$\langle\rangle$'. Since the empty tuple does not contain elements, there are no indexing operators. The empty tuple specification is useful, since it is the formal interpretation of χ's void type (see Section 6.1).

spec TUPLE[]
sort tuple.
constructor
$\langle\rangle$: → tuple.
end

2.12 · Element views of generic data types

In this section we define three more views. The new views are themselves parameterised by theories. Therefore, using a new view to instantiate, for example, specification SET[X :: ELEMENT], results in a parameterised specification which can be instantiated as usual.

The views we define here are views from ELEMENT to SET[X :: ELEMENT], from ELEMENT to LIST[X :: ELEMENT], and from ELEMENT to TUPLE[X$_0$, X$_1$ :: ELEMENT]. Therefore, we now can define a (parameterised) specification of sets of lists of something.

All three views are parameterised by the ELEMENT theory. To build sets of lists of booleans, the module SET[X :: ELEMENT] should be instantiated with the view LIST[X :: ELEMENT], which in turn should be instantiated with the view BOOL, as in SET[LIST[BOOL]].

Our explanation for the specification scheme of tuples (see Section 2.11) also applies to the view scheme of tuples given below. That is, we do not give n MEL views of n-tuples for all n, but we describe a view scheme by which we can generate such views.

view SET[X :: ELEMENT]
from ELEMENT **to** SET[X]
sort elt **to** set.
end

view LIST[X :: ELEMENT]
from ELEMENT **to** LIST[X]
sort elt **to** list.
end

view TUPLE[X_0, ..., $X_(n-1)$ :: ELEMENT]
from ELEMENT **to** TUPLE[$X_0, \ldots, X_(n-1)$]
sort elt **to** tuple.
end

2.13 · Additional data types

As mentioned in the introduction of this chapter, χ has data types that are not or cannot be defined by means of AS. These types are discussed below.

Type void is χ's so-called *empty* type. It contains no elements. The void type is used to construct synchronisation ports and χ-channels (see Section 3.3).

Type void can be defined by means of a MEL specification, but we choose not to. In χ_σ, we use type tuple[ ] and its only element '$\langle\rangle$' to model synchronisation (see Sections 2.11 and 6.3).

Type real represents the real numbers.

Type string represents arbitrary sequences of characters enclosed by double quotes.

Type file can be used to import or export data from or to a file.

Distributions are defined on the types bool, nat, int, and real by using the type constructor 'dist[t]'. For instance, a real-valued distribution is of type dist[real].

2.14 · Discussion

The main contribution of this chapter is a formal treatment of χ data types. The formalised data types of χ are, per definition, the data types of χ_σ. Treating data types in a formal way is a necessary step towards a formalisation of χ.

For the formalisation of data types, we used AS, and in particular MEL. The MEL specifications have been validated using Maude. We conclude that the MEL-Maude combination provides powerful techniques and support to define data types. However, for some data types it is less effective. For instance, for the definition of tuples we had to resort to specification schemes, and real numbers cannot be specified in MEL at all.

The specification language $\chi \cdot 3$

This chapter introduces the specification language χ. It serves as a starting point for the formalisation described in the next section. Furthermore, it is also needed to understand the discussions concerning χ and χ_σ in Chapters 5 and 6.

The specification language χ is inspired by CSP and the *guarded command language*. Similar to CSP, the behaviour of system components is described by *processes* that communicate via *channels*. Communication in χ is synchronous, unidirectional, and timeless. In case two processes synchronise via a channel no information is communicated and communication is undirected. In addition, statements can be preceded by guards like in guarded command languages. Processes can be grouped into *systems* by means of parallel composition. Such a system can act as a process; it can be combined with other processes and systems to form a new system.

The data types of χ are defined by the equational specifications of the previous chapter. The closed terms of these data types are called *constant expressions* or *values*. The set of these constant expressions is called *Value*. Typical elements of *Value* are denoted by c, c', ...

We assume there is a set of typed programming variables called *Var*. Programming variables are typically denoted by x, x', ... The type of a programming variable is a sort defined in the MEL specifications of the previous chapter. Furthermore, programming variables of a sort s can occur whenever a term of sort s can occur. This results in a new set of terms called *Expr*. Elements of this set are *expressions* and typically denoted by e, e', ... Notice that *Expr* contains new normal forms since we do not define additional equations. As a consequence, by adding programming variables, the data types change. However, we are only interested in χ specifications where all programming variables have a value. That is, before terms are evaluated, programming variables are substituted by their values. Therefore, we can ignore the new normal forms in calculations on data types.

Simulation time is modelled by the global read-only programming variable τ of type real. Via τ we can refer to the current simulation time in expressions. An increase of the value of τ is interpreted as passage of time. That is, if a process delays, the programming variable τ is increased by the number of time units delayed.

We use a standard format for describing the syntax of χ called the Backus Naur Form (BNF) [10]. A syntax definition in BNF consists of definitions of the form *identifier* ::= *definition*, where *identifier* is a term that describes a particular part of the syntax, '::=' should be interpreted as 'consists of', and *definition* is a list of what this part of the syntax may contain. This list may contain other identifiers, or literal strings. Within such a definition, '|' can be used to separate alternatives.

This chapter is organised as follows. First, we define type aliases and constants (Sections 3.1 and 3.2). Then, we define processes, systems, and functions (Sections 3.3, 3.4, and 3.5). Next, we discuss how χ experiments are defined (Section 3.6). This chapter is concluded by a discussion (Section 3.7).

3.1 · Type aliases

A type alias is a user-defined name for a type. Type aliasing can make specifications more readable. For example, if products are represented by natural numbers, we can define a type alias 'prod' for type nat. This gives us the opportunity to declare programming variables of 'type' prod and perform operations on these programming variables as if they were of type nat (addition, multiplication, etc.). This can be achieved by writing

type prod = nat.

In general, a χ type alias definition T has syntax

T ::= type T',

T' ::= $id = DT$ (type alias)
| T', T' (type alias list),

with id a fresh user-defined identifier and DT data type as defined in Chapter 2 or a type alias (the text between parentheses are comments). Remark that user-

defined identifiers should be different from χ identifiers (keywords, operator names, and function names).

3.2 · Constants

Constants are defined in the same way as type aliases are defined. Using the keyword const we define the constant, its type, and its value. Consider for example the definition of the constants *pi* and *batchsize*:

const pi: real $= 3.1415$, $batchsize$: nat $= 4$.

In general, a χ constant definition C has the following syntax, with id a fresh user-defined identifier and c a value:

$C ::=$ const C',

$C' ::= id$: $DT = c$ (constant definition)
$\quad\quad | \; C'$, C' (constant definition list).

3.3 · Processes

As mentioned in the preamble of this chapter, χ processes can communicate via channels. Channels are connected to processes via *ports*. We have send, receive, and synchronisation ports. Send and receive ports can be constructed by preceding a type by an exclamation mark and a question mark, respectively. A synchronisation port can be constructed by preceding type void by a tilde.

Before we define the syntax of χ processes, we first discuss a small example. Consider a machine M that processes products. It requires 5 time units to process a product. After a product has been processed, it is transported to the next workstation. We define a process M with two ports. Products are received and sent via these ports. We represent products by natural numbers and define type alias prod. The specification then reads

type prod $=$ nat

proc $M(a$: ? prod, b: ! prod$) = [\![\, x$: prod $| *[$true $\longrightarrow a \,?\, x \;;\; \Delta 5 \;;\; b \,!\, x] \,]\!]$.

A process description is declared by the keyword proc followed by its name, in this case M, possibly extended with a parameter declaration. As can be seen, M has two ports (parameters a and b, respectively), one for receiving and one for sending data of type prod.

The body of the process description is surrounded by a pair of double brackets. A process body is divided into two parts, the declaration part and the action part. The two are separated by the separator '|'.

In the first part of the process body programming variables are declared. This means that we introduce programming variables and also specify their type. Process M has one programming variable, named x, of type prod.

The first statement of M is a repetitive guarded command statement of the form $*[e \longrightarrow S]$. The meaning of this statement is that as long as boolean expression e evaluates to true, the sequence of statements S is executed. In this example the value of the boolean guard is always true, so the statements following the '$\longrightarrow$' will be executed forever. The first statement after the arrow is receive statement $a\,?\,x$. It denotes that M wants to receive a product via port a. The product is 'stored' in programming variable x. Then we specify that processing a product takes 5 time units by statement $\Delta 5$. Finally, we send the processed product away via statement $b\,!\,x$. It is possible that process M has to wait for communication via port a or b because the environment is not able to communicate immediately. It can even be the case that the environment never offers communication via a or b. In that case process M deadlocks.

Next, we present the syntax rules for χ processes. In general, a χ process definition P has syntax

$$
\begin{aligned}
P ::= {} & \mathsf{proc}\ id(D_0) = [\![\ S_p\ ]\!] \\
| {} & \mathsf{proc}\ id(D_0) = [\![\ D_1 \mid S_p\ ]\!],
\end{aligned}
$$

with id an identifier, D_0 and D_1 declarations, and S_p a process statement. Declarations D_0 and D_1 have the following syntax, with x a programming variable:

$$
\begin{aligned}
D_0 ::= {} & \qquad \text{(empty formal parameter list)} \\
| {} & D_0',
\end{aligned}
$$

$D'_0 ::= x\colon DT$ (formal parameter declaration)
$\quad | \; x\colon \; !\, DT$ (formal send port parameter declaration)
$\quad | \; x\colon \; ?\, DT$ (formal receive port parameter declaration)
$\quad | \; x\colon {}^{\sim}\text{void}$ (formal synchronisation port parameter declaration)
$\quad | \; D'_0,\; D'_0$ (formal parameter declaration list),

$D_1 ::= x\colon DT$ (programming variable declaration)
$\quad | \; D_1,\; D_1$ (programming variable declaration list).

Process statement S_p has the following syntax, with d a distribution, e an expression, e_b a boolean expression, e_{num} a numerical expression, i an iterator, l a lower bound for iterator i, p a port, u an upper bound for iterator i, and x a programming variable:

$S_p ::= \mathsf{skip}$ (skip statement)
$\quad | \; \mathsf{terminate}$ (terminate statement)
$\quad | \; \mathsf{setseed}(d, e_{num})$ (set seed statement)
$\quad | \; x := e$ (assignment statement, $x \not\equiv \tau$ since τ is read only)
$\quad | \; E$ (event statement)
$\quad | \; S_p \; ; \; S_p$ (sequential composition)
$\quad | \; [GC\,]$ (guarded command statement)
$\quad | \; *[GC\,]$ (repetitive guarded command statement)
$\quad | \; [SW\,]$ (selective waiting statement)
$\quad | \; *[SW\,]$ (repetitive selective waiting statement)
$\quad | \; !e$ (print statement)
$\quad | \; ?x$ (input statement),

$E ::= \Delta\; e_{num}$ (delay statement)
$\quad | \; p\,!\,e$ (send statement)
$\quad | \; p!$ (synchronisation send statement)
$\quad | \; p\,?\,x$ (receive statement)
$\quad | \; p?$ (synchronisation receive statement)
$\quad | \; p^{\sim}$ (synchronisation statement),

$GC ::= R : e_b \longrightarrow S_p$ (iterative alternative composition)
$\quad | \; GC \; [\!] \; GC$ (alternative composition),

$SW ::= R : e_b \; ; \; E \longrightarrow S_p$ (iterative alternative composition)
$\quad | \; SW \; [\!] \; SW$ (alternative composition),

$$\begin{array}{ll} R ::= i\colon \text{nat} \leftarrow l..u & \text{(range including } l\text{, excluding } u\text{)} \\ \quad \mid R, R & \text{(range list).} \end{array}$$

The following abbreviations are allowed in process definitions.

- In a declaration, $x\colon DT,\ y\colon DT$ can be written as $x, y\colon DT$.
- In a guarded command, $R : e_b \longrightarrow S_p$ can be written as $e_b \longrightarrow S_p$ if the range(s) in R contain(s) one element.
- In a selective waiting, $R : e_b\ ;\ E \longrightarrow S_p$ can be written as $e_b\ ;\ E \longrightarrow S_p$ if the range(s) in R contain(s) one element.

Below, we explain the elements of process statement S_p.

- The skip statement in χ is in fact the empty statement; it does nothing.
- The terminate statement is for simulation purposes. Its execution aborts the simulation.
- The setseed statement takes two arguments, a distribution and an expression of type nat. The second argument determines the actual sample results of the distribution given in the first argument.

  When a distribution d is initialised, a seed is created which determines the results of every sample of d to come. Consecutive samples are still random but the actual result of the samples is determined by that particular seed. Every time the program is executed the distribution is initialised and a new seed is created. This new seed differs from the previous one and so do the samples taken from the distribution.

  The setseed statement can be used to set a particular seed. This generated seed is then the same for every new execution of the program. In that way, the same sample results can be obtained in consecutive program executions. This is especially useful for debugging purposes.

  For example, suppose that we want to model a machine M that processes a lot with a processing time that is Gamma distributed. The Gamma distribution has parameters p and q where the distribution mean m equals $p \times q$ and its variance v equals $p \times q^2$. For debugging purposes, we want to set the seed ourselves. This can be done as follows where $\sigma\ d$ draws a sample from distribution d:

$$
\begin{array}{l}
\mathsf{proc}\ M(a\colon\ ?\,\mathrm{lot},\ b\colon\ !\,\mathrm{lot},\ m,v\colon \mathrm{real}) = \\
[\![\ d\colon \mathrm{dist}[\mathrm{real}], x\colon \mathrm{lot}, p, q\colon \mathrm{real} \\
\mid p := \tfrac{m^2}{v}\ ;\ q := \tfrac{v}{m} \\
;\ d := \mathit{gamma}(p,q)\ ;\ \mathsf{setseed}(d, 37) \\
;\ *[\mathrm{true} \longrightarrow a\,?\,x\ ;\ \Delta(\sigma\ d)\ ;\ b\,!\,x] \\
]\!].
\end{array}
$$

- The assignment statement assigns a value to a programming variable. In assignment statement $x := e$, programming variable x is assigned the value that evaluation of expression e yields. The types of x and e should be compatible. Again, note that $x \not\equiv \tau$ because τ is a read only programming variable.

  It is also allowed to do what is called *multiple assignments* in one statement. For example, consider the process definition

  $$\mathsf{proc}\ P(\langle x, y\rangle\colon \mathrm{tuple}[\mathrm{nat}, \mathrm{nat}]) = [\![\ \langle x, y\rangle := \langle y, x\rangle\ ]\!].$$

  This simple process swaps the values of the parameters x and y in one statement.

- A delay statement Δe enables a process to delay. If $e > 0$, then for $0 < d \leq e$ this process can delay d time units. If $e = 0$, this statement ends, and if $e < 0$, this statement deadlocks. Furthermore, it can be used as a time out when placed in a selective waiting statement (more information on time outs follows in the explanation of the repetitive selective waiting statement).

- A send statement $p\,!\,e$ can be used to send the value of expression e over the channel connected to port p if at the same time another process is able to receive that value from the same channel. In case we only synchronise and do not exchange information, we write $p!$.

- A receive statement $p\,?\,x$ can be used to receive a value from the channel connected to port p and assign it to programming variable x if at the same time another process is able to send that value over the same channel. In case we only synchronise and do not exchange information, we write $p?$.

- A synchronisation statement $p^{\sim}$ can be used to synchronise via a channel connected to port p if at the same time another process is able to synchronise via the same channel.

- In order to denote that process statement S'_p is to be executed after process statement S_p, we write $S_p\ ;\ S'_p$.

- A guarded command statement can be used to select different statements, depending on the value of the guards. Upon execution of a guarded command, all guards are evaluated. If none of the guards evaluate to *true*, then execution of the guarded command statement fails; it deadlocks. In case more than one guard evaluates to *true*, one alternative is chosen non-deterministically and the corresponding statement is executed.

- A repetitive guarded command statement can be used if we want a guarded command to be executed repeatedly as long as one or more guards evaluate to *true*. The repetition ends if all guards yield *false*, then the statement following the repetitive guarded command is executed.

- A selective waiting statement is somewhat like a guarded command statement except that an event statement *ES* follows the guard. For all the guards that evaluate to *true*, the construct waits until at least one of the event statements is enabled, that is, a time out has elapsed or a communication or synchronisation can take place. From these enabled alternatives one alternative is chosen non-deterministically, the event is executed, and execution continues with the statements following the '$\longrightarrow$'. If none of the guards evaluate to *true*, the construct deadlocks.

- A repetitive selective waiting statement can be used if we want a selective waiting to be executed repeatedly as long as one of the guards evaluates to *true*. The repetition ends if all guards yield *false*. In that case, the statement following the repetitive selective waiting is executed.

  For example, suppose we want to model a conveyor belt. This conveyor receives lots on one side of the belt and delivers these lots at a workstation on the other side of the belt. Transportation takes t time units. Now, consider the process definition

  $$
  \begin{array}{l}
  \mathsf{proc}\ C(a\colon\ ?\,\mathrm{lot},\ b\colon\ !\,\mathrm{lot},\ t\colon \mathrm{real}) = \\
  [\![\ xs\colon \mathrm{list}[\mathrm{tuple}[\mathrm{lot}, \mathrm{real}]],\ x\colon \mathrm{lot} \\
  \mid\ xs := [\,] \\
  ;\ *[\ \mathrm{true} \quad ;a\,?\,x \quad \longrightarrow xs := xs +\!\!+ [\langle x, \tau + t\rangle] \\
  \quad [\![\ len(xs) > 0\ ;\Delta hd(xs).1 - \tau \longrightarrow b\,!\,hd(xs).0\ ;\ xs := tl(xs) \\
  \quad]\ ]\!].
  \end{array}
  $$

  If a lot x is received at the conveyor by statement $a\,?\,x$, it is stored in a tuple together with the moment in time lot x reaches the end of the conveyor.

This moment in time equals the current time τ plus the conveying time t: $\tau + t$. Now, we can consider our lots labelled with the moment in time that they have to leave the conveyor. All these labelled lots are stored in a list. If the conveyor contains lots, that is, if the length of list xs is greater than zero ($len(xs) > 0$), then the lot in the head of list xs is the first one to leave the conveyor. This is the case if the second element of that lot is equal to the value of the current simulated time, as in $\langle x, t\rangle.1 = \tau$. In that case, $hd(xs).1 - \tau = 0$, and product $hd(xs).0$ can be sent via b.

As can be seen in the process description, we have a rather strange selective waiting statement. We don't have a send or receive statement following the guard $len(xs) > 0$, but the delay statement $\Delta hd(xs).1 - \tau$ instead. This is what is called a time-out statement. In case of the selective waiting statement in C, a time-out occurs as soon as the expression $hd(xs).1 - \tau$ equals 0. At that moment the statements following the '$\longrightarrow$' of the time-out can be executed. This means that the lot which was ready to leave the system can be sent away.

- The print statement $!e$ writes an expression e to standard output. The expression to be printed can be a comma separated list that can contain strings, programming variables, and functions like $tab()$ (to next tab stop) and $nl()$ (to next line).

- The input statement $?x$ reads input from standard input and stores it in programming variable x.

3.4 · Systems

As mentioned in the preamble of this chapter, processes can be grouped into a system. Before we define the syntax of χ systems, we first discuss a small example.

Consider the processes G, M, and E as defined below. Process G and E represent the environment of machine M (generator and exit, respectively). We use these processes to define system GME. Within system GME the three processes are connected by the channels $\sim gm$ and $\sim me$. System GME is depicted in Figure 3.1.

For the definitions of G, M, and E we have

```
type prod = nat
```

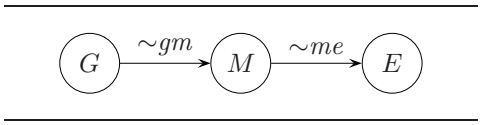


Figure 3.1 · System *GME*.

$$\textsf{proc}\ G(a\colon\ !\,\text{prod}) =$$
$$[\![\, n\colon \text{prod} \mid n := 0\ ;\ *[\text{true} \longrightarrow a\,!\,n\ ;\ n := n+1\,]\,]\!]$$

$$\textsf{proc}\ M(a\colon\ ?\,\text{prod},\ b\colon\ !\,\text{prod},\ t\colon \text{real}) =$$
$$[\![\, x\colon \text{prod} \mid *[\text{true} \longrightarrow a\,?\,x\ ;\ \Delta t\ ;\ b\,!\,x\,]\,]\!]$$

$$\textsf{proc}\ E(a\colon\ ?\,\text{prod}) =$$
$$[\![\, x\colon \text{prod} \mid *[\text{true} \longrightarrow a\,?\,x\,]\,]\!].$$

System *GME* can now be defined as

$$\textsf{syst}\ GME(t\colon \text{real}) =$$
$$[\![\, {\sim}gm, {\sim}me\colon\ -\,\text{prod} \mid G({\sim}gm) \parallel M({\sim}gm, {\sim}me, t) \parallel E({\sim}me)\,]\!].$$

A system description is declared by the keyword syst followed by its name, in this case *GME*, possibly extended with a parameter declaration. In this case the processing time t of machine M is a parameter of *GME*. The fact that it has no incoming or outgoing channels makes it a *closed* system.

The body of a system description, like process descriptions, is surrounded by a pair of double brackets. In the declaration part, channels are declared. They are declared in a way similar to declarations of programming variables in processes. A channel is constructed by placing the minus symbol in front of the type that is sent via this channel. Just like in process descriptions, the declaration part is separated from the action part by the separator '|'. In the action part, we instantiate processes and systems with the appropriate channels and parameters. The different processes are put in parallel by means of the operator '‖'. This operator denotes that process statements are executed concurrently.

In general, a χ system definition S has syntax

$$\begin{aligned} S ::= {} & \textsf{syst}\ id(D_0) = [\![\, S_s\,]\!] \\ & \mid \textsf{syst}\ id(D_0) = [\![\, D_2 \mid S_s\,]\!], \end{aligned}$$

with *id* an identifier, D_0 and D_2 declarations, and S_s a system statement. Declaration D_0 has already been defined for processes in Section 3.3. Declaration D_2 has the following syntax, with c a channel:

$$\begin{array}{ll} D_2 ::= c\colon -DT & \text{(channel declaration)} \\ \quad \mid D_2, D_2 & \text{(channel declaration list).} \end{array}$$

System statement S_s has the following syntax, with e an expression, I a process or system instantiation, and R a range as defined for processes in Section 3.3:

$$\begin{array}{ll} S_s ::= R : I & \text{(iterative parallel instantiation)} \\ \quad \mid S_s \parallel S_s & \text{(parallel instantiation),} \end{array}$$

$$I ::= id(L) \quad \text{(instantiation),}$$

$$\begin{array}{ll} L ::= & \text{(empty actual parameter list)} \\ \quad \mid L', & \end{array}$$

$$\begin{array}{ll} L' ::= e & \text{(actual parameter)} \\ \quad \mid L', L' & \text{(actual parameter list).} \end{array}$$

The following abbreviations are allowed in system definitions.

- In a channel declaration, $c\colon -DT,\ d\colon -DT$ can be abbreviated to $c, d\colon -DT$.
- In a iterative parallel instantiation, $R : I$ can be abbreviated to I if the range(s) in R contain(s) one element.

The parallel composition operator is explained below.

- The parallel composition operator executes two processes and/or systems concurrently. Execution of the statements contained by parallel processes is timeless (with respect to the simulated time) except for the delay statement. A delay statement can only be executed if other processes in parallel with the process containing the delay statement can also delay. This is the case if the other processes are at that moment able to execute a delay statement or a send or receive statement.

3.5 · Functions

Real-life models usually consist of many processes and systems and often perform a lot of complex data manipulation in the form of complicated algorithms. It would be convenient if we could refer to these algorithms by a name augmented with the required arguments. This shortens a specification and also improves its readability.

In order to neatly describe such algorithms and in order to separate them from the process specification itself, *functions* can be defined. Before we define the syntax for χ functions, we first discuss a small example.

Consider the function *reverse* which reverses lists of natural numbers. So, for example $reverse([1,2,3]) = [3,2,1]$. The definition of function *reverse* reads

```
func reverse(xs: list[nat]) → list[nat] =
⟦ ys: list[nat]
| ys := [ ]
; ∗[len(xs) > 0 ⟶ ys := [hd(xs)] ++ ys ; xs := tl(xs)]
; ↑ys
⟧.
```

In the same way as we specify a process or system, we can specify a function by the keyword func followed by its name, its arguments, and its return type. In this case, the function's argument *xs* is of type list[nat] and its return type (the type after the '→') is also of type list[nat]. Also, like in process descriptions, we can declare local programming variables (in this example the programming variable *ys*), and use them in the function body. The calculated result is returned by return statement $\uparrow e$ with e an expression. A user-defined function call is similar to function calls for predefined functions such as *hd* and *tl*. That is, a function is called by its name and its arguments enclosed in parentheses. A function call is an expression and may occur wherever ordinary expressions may occur provided that type correctness is preserved. Furthermore, note that we can use *xs* in the same way as *ys*. Changes made to *xs* have no influence outside the function because χ uses a call by value parameter mechanism.

Although recursion is not allowed in χ processes and systems, it is allowed in χ functions. The definition of function *reverse* that uses recursion reads

$$
\begin{array}{l}
\mathsf{func}\ \mathit{reverse}(\mathit{xs}\colon \mathrm{list[nat]}) \to \mathrm{list[nat]} = \\
[\!][\ [\ \mathit{len}(\mathit{xs}) = 0 \longrightarrow \uparrow[\,] \\
\quad [\!]\ \mathit{len}(\mathit{xs}) > 0 \longrightarrow \uparrow \mathit{reverse}(\mathit{tl}(\mathit{xs})) +\!\!+ [\mathit{hd}(\mathit{xs})] \\
\quad] \\
]\!].
\end{array}
$$

In general, a χ function definition F has syntax

$$
\begin{array}{l}
F ::= \mathsf{func}\ \mathit{id}(D_3) \to DT = [\!][\ S_f\ ]\!] \\
\qquad \mid \mathsf{func}\ \mathit{id}(D_3) \to DT = [\!][\ D_1 \mid S_f\ ]\!],
\end{array}
$$

with id an identifier, D_1 and D_3 declarations, and S_f a function statement. Declaration D_1 has already been defined for processes in Section 3.3. Declaration D_3 has the following syntax:

$$
\begin{array}{ll}
D_3 ::= & \text{(empty formal parameter list)} \\
\quad \mid D_1. &
\end{array}
$$

Function statement S_f has the following syntax, with e an expression, e_b a boolean expression, x a programming variable, and R a range expression as defined for processes in Section 3.3:

$$
\begin{array}{ll}
S_f ::= \uparrow e & \text{(return statement)} \\
\quad \mid \mathsf{skip} & \text{(skip statement)} \\
\quad \mid x := e & \text{(assignment statement, } x \not\equiv \tau\text{)} \\
\quad \mid S_f\ ;\ S_f & \text{(sequential composition)} \\
\quad \mid [GC'] & \text{(guarded command statement)} \\
\quad \mid *[GC'] & \text{(repetitive guarded command statement),}
\end{array}
$$

$$
\begin{array}{ll}
GC' ::= R : e_b \longrightarrow S_f & \text{(iterative alternative composition)} \\
\quad \mid GC' [\!]\ GC' & \text{(alternative composition).}
\end{array}
$$

The following abbreviations are allowed in function definitions.

- In a declaration, $x\colon DT,\ y\colon DT$ can be abbreviated to $x, y\colon DT$.
- In a guarded command, $R : e_b \longrightarrow S_f$ can be abbreviated to $e_b \longrightarrow S_f$ if the range(s) in R contain(s) one element.

The elements of function statement S_f that also appear as elements of process statement S_p act similarly in a S_f context. The only statement new here is the return statement which is explained below.

- The return statement returns the value of expression e to the statement that performed the actual function call.

In χ, a function is to be used in a strict mathematical sense: every call of the same function with the same parameter values should result in the same outcome. This restriction is not enforced by the χ function syntax, but should be respected by the user. For instance, the following 'function' is not a valid χ function:

$\mathsf{func}\ \mathit{illegal}() \to \mathrm{nat} = [\![\, [\mathit{true} \longrightarrow \uparrow 0 \,[\!]\, \mathit{true} \longrightarrow \uparrow 1\,]\,]\!]$.

3.6 · Experiments

An experiment is a concrete instantiation of a process or system description. As an example, we return to system *GME* of Section 3.4. Suppose we want to do an experiment on system *GME* using the χ simulator. In that case, we have to instantiate system *GME* in the experiment environment. We define an experiment on *GME* for the case that process M has a processing time of 3.5 time units:

$\mathsf{xper} = [\![\, \mathit{GME}(3.5)\,]\!]$.

In general, a χ experiment definition E has the following syntax, with I an instantiation as defined in Section 3.4:

$E ::= \mathsf{xper} = [\![\, I\,]\!]$ (experiment definition).

3.7 · Discussion

This chapter introduced the specification language χ. It presented the syntax of χ and described its semantics. This chapter serves as a starting point for the formalisation described in the next chapter. It is also needed to understand the discussions concerning χ and χ_σ in Chapters 5 and 6.

The specification language χ_σ · 4

In this chapter, we define the formal syntax and semantics of χ_σ. The formal definitions are illustrated by examples in Chapter 8. We follow the standard process algebraic approach where the semantics of processes is expressed in terms of *process graphs*. A process graph is a special *Labelled Transition System* (LTS). These process graphs are defined in a *Structural Operational Semantics* style (SOS), also called Structured Operational Semantics style [1, 163].

This chapter is organised as follows. We introduce preliminary notions on states and stacks in Section 4.1. Next, Section 4.2 defines the semantical model of χ_σ. In Section 4.3, we discuss timing aspects and explain our decisions regarding χ_σ's time model. Strong bisimulation on χ_σ processes is defined in Section 4.4. The Sections 4.5 through 4.14 define χ_σ processes and operators by means of deduction rules. Section 4.15 defines a stratification which shows that these deduction rules are meaningful. In addition, Section 4.16 shows that strong bisimulation is a congruence for all χ_σ process operators. Section 4.17 discusses properties of χ_σ processes, and Section 4.18 describes how process definitions are specified. This chapter is concluded by a discussion in Section 4.19.

4.1 · States and stacks

This section summarises the definitions on states and stacks that appear in Appendix A. Furthermore, Appendix A contains a number of lemmas on states and stacks. Some of these lemmas are used in proofs presented in this chapter.

We assume there is a countably infinite number of distinct identifiers, which are typically denoted by i, i', ... Identifiers can be used to denote *programming variables* or *channels*. Recall that programming variables are typically denoted by x, x', ... (see Chapter 3). Channels are typically denoted by m, m', ... Programming variable identifiers and channel identifiers are associated with values

(also called constant expressions, see Chapter 3). Recall that values are typically denoted by c, c', ... The association of an identifier and a value is called a *valuation* and is denoted by $i \mapsto c$ as defined in Definition A.1. Valuations are typically denoted by v, v', ... The notation $i \mapsto \perp$ denotes that there is a value c such that $i \mapsto c$. This notation allows the value of an identifier to be unspecified.

States are lists of valuations as defined in Definition A.2 and are typically denoted by s, s', ... Furthermore, it is required that the identifiers occurring in the valuations be mutually distinct. That is, each identifier occurs at most once in a state. The empty state is denoted by λ_s. A nonempty state is constructed from a valuation v and a state s and is denoted by $v : s$. The set of all states is called *State*.

The function *dom* is defined in Definition A.3 and returns the domain of a state, that is, it returns the set of identifiers in the state. An identifier is *defined* in a state if and only if it is in the domain of the state. If it is not in the domain, it is *undefined* in that state.

The value associated with an identifier in a state can be changed by the *substitution* operator. Substitution on states is defined in Definition A.4 as

$$\begin{aligned} \lambda_s[c/i] &= \lambda_s, \\ (i \mapsto c : s)[c'/i] &= i \mapsto c' : s, \\ (i \mapsto c : s)[c'/i'] &= i \mapsto c : s[c'/i'] \quad \text{if } i \neq i'. \end{aligned}$$

Note that an update can never add new valuations (and consequently new identifiers) to states. If the identifier to be updated does not occur in a valuation in a state, then substitution is the identity operation.

Recall that in Chapter 3, we defined the set *Expr* of expressions. Expressions are typically denoted by e, e', ... The set *Expr* contains terms according to the MEL specifications defined in Chapter 2. Furthermore, these terms may contain programming variables, as mentioned in Chapter 3.

If e is an expression and s is a state, then the evaluation of e in s is written as $s(e)$. By evaluating an expression, a value may result. However, since e can contain programming variables that are undefined in a state s, it is possible that evaluation $s(e)$ is not a value, but remains an expression.

To evaluate expressions, identifiers have to be looked up in states. Looking up identifiers in states is defined in Definition A.5 as

$$\begin{aligned} \lambda_s(i) &= i, \\ (i \mapsto c : s)(i) &= c, \\ (i \mapsto c : s)(i') &= s(i') \quad \text{if } i \neq i'. \end{aligned}$$

Two states are equivalent if for every identifier i evaluation of i in those two states has the same result. Equivalence on states is defined in Definition A.6 as

$$s = s' \quad \text{if } \forall i : s(i) = s'(i).$$

The *set* function updates the value of an identifier in a state, or, if the identifier does not occur in the state, adds the identifier and the value to a state. It is defined in Definition A.7 as

$$\begin{aligned} set(s, \lambda_s) &= s, \\ set(s, i \mapsto c : s') &= set(s[c/i], s') \qquad \text{if } i \in dom(s), \\ set(s, i \mapsto c : s') &= set(i \mapsto c : s, s') \quad \text{if } i \notin dom(s). \end{aligned}$$

States can be stacked in so-called *state stacks* as defined in Definition A.8. These state stacks are typically denoted by σ, σ', ... The empty state stack is denoted by λ_σ. A nonempty state stack is constructed by a state s and a state stack σ and is denoted by $s :: \sigma$. The set of all state stacks is called *Stack*. In the remainder of this document, we mostly refer to state stacks by simply using the word stack. In a stack, the same identifier can occur more than once, but only in different states of the stack.

The function *dom* on stacks as defined in Definition A.9, returns the domain of a stack. That is, it returns the set of identifiers in the (states of the) stack. A identifier *defined* in a stack if and only if it is in the domain of the stack. If it is not in the domain, it is *undefined* in that stack.

Substitution is generalized on stacks in Definition A.10 as

$$\begin{aligned} \lambda_\sigma[c/i] &= \lambda_\sigma, \\ (s :: \sigma)[c/i] &= s[c/i] :: \sigma \quad \text{if } i \in dom(s), \\ (s :: \sigma)[c/i] &= s :: \sigma[c/i] \quad \text{if } i \notin dom(s). \end{aligned}$$

Similarly, evaluation of expressions is generalized to stacks. For instance, if e is an expression and σ a stack, then $\sigma(e)$ is the evaluation of e in σ. Consequently, looking up identifiers is also generalized on stacks, as defined in Definition A.11 as

$$\begin{aligned} \lambda_\sigma(i) &= i, \\ (s :: \sigma)(i) &= s(i) \quad \text{if } i \in dom(s), \\ (s :: \sigma)(i) &= \sigma(i) \quad \text{if } i \notin dom(s). \end{aligned}$$

Equivalence on stacks is defined in Definition A.12 as

$$\begin{aligned} \lambda_\sigma &= \lambda_\sigma, \\ s :: \sigma &= s' :: \sigma' \quad \text{if } s = s' \wedge \sigma = \sigma'. \end{aligned}$$

In addition, observational equivalence on stacks is defined in Definition A.13 as

$$\sigma \stackrel{\circ}{=} \sigma' \quad \text{if } \forall i : \sigma(i) = \sigma'(i).$$

4.2 · A semantical model for χ_σ

In this section, we define a semantical model for χ_σ. As mentioned above, we use process graphs to express the semantics of χ_σ processes. A process graph is a special form of an LTS.

Definition 4.1 · (LTS) *An LTS is a triple $(S, R_{S\times S}, R_S)$, with S a set of states, $R_{S\times S}$ a set of binary relations on states, and R_S a set of unary relations on states.*

Suppose we have an LTS $(S, R_{S\times S}, R_S)$ and $s, s' \in S$, $r \in R_{S\times S}$, and $r' \in R_S$. If there is a pair $(s, s') \in r$ or $s \in r'$, then we say there is a *transition* from s to s' or a transition for s, respectively. In this thesis, we are interested only in LTSs where each state is a closed χ_σ process term. The signature of χ_σ processes is presented in Sections 4.5 through 4.14. The set of all χ_σ process terms is called P and the set of all closed χ_σ process terms is called $C(P)$. Often, we write 'process', where formally we should write 'process term'.

The semantics of χ_σ processes defines their *action behaviour*, *delay behaviour*, and *termination behaviour*. Action behaviour and delay behaviour define how processes evolve into other processes by performing *actions* or *delays*. Action behaviour and delay behaviour depends on the context and can change the context. For instance, the action behaviour of a process that should update a programming variable x using the value of a programming variable y, depends on the value of y and changes the value of x. We use stacks to represent these contexts. Termination behaviour defines whether processes have finished properly. Similarly to action behaviour and delay behaviour, termination behaviour of processes depends on the context.

The set of all actions is called *Action* and contains the *internal* action τ (not to be confused with the current time expression τ of χ), the *assignment* action $aa(x, c)$, the *send* action $sa(m, c)$, the *receive* action $ra(m, x)$, and the *communication* action $ca(m, x, c)$:

$$Action = \{\tau, aa(x, c), sa(m, c), ra(m, x), ca(m, x, c)\},$$

where $x \in Id$, $c \in Value$, and $m \in Channel$. The sets *Id*, *Value*, and *Channel* represent the set of identifiers, the set of values, and the set of channels, respectively. The set of all delays is the set $R_{>0}$ of positive real numbers. So, if a process delays, the duration of this delay is defined by a positive real number $d \in R_{>0}$.

In order to put the action behaviour, delay behaviour, and termination behaviour of processes in the framework of LTSs, we will now define binary relations for action behaviour and delay behaviour and unary relations for termination behaviour. The binary relations are given by triples (σ, ℓ, σ') where $\sigma, \sigma' \in Stack$ and $\ell \in Action \cup R_{>0}$. The unary relations are given by stacks σ with $\sigma \in Stack$. This results in the following definition of χ_σ-LTSs.

Definition 4.2 • (χ_σ-LTS) *A χ_σ-LTS, is an LTS $(S, R_{S\times S}, R_S)$, such that*

- $S \subseteq C(P)$,
- *all $r \in R_{S\times S}$ are binary relations on closed χ_σ processes given by triples*
  - $(\sigma, a, \sigma') \in Stack \times Action \times Stack$ *or*
  - $(\sigma, d, \sigma') \in Stack \times R_{>0} \times Stack$, *and*
- *all $r \in R_S$ are unary relations on closed χ_σ processes given by stacks $\sigma \in Stack$.*

For instance, let triple $(S, R_{S\times S}, R_S)$ be a χ_σ-LTS. If $r \in R_{S\times S}$ and r is given by (σ, a, σ'), then for all $(p, p') \in r$ we say there is an *action transition* from p to p', which is denoted by $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$. Similarly, if r is given by (σ, d, σ'), we say there is a *delay transition* for all $(p, p') \in r$. Delay transitions are denoted by $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$. Finally, if $r \in R_S$ and r is given by σ, then for all $p \in r$ we say there is a *termination* for p, which is denoted by $\langle p, \sigma \rangle\downarrow$.

As mentioned above, we use process graphs as the semantical model of χ_σ processes. Process graphs are LTSs with one distinguished state, called the *initial state*, and all other states reachable from that state. If a process graph is represented graphically, the states are nodes, action transitions and delay transitions are edges, and terminations are node labels. We use the following conventions.

- Action transitions are solid edges labelled by an action.
- Delay transitions are dashed edges labelled by a positive real number.
- Terminations are represented by grey states.
- The initial state has a double circle.
- States without (outgoing) transitions and terminations are black.

Despite the fact that the last two items are not defined as unary relations, we do have a convention for these properties. The initial state indicates the starting point of the process graph and is easily recognisable by the double circle. States without transitions (that is, nodes without outgoing edges) and terminations are *deadlock* states and usually indicate design errors. We illustrate our convention by a process that can perform an action a and terminate, or delay for 2 time units and deadlock. The process graph of this process is depicted in Figure 4.1. The numbers in the states are for referring purposes. As can be seen, node 2 is the root node, node 1 is a termination node, and node 0 is a deadlock node.

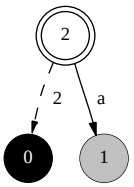


Figure 4.1 · Example process graph.

As mentioned, we use SOS theory to define the operational semantics of χ_σ. That is, we define a set of deduction rules that describes how χ_σ processes can evolve into other χ_σ processes. A deduction rule consists of hypotheses and a conclusion. Hypotheses and conclusions are formulas.

We assume there is a predicate TRUE on boolean expressions. TRUE(e_b) holds if and only if the equality $e_b = \mathit{true}$ can be proven according to the data type specifications of Chapter 2.

Definition 4.3 · (Formulas) *A formula has one of the following forms, where* $e_b \in$ bool, $p, p' \in P$, $\sigma, \sigma' \in Stack$, $a \in Action$, *and* $d \in R_{>0}$*:*

1. $\text{TRUE}(e_b)$,
2. $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$,
3. $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$,
4. $\langle\, p, \sigma \,\rangle\!\downarrow$,
5. $\neg\exists p' \in C(P), \sigma', a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$,
6. $\neg\exists p' \in C(P), \sigma', d : \langle\, p, \sigma \,\rangle \xrightarrow{d} \langle\, p', \sigma' \,\rangle$,
7. $\neg\langle\, p, \sigma \,\rangle\!\downarrow$.

For obvious reasons, forms 1–4 are called *positive* formulas and forms 5–7 are called *negative* formulas. Note that if $p, p' \in C(P)$, the forms 2–4 are action transitions, delay transitions, and terminations, respectively.

By convention, a formula $\text{TRUE}(e_b)$ can be abbreviated to e_b. Also, formulas of forms 5–7 can be abbreviated to $\langle\, p, \sigma \,\rangle \not\rightarrow$, $\langle\, p, \sigma \,\rangle \not\mapsto$, and $\langle\, p, \sigma \,\rangle \not\downarrow$, respectively.

Definition 4.4 · (Deduction rule) *A deduction rule consists of a set of formulas* H *and a formula* c*.* H *is the set of* hypotheses *and* c *is the* conclusion*. Furthermore,* c *is of the form 2, 3, or 4 of Definition 4.3. A deduction rule is denoted by* $\frac{H}{c}$.

We use the following convention: two rules $\frac{H}{c}$ and $\frac{H}{c'}$ can be written as $\frac{H}{c, c'}$.

Validity of the hypotheses of a deduction rule, under a certain substitution θ, implies validity of the conclusion of this rule under θ. In this way proofs (of action transitions, delay transitions, and terminations) can be established. In order to establish a proof for a negative formula, it should be manifestly impossible to derive the positive counterpart of the formula. That is, to prove $\neg\exists p', \sigma', a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, one has to show that it is impossible to prove $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, for all $p' \in C(P)$ and all $\sigma, \sigma' \in Stack$, and $a \in Action$.

The deduction rules defined in this chapter constitute a *transition system specification* (TSS) as described in [1, 71]. The transitions that can be proven from a TSS (in the general setting of [71] transitions are elements of both binary and unary relations) define an LTS. In our case, the χ_σ-LTS contains action transitions, delay transitions, and terminations that can be proven from the deduction

rules. In general, TSSs with negative hypotheses might not be *meaningful*. That is, it might be unclear whether the TSS defines an LTS and if it does, it might be unclear whether it defines a unique LTS. Because the SOS of χ_σ has negative hypotheses (hypotheses of the forms 5–7 of Definition 4.3), we have to show that the set of deduction rules that define the TSS of χ_σ is meaningful. This is done in Section 4.15 using SOS theory.

4.3 · A time model for χ_σ

This section discusses some timing aspects that need to taken into consideration. For example, we have to decide whether to implement a discrete or a continuous time model. Also, we have to decide in which way passage of time interacts with the ability to perform actions and the ability to terminate.

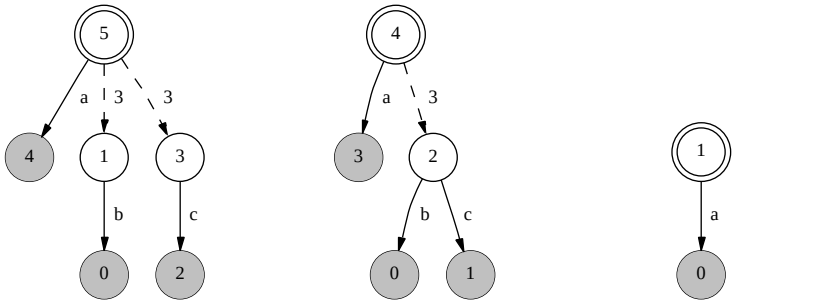
Firstly, we mention discrete and continuous time. A discrete time implementation describes time by viewing the time domain as an enumeration of time slices, where every slice covers the same amount of time, expressed in an arbitrary time unit. In a continuous time implementation all timing is measured on a continuous time scale. Since χ has continuous time, χ_σ has continuous time, too.

Two other timing aspects are *time factorisation* and *maximal progress*. Time factorisation preserves choices between alternatives that can delay for the same amount of time. That is, progress of time does not make a choice in that case. For instance, consider a machine that processes different types of products. For each product type, there is a different operation mode. Suppose this machine is idle and waits for input. Then the choice for a specific operation mode is determined *not* before or while the machine waits, but as soon as a product is received (after some delay).

Sometimes, an additional condition applies to passage of time. Namely, that passage of time is allowed only if no other activity is possible. This is called *strong time factorisation*, or maximal progress [13]. The weaker variant is then called *weak time factorisation*. Timed extensions of the process algebra ACP have weak time factorisation built in [16]. The Algebra of Timed Processes has strong time factorisation built in [155]. From here on, we use the terms time factorisation and maximal progress where with time factorisation we mean weak time factorisation.

For example, consider a process that can perform an action a, or delay for 3 time units and perform action b, or delay for 3 time units and perform action c. The

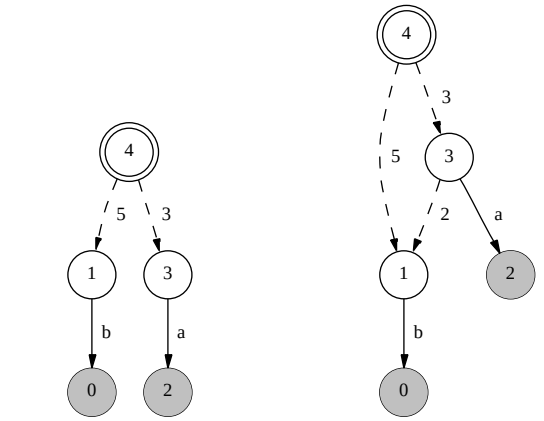
process graph for the case that time factorisation is not implemented is depicted in Figure 4.2(a). Here, we can see that progress of time does make a choice. The process graph for the case that time factorisation is implemented is depicted in Figure 4.2(b). As can be seen, progress of time does not make a choice here, where it did in the case without time factorisation. The choice to perform action b or c is preserved until after the delay. The process graph for the case that maximal progress is implemented is depicted in Figure 4.2(c). It is obvious that here the opportunity to delay is lost since we only allow this if no other activity is possible.



(a) No time factorisation. (b) Time factorisation. (c) Maximal progress.

Figure 4.2 • Time factorisation and maximal progress.

Note that the common interpretation of time factorisation does not prohibit a choice to be made between two alternatives that do not necessarily delay for the same amount of time. So, it does not prohibit the execution of a large delay thereby losing alternatives that cannot perform that large delay. Consider for example a process that can delay for 3 time units and perform action a, or delay for 5 time units and perform action b. The process graph for the case that time factorisation is not implemented is depicted in Figure 4.3(a) and the case that time factorisation is implemented is depicted in Figure 4.3(b). As can be seen, time factorisation does not prohibit the delay of 5 time units to be executed at once. Another possible interpretation is the one depicted in Figure 4.3(c). In that case time factorisation establishes that progress of time never determines a choice. Here, it implies that the opportunity to perform action a after 3 time units should not go unnoticed if time progresses.



(a) No time factorisation.

(b) Time factorisation (interpretation 1).

(c) Time factorisation (interpretation 2).

Figure 4.3 · Different interpretations of time factorisation.

The specification language χ has both statements with time factorisation (selective waiting statements) and statements without (guarded command statements). We refrained from including both in χ_σ and decided to include time factorisation. Furthermore, if time factorisation is undesired, it can be suppressed. For example, see the translation of guarded command statements in Chapter 6.

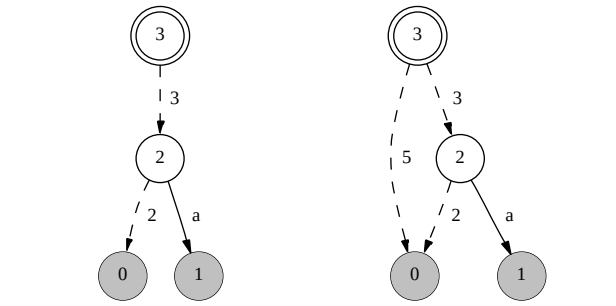
We decided to implement time factorisation so that progress of time can never determine a choice (the interpretation as depicted in Figure 4.3(c)). So, if two or more alternatives can delay, then they will delay together. As a consequence, opportunities for action performance or termination cannot be ignored. Furthermore, since we want a process to be able to wait until communication with another process is possible, but do not want two processes to continue waiting if they can communicate, we also need maximal progress. We decided not to incorporate maximal progress in all the rules defining χ_σ because we implemented χ_σ's communication mechanism using delayable send and receive processes. Therefore, we define an operator that introduces maximal progress.

In addition, we mention *time determinism*. In [76, 115] time determinism is defined as follows. If a process p can evolve into a process q by delaying and p can also evolve into q' by the same amount of delay, then q and q' are equal. So, a delay step

always leads to a unique result. In [191] this is called time determinacy. Another, different, definition can be found in [70]. There time determinism expresses that choices can be decided by passage of time as illustrated in Figure 4.2(a). In this thesis we adhere to the first interpretation. The semantics of χ_σ incorporates time determinism as we show in Lemma 4.48.

Finally, we mention *time additivity*. In [115], time additivity is defined as follows. If a process p can evolve into a process q by delaying t time units and process q can evolve into process r by delaying t' time units, then p can evolve into r by delaying $t + t'$ time units.

For example, consider a process that can delay for 3 time units and can then choose to perform an action a or delay for 2 more time units. Figure 4.4(a) shows the process graph of this process in case time additivity is not implemented. In case we do have time additivity, that process can also delay for 5 time units from the start and ignore the opportunity to perform action a after 3 time units. This is depicted in Figure 4.4(b).



(a) No time additivity. (b) Time additivity.

Figure 4.4 • Time additivity.

Because of our opinion that opportunities for action performance or termination should not go unnoticed if time progresses, we choose not to implement time additivity. As a consequence, we also do not have time additivity in a case where it seems reasonable. Namely, consider the process that first delays for 3 time units followed by a delay for 2 time units. It seems reasonable to allow this process to delay also for 5 time units at once. On the other hand, one can argue if that

was intended in the first place, the process description should be that this process delays for 5 time units.

4.4 · Equivalences on χ_σ processes

In order to define properties of and theorems about χ_σ processes, we need a notion of equivalence on these processes. It is the standard *strong bisimulation* concept [79, 142, 20, 159] we use for this. A bisimulation is a relation on processes expressed in terms of defined relations on these processes.

Definition 4.5 · (Strong Bisimulation) *A strong bisimulation on processes is a relation $R \in P \times P$ such that for all $(p, q) \in R$ the following holds:*

1. $\forall \sigma : \langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\!\downarrow$,
2. $\forall \sigma, a, p', \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \wedge (p', q') \in R$,
3. $\forall \sigma, a, q', \sigma' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \wedge (p', q') \in R$,
4. $\forall \sigma, d, p', \sigma' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \wedge (p', q') \in R$,
5. $\forall \sigma, d, q', \sigma' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \wedge (p', q') \in R$.

Two processes p and q are strongly bisimilar, denoted by $p \leftrightarrows q$, if there exists a bisimulation relation R such that $(p, q) \in R$.

Notice that the definition of strong bisimulation treats delay transitions exactly the same as action transitions. That is, strong bisimulation as defined here is an instantiation of strong bisimulation on general LTSs.

A strong bisimulation relation as defined above is an equivalence relation. That is, it is reflexive, symmetric, and transitive. The proofs for reflexivity and symmetry are trivial. Transitivity can be proved as follows: suppose there are processes p, q and r, such that $p \leftrightarrows q$ and $q \leftrightarrows r$, then according to Definition 4.5, there exist bisimulation relations R_{pq} and R_{qr} such that $(p, q) \in R_{pq}$ and $(q, r) \in R_{qr}$. Define $R_{pr} = \{(x, z) \mid \exists y : (x, y) \in R_{pq} \wedge (y, z) \in R_{qr}\}$ and show that R_{pr} is a strong bisimulation by verifying that it satisfies the conditions of Definition 4.5.

In Section 4.16, we prove that strong bisimulation equivalence is a *congruence* for the process operators of χ_σ.

In some proofs to come, we use the 'bisimulation up to' technique [143, 141] to prove that two processes are strongly bisimilar. A 'bisimulation up to $\underline{\leftrightarrow}$'-relation is defined as follows.

Definition 4.6 · (Bisimulation up to $\underline{\leftrightarrow}$) *A relation $R \in P \times P$ on processes is a 'bisimulation up to $\underline{\leftrightarrow}$' if for all $(p, q) \in R$ the following holds:*

1. $\forall \sigma : \langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\!\downarrow$,
2. $\forall \sigma, a, p', \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow$
   $\exists p'', q', q'' : p' \underline{\leftrightarrow} p'' \wedge \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \wedge q' \underline{\leftrightarrow} q'' \wedge (p'', q'') \in R$,
3. $\forall \sigma, a, q', \sigma' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \Rightarrow$
   $\exists q'', p', p'' : q' \underline{\leftrightarrow} q'' \wedge \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \wedge p' \underline{\leftrightarrow} p'' \wedge (p'', q'') \in R$,
4. $\forall \sigma, d, p', \sigma' : \langle\, p, \sigma \,\rangle \xmapsto{d} \langle\, p', \sigma' \,\rangle \Rightarrow$
   $\exists p'', q', q'' : p' \underline{\leftrightarrow} p'' \wedge \langle\, q, \sigma \,\rangle \xmapsto{d} \langle\, q', \sigma' \,\rangle \wedge q' \underline{\leftrightarrow} q'' \wedge (p'', q'') \in R$,
5. $\forall \sigma, d, q', \sigma' : \langle\, q, \sigma \,\rangle \xmapsto{d} \langle\, q', \sigma' \,\rangle \Rightarrow$
   $\exists q'', p', p'' : q' \underline{\leftrightarrow} q'' \wedge \langle\, p, \sigma \,\rangle \xmapsto{d} \langle\, p', \sigma' \,\rangle \wedge p' \underline{\leftrightarrow} p'' \wedge (p'', q'') \in R$.

The following lemma shows that in order to prove that two processes are strongly bisimilar, it suffices to prove that there exists a 'bisimulation up to $\underline{\leftrightarrow}$' relation between the processes.

Lemma 4.7 · *Let R be a 'bisimulation up to $\underline{\leftrightarrow}$' relation and p and q be processes. If $(p, q) \in R$ then $p \underline{\leftrightarrow} q$.*

Proof · (Lemma 4.7) We have to prove that if a relation R is a 'bisimulation up to $\underline{\leftrightarrow}$' relation and p and q processes such that $(p, q) \in R$ then $p \underline{\leftrightarrow} q$. Therefore, we have to show that there exists a strong bisimulation relation R' between p and q. We define R' as

$$R' = \{(p, q) \mid \exists p', q' : p \underline{\leftrightarrow} p' \wedge (p', q') \in R \wedge q' \underline{\leftrightarrow} q\}.$$

That is, $R' = \underline{\leftrightarrow} \circ R \circ \underline{\leftrightarrow}$. Note that $R \subseteq R'$ since $\underline{\leftrightarrow}$ is a reflexive relation. We have to prove that for all $(p, q) \in R'$ the five bisimulation conditions of Definition 4.5 hold.

Condition 1: We have to prove $\forall \sigma : \langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\!\downarrow$. Since $(p, q) \in R'$ we know that there are p' and q' such that $p \underline{\leftrightarrow} p'$, $(p', q') \in R$, and $q' \underline{\leftrightarrow} q$. So, we can make the following computation:

$$
\begin{array}{ll}
 & \langle\, p, \sigma \,\rangle\downarrow \\
\Leftrightarrow & \{\text{Condition 1 of the strong bisimulation relation}\} \\
 & \langle\, p', \sigma \,\rangle\downarrow \\
\Leftrightarrow & \{\text{Condition 1 of the 'bisimulation up to } \underline{\leftrightarrow}\text{' relation}\} \\
 & \langle\, q', \sigma \,\rangle\downarrow \\
\Leftrightarrow & \{\text{Condition 1 of the strong bisimulation relation}\} \\
 & \langle\, q, \sigma \,\rangle\downarrow.
\end{array}
$$

Condition 2: We have to prove $\forall a, p_a, \sigma, \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma' \,\rangle \Rightarrow \exists q_a : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q_a, \sigma' \,\rangle \wedge (p_a, q_a) \in R'$. Figure 4.5 illustrates the proof. Since $(p, q) \in R'$, we know that there exists p' and q' such that $p \underline{\leftrightarrow} p'$, $(p', q') \in R$, and and $q' \underline{\leftrightarrow} q$.

So, suppose $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma' \,\rangle$. Then by the definition of strong bisimulation, we know that there exists a p'_a such that $\langle\, p', \sigma \,\rangle \xrightarrow{a} \langle\, p'_a, \sigma' \,\rangle$ and $p_a \underline{\leftrightarrow} p'_a$. By the definition of 'bisimulation up to $\underline{\leftrightarrow}$' we also know that there exist p''_a, q'_a, and q''_a such that $\langle\, q', \sigma \,\rangle \xrightarrow{a} \langle\, q'_a, \sigma' \,\rangle$, $p'_a \underline{\leftrightarrow} p''_a$, $(p''_a, q''_a) \in R$, and $q''_a \underline{\leftrightarrow} q'_a$. Therefore, we can use the definition of strong bisimulation again to derive that there exists a process q_a such that $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q_a, \sigma' \,\rangle$ and $q'_a \underline{\leftrightarrow} q_a$. Furthermore, using transitivity of $\underline{\leftrightarrow}$, we get $p_a \underline{\leftrightarrow} p''_a$ and $q''_a \underline{\leftrightarrow} q_a$. So, we have $p_a \underline{\leftrightarrow} p''_a$, $(p''_a, q''_a) \in R$, and $q''_a \underline{\leftrightarrow} q_a$. Therefore, $(p_a, q_a) \in R'$, which concludes the proof for condition 2.

Condition 3–5: The proof is similar to the proof of condition 2. □

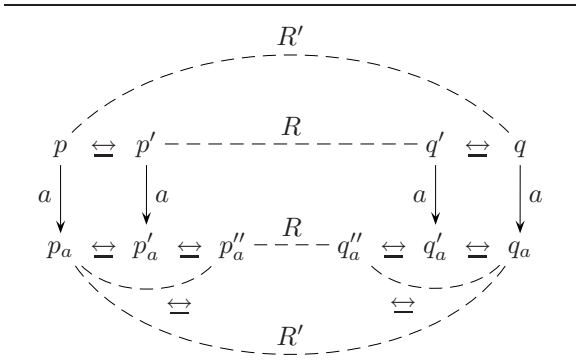


Figure 4.5 · Diagram for the proof of Lemma 4.7

Sometimes, the notion of strong bisimulation is too restrictive. That is, instead of verifying that two given processes behave exactly alike, it suffices to verify that except for some 'irrelevant' behaviour, the two processes behave alike. Specification-implementation checks are typical examples of verifications where all that is needed is equality up to some irrelevant behaviour (internal behaviour). To enable such verifications, χ_σ has the internal action τ, as introduced in [141], and an abstraction operator (see Section 4.14). The τ action can be used to model activity of processes that does not need to be specified in detail. For instance, if the input-output behaviour of a production system is analysed, the activity to transform an input into an output does not need to be specified in every detail, but it can be modelled by a τ action. The abstraction operator, *renames* some particular actions of p into τ. Thus, the abstraction operator can be used to abstract from particular actions by turning them into internal actions. The next step is to define an equivalence relation on (timed) processes that takes into account internal activity as modelled by the τ action. Timed *branching bisimulation* does just that. Coming up with a proper definition of timed branching bisimulation (such that it is a congruence) requires a substantial amount of theoretical research [79, 22, 77, 123]. Another option is to abstract from delay transitions as is the case in time-abstracting bisimulations [189]. Recall that our objective is to develop a formal method (consisting of a formal language, a mathematical framework, and tools) and to assess it by performing case studies. In order to evenly divide our efforts, we only investigated one equivalence relation (strong bisimulation) in our mathematical framework. We refrained from defining timed branching bisimulation on χ_σ processes. Consequently, we cannot perform specification-implementation checks in the mathematical framework presented in this chapter. However, in Section 7.4, we present theoretical results enabling specification-implementation checks to some extent.

4.5 · Atomic processes

The atomic processes of χ_σ are called atomic because they are the χ_σ process constructors. More complex processes are constructed using the process operators we define in the sections to come. So, they are called atomic because they are not composed from any process operators defined in χ_σ and cannot be split into smaller χ_σ processes. This makes them χ_σ's elementary processes.

In Definition 4.8, we define the following atomic processes.

- The empty process ε, which is empty in the sense that it cannot display any activity (perform an action or delay), but can only terminate.
- The deadlock process δ, which denotes 'no behaviour'. That is, if a process deadlocks, then neither it is able to continue any form of activity nor terminate correctly.
- The skip process skip, which performs the internal action τ.
- The delay process Δe, which is able to delay an arbitrary number of time units less than or equal to the value of expression e.
- The assignment process $x := e$, which assigns the value of expression e to programming variable x.
- The send process $m\,!\,e$, which sends the value of expression e via channel m.
- The receive process $m\,?\,x$, which receives a value via channel m and assigns it to programming variable x.

Definition 4.8 · (Atomic processes) *The atomic processes of χ_σ have the following signature with $Expr_R$ the set of real number expressions:*

$$\begin{array}{rcl} \varepsilon & : & P, \\ \delta & : & P, \\ \mathsf{skip} & : & P, \\ \Delta\_ & : & Expr_R \to P, \\ \_ := \_ & : & Id \times Expr \to P, \\ \_\,!\,\_ & : & Channel \times Expr \to P, \\ \_\,?\,\_ & : & Channel \times Id \to P. \end{array}$$

The deduction rules for χ_σ's atomic processes are listed in Table 4.1.

Rule 1 states that the empty process can terminate. Rule 2 states that a delay process can terminate if the argument of the delay process evaluates to zero. Rules 3 through 6 state that skip, $x := e$, $m\,!\,e$, and $m\,?\,x$ can perform their corresponding actions to the empty process. The stack is updated if necessary. If a programming variable is updated (in case of the assignment and receive process), the value is assumed to be of the same type. Note that the send process *sets* the value of a channel and the receive process *gets* the value of a channel. Besides their corresponding actions, the processes $m\,!\,e$ and $m\,?\,x$ can also perform a delay d. This is defined by Rule 7 and 8. Rules 9 describes transition behaviour of the delay

$$\frac{}{\langle\, \varepsilon, \sigma \,\rangle\downarrow}\,1 \qquad \frac{\sigma(e) = 0}{\langle\, \Delta e, \sigma \,\rangle\downarrow}\,2 \qquad \frac{}{\langle\, \mathsf{skip}, \sigma \,\rangle \xrightarrow{\tau} \langle\, \varepsilon, \sigma \,\rangle}\,3$$

$$\frac{\sigma(e) = c}{\langle\, x := e, \sigma \,\rangle \xrightarrow{aa(x,c)} \langle\, \varepsilon, \sigma[c/x] \,\rangle}\,4$$

$$\frac{\sigma(e) = c}{\langle\, m\,!\,e, \sigma \,\rangle \xrightarrow{sa(m,c)} \langle\, \varepsilon, \sigma[c/m] \,\rangle}\,5$$

$$\frac{\sigma(m) = c}{\langle\, m\,?\,x, \sigma \,\rangle \xrightarrow{ra(m,x)} \langle\, \varepsilon, \sigma[c/x] \,\rangle}\,6$$

$$\frac{}{\langle\, m\,!\,e, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, m\,!\,e, \sigma \,\rangle}\,7 \qquad \frac{}{\langle\, m\,?\,x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, m\,?\,x, \sigma \,\rangle}\,8$$

$$\frac{d \leq \sigma(e)}{\langle\, \Delta e, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e - d, \sigma \,\rangle}\,9$$

Table 4.1 · Deduction rules for χ_σ's atomic processes.

process. It states that a delay process can perform a delay bigger than zero, but smaller than or equal to the value of the argument of the delay process. Since the deadlock process δ denotes 'no behaviour', there are no deduction rules for it.

Using Definition 4.5 on process equivalence and Definition 4.8 above, we can now prove that the process $\Delta 0$ is bisimilar to process ε as stated in Lemma 4.9 below.

Lemma 4.9 · $\Delta 0 \leftrightarrows \varepsilon$.

Proof · (Lemma 4.9) We have to prove that $\Delta 0 \leftrightarrows \varepsilon$. In this case, we define a relation $R \subseteq P \times P$ such that $(\Delta 0, \varepsilon) \in R$ and R is a bisimulation.

We define R as

$$R = \{(\Delta 0, \varepsilon)\}$$

and show that pair $(\Delta 0, \varepsilon) \in R$ satisfies the five bisimulation conditions of Definition 4.5.

Condition 1: We have to prove $\forall \sigma : \langle\, \Delta 0, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, \varepsilon, \sigma \,\rangle\!\downarrow$. According to Rules 1 and 2 both $\langle\, \Delta 0, \sigma \,\rangle\!\downarrow$ and $\langle\, \varepsilon, \sigma \,\rangle\!\downarrow$ hold. Therefore, the condition holds.

Condition 2–5: The proof is trivial since the left-hand side of the implication does not hold. □

4.6 · Guard operator

In Definition 4.10, we define the guard operator ':→'. A process $e :\!\rightarrow p$ can behave like p if guard e evaluates to *true*.

Definition 4.10 · (Guard operator) *The guard operator has the following signature with* bool *the set of boolean expressions according to specification* BOOL *from Section 2.2:*

$$\_ :\!\rightarrow \_ \quad : \quad \text{bool} \times P \rightarrow P.$$

The deduction rules for the guard operator are listed in Table 4.2.

$$\frac{\sigma(e) = \mathit{true},\ \langle\, p, \sigma \,\rangle\!\downarrow}{\langle\, e :\!\rightarrow p, \sigma \,\rangle\!\downarrow}\,10 \qquad \frac{\sigma(e) = \mathit{true},\ \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, e :\!\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}\,11$$

$$\frac{\sigma(e) = \mathit{true},\ \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}{\langle\, e :\!\rightarrow p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}\,12$$

Table 4.2 · Deduction rules for the guarded operator.

Rule 10 states that a guarded process can terminate if the guard evaluates to *true* and if its process argument p can terminate. Rule 11 states that a guarded process can perform an action if the guard evaluates to *true* and if its process argument can perform that action. In the same way, Rule 12 states that a guarded process

can perform a delay if the guard evaluates to *true* and if its process argument can perform that delay.

Lemma 4.11 · *Let p be a process, then*

$$true :\rightarrow p \leftrightarrows p.$$

Proof · (Lemma 4.11) We have to prove that $true :\rightarrow p \leftrightarrows p$ for all processes p. In this case, we define a relation $R \subseteq P \times P$ such that $(true :\rightarrow p, p) \in R$ and R is a bisimulation. We define R as

$$R = \{(p, p)\} \cup \{(true :\rightarrow p, p)\}$$

and show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs are trivial for the pairs (p, p), we only give the proofs for the pairs $(true :\rightarrow p, p)$.

Condition 1: We have to prove $\forall \sigma : \langle\, true :\rightarrow p, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, p, \sigma \,\rangle\downarrow$. We first prove the right implication. Suppose $\langle\, true :\rightarrow p, \sigma \,\rangle\downarrow$, which means that Rule 10 applies. Therefore, we obtain $\langle\, p, \sigma \,\rangle\downarrow$. This concludes the right implication of Condition 1. For the left implication we find the following. Suppose $\langle\, p, \sigma \,\rangle\downarrow$. Since the guard equals *true*, by Rule 10 we obtain $\langle\, true :\rightarrow p, \sigma \,\rangle\downarrow$.

Condition 2: We have to prove $\forall \sigma, \sigma', q, a : \langle\, true :\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (q, r) \in R$. Suppose $\langle\, true :\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$, then Rule 11 applies. Therefore, we immediately obtain $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and consequently $(q, r) \in R$.

Condition 3: We have to prove $\forall \sigma, \sigma', q, a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, true :\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (r, q) \in R$. Suppose $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$, then, since the guard equals *true*, by Rule 11 we obtain $\langle\, true :\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and consequently $(r, q) \in R$.

Condition 4: We have to prove $\forall \sigma, \sigma', q, d : \langle\, true :\rightarrow p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle \wedge (q, r) \in R$. Suppose $\langle\, e :\rightarrow p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle$, which means that Rule 12 applies. So, we immediately obtain $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and consequently $(q, r) \in R$.

Condition 5: We have to prove $\forall \sigma, \sigma', q, d : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, e :\rightarrow p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle \wedge (r, q) \in R$. Suppose $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle$, then, since $e = true$, by Rule 12 we immediately obtain $\langle\, true :\rightarrow p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and consequently $(r, q) \in R$. □

Lemma 4.12 · *Let p be a process, then*

$$\mathit{false} :\rightarrow p \leftrightarrows \delta.$$

Proof · (Lemma 4.12) We have to prove that $\mathit{false} :\rightarrow p \leftrightarrows \delta$ for all processes p. In this case, we define a relation $R \subseteq P \times P$ such that $(\mathit{false} :\rightarrow p, \delta) \in R$ and R is a bisimulation. We define R as

$$R = \{(\mathit{false} :\rightarrow p, \delta)\}$$

and show that pair $(\mathit{false} :\rightarrow p, \delta) \in R$ satisfies the five bisimulation conditions of Definition 4.5.

Condition 1: We have to prove $\forall\sigma : \langle\, \mathit{false} :\rightarrow p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, \delta, \sigma \,\rangle\!\downarrow$. Since the guard equals *false*, no rule applies to $\langle\, \mathit{false} :\rightarrow p, \sigma \,\rangle$, which means we have $\langle\, \mathit{false} :\rightarrow p, \sigma \,\rangle\not\downarrow$. Also, no rules apply to $\langle\, \delta, \sigma \,\rangle$, which means we have $\langle\, \delta, \sigma \,\rangle\not\downarrow$.

Condition 2: We have to prove $\forall\sigma, \sigma', q, a : \langle\, \mathit{false} :\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, \delta, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (q, r) \in R$. Suppose $\langle\, \mathit{false} :\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$, then Rule 11 should apply and $\sigma(\mathit{false}) = \mathit{true}$ and $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$ should hold. However, since the guard equals *false*, $\sigma(\mathit{false})$ cannot be *true*, which means we have a contradiction. Therefore, $\langle\, \mathit{false} :\rightarrow p, \sigma \,\rangle \not\rightarrow$ and Condition 2 holds trivially.

Condition 3: We have to prove $\forall\sigma, \sigma', q, a : \langle\, \delta, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, \mathit{false} :\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (r, q) \in R$. Since there are no action transitions defined for δ, the condition holds trivially.

Condition 4: The proof is similar to the proof of Condition 2.

Condition 5: The proof is similar to the proof of Condition 3. □

4.7 · Alternative composition operator

In Definition 4.13, we define the alternative composition operator '$\|$'. With respect to action behaviour, a process $p \,\|\, q$ either executes p or q where the choice is non-deterministic. Delay behaviour is handled more subtly, because of time factorisation (see Section 4.3).

Definition 4.13 · (Alternative composition operator) *The alternative composition operator has the following signature:*

$$\_ \between \_ \quad : \quad P \times P \to P.$$

The deduction rules for the alternative composition operator are listed in Table 4.3.

$$\frac{\langle p, \sigma \rangle\downarrow}{\langle p \between q, \sigma \rangle\downarrow,\ \langle q \between p, \sigma \rangle\downarrow}\ 13 \qquad \frac{\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle}{\langle p \between q, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle,\ \langle q \between p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle}\ 14$$

$$\frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle,\ \langle q, \sigma \rangle \not\overset{d}{\longmapsto}}{\langle p \between q, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle,\ \langle q \between p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle}\ 15$$

$$\frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle,\ \langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle}{\langle p \between q, \sigma \rangle \overset{d}{\longmapsto} \langle p' \between q', \sigma' \rangle}\ 16$$

Table 4.3 · Deduction rules for the alternative composition operator.

Rule 13 states that an alternative composition of two processes p and q can terminate if one of the processes p or q can terminate. Rule 14 states that an alternative composition of two processes can perform an action if one of the two processes can perform that action. Rule 15 and 16 describe how the alternative composition of two processes delays. Rule 15 states that if one of the two processes can perform a delay and the other cannot, then the alternative composition can also perform that delay but loses the alternative that could not delay. On the other hand, if both processes can perform a delay, then its alternative composition can perform that delay too and both alternatives are preserved. This is stated in Rule 16.

Lemma 4.14 · *Let p be a process, then*

$$p \between \delta \leftrightarrows p.$$

Proof · (Lemma 4.14) We have to prove that $p \between \delta \leftrightarrows p$ for all processes P. In this case, we define a relation $R \subseteq P \times P$ such that $(p \between \delta, p) \in R$ and R is a bisimulation. We define R as

$$R = \{(p,p)\} \cup \{(p \parallel \delta, p)\}$$

and show that all pairs $(p,q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proof for pairs (p,p) is trivial, we will only consider pairs of the form $(p \parallel \delta, p)$. So, assume $p \equiv x \parallel \delta$ and $q \equiv x$ for some process x.

Condition 1: We have to prove $\forall\sigma : \langle\, p,\sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q,\sigma \,\rangle\!\downarrow$. The following computation shows that this holds $\langle\, p,\sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, x,\sigma \,\rangle\!\downarrow \vee \langle\, \delta,\sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, x,\sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q,\sigma \,\rangle\!\downarrow$.

Condition 2: We have to prove $\forall a, p_a, \sigma, \sigma' : \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p_a,\sigma' \,\rangle \Rightarrow \exists q_a : \langle\, q,\sigma \,\rangle \xrightarrow{a} \langle\, q_a,\sigma' \,\rangle$ and $(p_a,q_a) \in R$. So, assume $\langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p_a,\sigma' \,\rangle$, which means Rule 14 applies to p. Since there are no action transitions defined for δ, we must have $\langle\, x,\sigma \,\rangle \xrightarrow{a} \langle\, x_a,\sigma' \,\rangle$ and $p_a \equiv x_a$. Therefore, we get $\langle\, q,\sigma \,\rangle \xrightarrow{a} \langle\, x_a,\sigma' \,\rangle$. So , take $q_a \equiv x_a$ and note that $(p_a,q_a) \in R$.

Condition 3: We have to prove $\forall a, q_a, \sigma, \sigma' : \langle\, q,\sigma \,\rangle \xrightarrow{a} \langle\, q_a,\sigma' \,\rangle \Rightarrow \exists p_a : \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p_a,\sigma \,\rangle$. So, assume $\langle\, q,\sigma \,\rangle \xrightarrow{a} \langle\, q_a,\sigma' \,\rangle$. According to Rule 14, we obtain $\langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, q_a,\sigma' \,\rangle$. Finally, note that $(p_a,q_a) \in R$.

Condition 4: We have to prove $\forall d, p_d, \sigma, \sigma' : \langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d,\sigma' \,\rangle \Rightarrow \exists q_d : \langle\, q,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q_d,\sigma' \,\rangle$. So, assume $\langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d,\sigma' \,\rangle$. Since there are no delay transitions defined for δ, we must have $\langle\, x,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, x_d,\sigma' \,\rangle$ and $p_a \equiv x_a$. So, we also have $\langle\, q,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, x_d,\sigma' \,\rangle$. Therefore, we take $q_d \equiv x_d$ and note that $(p_d,q_d) \in R$.

Condition 5: We have to prove $\forall d, q_d, \sigma, \sigma' : \langle\, q,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q_d,\sigma' \,\rangle \Rightarrow \exists p_d : \langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d,\sigma' \,\rangle$. So, assume $\langle\, q,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q_d,\sigma' \,\rangle$. According to Rule 15, we obtain $\langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q_d,\sigma' \,\rangle$. Finally, note that $(p_d,q_d) \in R$. □

Lemma 4.15 · *Let p be a process, then*

$$p \parallel p \leftrightarrows p.$$

Proof · (Lemma 4.15) We have to prove that $p \parallel p \leftrightarrows p$ for any process p. The relevant deduction rules are Rules 13, 14, 15, and 16. Observing these rules, we notice the following. Whenever we can derive $\langle\, p \parallel p,\sigma \,\rangle\!\downarrow$, we can also derive $\langle\, p,\sigma \,\rangle\!\downarrow$ and vice versa. Whenever we can derive $\langle\, p \parallel p,\sigma \,\rangle \xrightarrow{a} \langle\, r,\sigma' \,\rangle$, we can also derive $\langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, r,\sigma' \,\rangle$ and vice versa. Whenever we can derive $\langle\, p \parallel p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, r,\sigma' \,\rangle$, we can also derive $\langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, r,\sigma' \,\rangle$ and vice versa. □

Lemma 4.16 · *Let p and q be processes, then*

$$p \parallel q \leftrightarrows q \parallel p.$$

Proof · (Lemma 4.16) We have to prove $p \parallel q \leftrightarrows q \parallel p$ for all processes p and q. The relevant deduction rules are Rules 13, 14, 15, and 16. Observing these rules, we see that whenever we can derive $\langle\, p \parallel q, \sigma \,\rangle\!\downarrow$, $\langle\, p \parallel q, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, or $\langle\, p \parallel q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, we can also derive $\langle\, q \parallel p, \sigma \,\rangle\!\downarrow$, $\langle\, q \parallel p, \sigma' \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, and $\langle\, q \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, respectively. □

Lemma 4.17 · *Let p, q, and r be processes, then*

$$(p \parallel q) \parallel r \leftrightarrows p \parallel (q \parallel r).$$

Proof · (Lemma 4.17) We have to prove $(p \parallel q) \parallel r \leftrightarrows p \parallel (q \parallel r)$ for all processes p, q, and r. In this case, we define a relation $R \subseteq P \times P$ such that $((p \parallel q) \parallel r, p \parallel (q \parallel r)) \in R$ and R is a bisimulation. We define R as

$$R = \{(p, p)\} \cup \{((p \parallel q) \parallel r, p \parallel (q \parallel r))\}$$

and show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proof for pairs of the form (p, p) is trivial, we will only consider pairs of the form $((p \parallel q) \parallel r, p \parallel (q \parallel r))$. Suppose $p \equiv (x \parallel y) \parallel z$ and $q \equiv x \parallel (y \parallel z)$ for some processes x, y, and z.

Condition 1: We have to prove $\forall \sigma : \langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\!\downarrow$. The following computation shows that this holds (we use Rule 13 and associativity of '$\vee$'):

$$\begin{array}{ll} & \langle\, p, \sigma \,\rangle\!\downarrow \\ \Leftrightarrow & (\langle\, x, \sigma \,\rangle\!\downarrow \vee \langle\, y, \sigma \,\rangle\!\downarrow) \vee \langle\, z, \sigma \,\rangle\!\downarrow \\ \Leftrightarrow & \langle\, x, \sigma \,\rangle\!\downarrow \vee (\langle\, y, \sigma \,\rangle\!\downarrow \vee \langle\, z, \sigma \,\rangle\!\downarrow) \\ \Leftrightarrow & \langle\, q, \sigma \,\rangle\!\downarrow. \end{array}$$

Condition 2: We have to prove $\forall a, p', \sigma, \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$ and $(p', q) \in R$. So, assume $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. This means that Rule 14 applies and therefore we have $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, p, \sigma' \,\rangle$, $\langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, or $\langle\, z, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. In either case, we can use Rule 14 to obtain $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. So, take $q' \equiv p'$ and note that $(p', q') \in R$.

Condition 3: The proof is similar to the proof of Condition 2.

Condition 4: We have to prove $\forall p', d, \sigma, \sigma' : \langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle p', \sigma' \rangle \Rightarrow \exists q' : \langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle q', \sigma' \rangle$ and $(p', q') \in R$. So, assume $\langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle p', \sigma' \rangle$. We distinguish the following cases.

- $\langle x, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle$, $\langle y, \sigma \rangle \stackrel{d}{\longmapsto} \langle y', \sigma' \rangle$, $\langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $p' \equiv (x' \parallel y') \parallel z'$. Using Rule 16 we obtain $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle x' \parallel (y' \parallel z'), \sigma' \rangle$. So, take $q' \equiv x' \parallel (y' \parallel z')$ and note that $(p', q') \in R$.
- $\langle x, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle$, $\langle y, \sigma \rangle \stackrel{d}{\longmapsto} \langle y', \sigma' \rangle$, $\langle z, \sigma \rangle \not\mapsto$: Note that $p' \equiv x' \parallel y'$. Using Rules 15 and 16 we obtain $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle x' \parallel y', \sigma' \rangle$. So, take $q' \equiv x' \parallel y'$ and note that $(p', q') \in R$.
- $\langle x, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle$, $\langle y, \sigma \rangle \not\mapsto$, $\langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $p' \equiv x' \parallel z'$. Using Rules 15 and 16 we obtain $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle x' \parallel z', \sigma' \rangle$. So, take $q' \equiv x' \parallel z'$ and note that $(p', q') \in R$.
- $\langle x, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle$, $\langle y, \sigma \rangle \not\mapsto$, $\langle z, \sigma \rangle \not\mapsto$: Note that $p' \equiv x'$. Using Rule 15 we obtain $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle$. So, take $q' \equiv x'$ and note that $(p', q') \in R$.
- $\langle x, \sigma \rangle \not\mapsto$, $\langle y, \sigma \rangle \stackrel{d}{\longmapsto} \langle y', \sigma' \rangle$, $\langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $p' \equiv y' \parallel z'$. Using Rules 15 and 16 we obtain $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle y' \parallel z', \sigma' \rangle$. So, take $q' \equiv y' \parallel z'$ and note that $(p', q') \in R$.
- $\langle x, \sigma \rangle \not\mapsto$, $\langle y, \sigma \rangle \stackrel{d}{\longmapsto} \langle y', \sigma' \rangle$, $\langle z, \sigma \rangle \not\mapsto$: Note that $p' \equiv y'$. Using Rule 15 we obtain $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle y', \sigma' \rangle$. So, take $q' \equiv y'$ and note that $(p', q') \in R$.
- $\langle x, \sigma \rangle \not\mapsto$, $\langle y, \sigma \rangle \not\mapsto$, $\langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $p' \equiv z'$. Using Rule 15 we obtain $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$. So, take $q' \equiv z'$ and note that $(p', q') \in R$.

Condition 5: The proof is similar to the proof of Condition 4. □

4.8 · Sequential composition operator

In Definition 4.18, we define the sequential composition operator ';'. With respect to action behaviour, a process p ; q first executes p and once terminated successfully executes q. As with the alternative composition operator discussed in the previous section, also here delay behaviour is handled more subtly, because of time factorisation.

Definition 4.18 · (Sequential composition operator) *The sequential composition operator has the following signature:*

$$\_ \,;\, \_ \;:\; P \times P \to P.$$

The deduction rules for the sequential composition operator are listed in Table 4.4.

$$\frac{\langle\, p,\sigma\,\rangle\!\downarrow,\ \langle\, q,\sigma\,\rangle\!\downarrow}{\langle\, p\ ;\ q,\sigma\,\rangle\!\downarrow}\,17 \qquad \frac{\langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle}{\langle\, p\ ;\ q,\sigma\,\rangle \xrightarrow{a} \langle\, p'\ ;\ q,\sigma'\,\rangle}\,18$$

$$\frac{\langle\, p,\sigma\,\rangle\!\downarrow,\ \langle\, q,\sigma\,\rangle \xrightarrow{a} \langle\, q',\sigma'\,\rangle}{\langle\, p\ ;\ q,\sigma\,\rangle \xrightarrow{a} \langle\, q',\sigma'\,\rangle}\,19 \qquad \frac{\langle\, p,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, p',\sigma'\,\rangle,\ \langle\, p,\sigma\,\rangle\not\!\downarrow}{\langle\, p\ ;\ q,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, p'\ ;\ q,\sigma'\,\rangle}\,20$$

$$\frac{\langle\, p,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, p',\sigma'\,\rangle,\ \langle\, q,\sigma\,\rangle\not\longmapsto}{\langle\, p\ ;\ q,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, p'\ ;\ q,\sigma'\,\rangle}\,21 \qquad \frac{\langle\, p,\sigma\,\rangle\!\downarrow,\ \langle\, q,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, q',\sigma'\,\rangle,\ \langle\, p,\sigma\,\rangle\not\longmapsto}{\langle\, p\ ;\ q,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, q',\sigma'\,\rangle}\,22$$

$$\frac{\langle\, p,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, p',\sigma'\,\rangle,\ \langle\, p,\sigma\,\rangle\!\downarrow,\ \langle\, q,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, q',\sigma'\,\rangle}{\langle\, p\ ;\ q,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, (p'\ ;\ q) \between q',\sigma'\,\rangle}\,23$$

Table 4.4 · Deduction rules for the sequential composition operator.

Rule 17 states that if processes p and q can terminate, then also the sequential composition of $p\,;q$ can. Rule 18 states that if process p can perform an action, then the sequential composition $p\ ;\ q$ can also perform the action. Rule 19 states that if process p terminates and process q can perform an action, then the sequential composition $p\,;q$ can also perform the action. The delay behaviour of the sequential composition operator is quite intricate. The reason for this is that we want to make sure that all process operators exhibit time factorisation. As a consequence, we distinguish three different cases for a sequential composition $p\ ;\ q$. In the first case, only the delay behaviour of p is relevant, as defined by Rule 20 and 21. Note that possible delay behaviour of q is irrelevant since either p cannot terminate (Rule 20) or q cannot delay (Rule 21). In the second case, only the delay behaviour of q is relevant since p can terminate and cannot delay (Rule 22). Finally, if both p and q can delay and p can also terminate, then they have to delay together (Rule 23).

Lemma 4.19 · *Let p be a process, then*

$$p\ ;\ \varepsilon \leftrightarrows p.$$

Proof · (Lemma 4.19) We have to prove that $p\ ;\ \varepsilon \leftrightarrows p$ for all processes p. In this case, we define a relation $R \subseteq P \times P$ such that $(p\ ;\ \varepsilon, p) \in R$ and R is a

bisimulation. We define R as

$$R = \{(p,p)\} \cup \{(p\ ;\ \varepsilon, p)\}$$

and show that all pairs $(p,q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs are trivial for the pairs (p,p), we only give the proofs for the pairs $(p\ ;\ \varepsilon, p)$.

Condition 1: We have to prove $\forall \sigma : \langle\, p\ ;\ \varepsilon, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, \sigma \,\rangle\!\downarrow$. Let us first prove the right implication of Condition 1. Suppose $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle\!\downarrow$. Then Rule 17 should apply and we obtain $\langle\, p, \sigma \,\rangle\!\downarrow$. This concludes the right implication of Condition 1. For the left implication we find the following. Suppose $\langle\, p, \sigma \,\rangle\!\downarrow$. Rule 1 combined with Rule 17 then gives $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle\!\downarrow$.

Condition 2: We have to prove $\forall \sigma, \sigma', q, a : \langle\, p\ ;\ \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (q,r) \in R$. Suppose $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$. The relevant Rules are 18 and 19. However, since there are no action transitions defined for ε, the second rule does not apply. So, Rule 18 applies and we obtain $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, p'\ ;\ \varepsilon, \sigma' \,\rangle$ and $q \equiv p'\ ;\ \varepsilon$. We also have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$ where $r \equiv p'$ and $(q,r) \in R$.

Condition 3: We have to prove $\forall \sigma, \sigma', q, a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p\ ;\ \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (r,q) \in R$. Suppose $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$, then, by Rule 18 we also have $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, where $r \equiv q\ ;\ \varepsilon$ and $(r,q) \in R$.

Condition 4: We have to prove $\forall \sigma, \sigma', q, d : \langle\, p\ ;\ \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle \wedge (q,r) \in R$. Suppose $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle$. The relevant Rules are 20, 21, 22, and 23. However, since there are no delay transitions defined for ε, the third and fourth rule do not apply. So, by either Rule 20 or Rule 21, we obtain $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p'\ ;\ \varepsilon, \sigma' \,\rangle$ and $q \equiv p'\ ;\ \varepsilon$. In both cases, we also have $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, where $r \equiv p'$ and $(q,r) \in R$.

Condition 5: We have to prove $\forall \sigma, \sigma', q, d : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p\ ;\ \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle \wedge (r,q) \in R$. Suppose $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle$, then, by Rule 20 or 21 we have $\langle\, p\ ;\ \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, where $r \equiv q\ ;\ \varepsilon$ and $(r,q) \in R$. □

Lemma 4.20 · *Let p be a process, then*

$$\varepsilon\ ;\ p \leftrightarrows p.$$

Proof · (Lemma 4.20) We have to prove that $\varepsilon\ ;\ p \leftrightarrows p$ for all processes p. In this case, we define a bisimulation relation $R \subseteq P \times P$ such that $(\varepsilon\ ;\ p, p) \in R$ and R

is a bisimulation. We define R as

$$R = \{(p,p)\} \cup \{(\varepsilon \ ; \ p,p)\}$$

and show that all pairs $(p,q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs are trivial for the pairs (p,p), we only give the proofs for the pairs $(\varepsilon \ ; \ p,p)$.

Condition 1: We have to prove $\forall\sigma : \langle\, \varepsilon \ ; \ p,\sigma \,\rangle\downarrow \Leftrightarrow \langle\, p,\sigma \,\rangle\downarrow$. Let us first prove the right implication of Condition 1. Suppose $\langle\, \varepsilon \ ; \ p,\sigma \,\rangle\downarrow$. Then Rule 17 should apply and we obtain $\langle\, p,\sigma \,\rangle\downarrow$. For the left implication we find the following. Suppose $\langle\, p,\sigma \,\rangle\downarrow$. Rule 1 combined with Rule 17 then gives $\langle\, \varepsilon \ ; \ p,\sigma \,\rangle\downarrow$.

Condition 2: We have to prove $\forall\sigma,\sigma',q,a : \langle\, \varepsilon \ ; \ p,\sigma \,\rangle \xrightarrow{a} \langle\, q,\sigma' \,\rangle \Rightarrow \exists r : \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, r,\sigma' \,\rangle \wedge (q,r) \in R$. Suppose $\langle\, \varepsilon \ ; \ p,\sigma \,\rangle \xrightarrow{a} \langle\, q,\sigma' \,\rangle$, then, Rule 19 should apply, since there are no action transitions defined for ε. So, we obtain $\langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, q,\sigma' \,\rangle$. In that case, we also have $\langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, r,\sigma' \,\rangle$, where $r \equiv q$ and $(q,r) \in R$.

Condition 3: We have to prove $\forall\sigma,\sigma',q,a : \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, q,\sigma' \,\rangle \Rightarrow \exists r : \langle\, \varepsilon \ ; \ p,\sigma \,\rangle \xrightarrow{a} \langle\, r,\sigma' \,\rangle \wedge (r,q) \in R$. Suppose $\langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, q,\sigma' \,\rangle$, then, by Rule 1 and 19 we also have $\langle\, \varepsilon \ ; \ p,\sigma \,\rangle \xrightarrow{a} \langle\, r,\sigma' \,\rangle$, where $r \equiv q$ and $(r,q) \in R$.

Condition 4: We have to prove $\forall\sigma,\sigma',q,d : \langle\, \varepsilon \ ; \ p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q,\sigma' \,\rangle \Rightarrow \exists r : \langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, r,\sigma' \,\rangle \wedge (q,r) \in R$. Suppose $\langle\, \varepsilon \ ; \ p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q,\sigma' \,\rangle$, then Rule 22 should apply, since there are no delay transitions defined for ε. So, we obtain $\langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q,\sigma' \,\rangle$. In that case, we also have $\langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, r,\sigma' \,\rangle$, where $r \equiv q$ and $(q,r) \in R$.

Condition 5: We have to prove $\forall\sigma,\sigma',q,d : \langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q,\sigma' \,\rangle \Rightarrow \exists r : \langle\, \varepsilon \ ; \ p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, r,\sigma' \,\rangle \wedge (r,q) \in R$. Suppose $\langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, q,\sigma' \,\rangle$, then, by Rule 1 and 22 we also have $\langle\, \varepsilon \ ; \ p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, r,\sigma' \,\rangle$, where $r \equiv q$ and $(r,q) \in R$. □

Lemma 4.21 · *Let p be a process, then*

$$\delta \ ; \ p \leftrightarrows \delta.$$

Proof · (Lemma 4.21) We have to prove that $\delta \ ; \ p \leftrightarrows \delta$ for all processes p. In this case, we define a relation $R \subseteq P \times P$ such that $(\delta \ ; \ p,\delta) \in R$ and R is a bisimulation. We define R as

$$R = \{(p,p)\} \cup \{(\delta \ ; \ p,\delta)\}$$

and show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since there are no terminations, action transitions, and delay transitions defined for δ, and therefore also not for $\delta \ ; \ p$, the five conditions hold trivially. □

Lemma 4.22 · *Let p, q, and r be processes, then*

$$(p \ ; \ q) \ ; \ r \leftrightarrows p \ ; \ (q \ ; \ r).$$

Proof · (Lemma 4.22) We have to prove that $(p \ ; \ q) \ ; \ r \leftrightarrows p \ ; \ (q \ ; \ r)$ for all processes p, q, and r. In this case, we define a relation $R \subseteq P \times P$ such that $((p \ ; \ q) \ ; \ r, p \ ; \ (q \ ; \ r)) \in R$ and R is a 'bisimulation up to $\leftrightarrows$' (see Definition 4.6). We define R as the least set satisfying

$$\begin{aligned} R &= R_0 \cup R_1 \cup R_2, \\ R_0 &= \{(p, p)\}, \\ R_1 &= \{((p \ ; \ q) \ ; \ r, p \ ; \ (q \ ; \ r))\}, \\ R_2 &= \{(p \parallel r, q \parallel r) \mid (p, q) \in R\} \end{aligned}$$

and show that all pairs $(p, q) \in R$ satisfy the conditions of Definition 4.6. During the proof it will become clear why R includes R_2. Since R is defined recursively, we will use structural induction on the elements of R. The proof consists of two parts. For the basis of the proof, we show that the five 'bisimulation up to $\leftrightarrows$' conditions hold for pairs $(p, q) \in R_0 \cup R_1$. For the inductive step, we assume the five 'bisimulation up to $\leftrightarrows$' conditions hold for $(p, q) \in R$ and show they hold for pairs $(p \parallel r, q \parallel r) \in R_2$.

Basis Since the proofs of the pairs of the form $(p, q) \in R_0$ are trivial, we will only consider the pairs $(p, q) \in R_1$. So, suppose $(p, q) \in R_1$ and $p \equiv (x \ ; \ y) \ ; \ z$ and $q \equiv x \ ; \ (y \ ; \ z)$ for some processes x, y, and z.

Condition 1: We have to prove $\forall \sigma : \langle\, p, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\downarrow$. Using Rule 17 multiple times, for the left-hand side we obtain $(\langle\, x, \sigma \,\rangle\downarrow \wedge \langle\, y, \sigma \,\rangle\downarrow) \wedge \langle\, z, \sigma \,\rangle\downarrow$ and for the right-hand side we obtain $\langle\, x, \sigma \,\rangle\downarrow \wedge (\langle\, y, \sigma \,\rangle\downarrow \wedge \langle\, z, \sigma \,\rangle\downarrow)$. Since the operator '$\wedge$' is associative, we are done.

Condition 2: We have to prove that $\forall a, p_a, \sigma, \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma' \,\rangle \Rightarrow \exists p'_a, q_a, q'_a :$ $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q_a, \sigma' \,\rangle$, $p_a \leftrightarrows p'_a$, $(p'_a, q'_a) \in R$, and $q'_a \leftrightarrows q_a$. So, suppose $\langle\, p, \sigma \,\rangle \xrightarrow{a}$ $\langle\, p_a, \sigma' \,\rangle$. We distinguish three cases.

$\langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma' \rangle$: Note that we have $p_a \equiv (x_a \ ; y) \ ; z$. Using Rule 18, we also find $\langle q, \sigma \rangle \xrightarrow{a} \langle x_a \ ; (y \ ; z), \sigma' \rangle$. Therefore, we take $p'_a \equiv p_a$, $q_a \equiv x_a \ ; (y \ ; z)$, and $q'_a \equiv q_a$. Note that $(p'_a, q'_a) \in R$.

$\langle x, \sigma \rangle\downarrow \wedge \langle y, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma \rangle$: Note that $p_a \equiv y_a \ ; z$. Using Rules 19 and 18, we obtain $\langle q, \sigma \rangle \xrightarrow{a} \langle y_a \ ; z, \sigma' \rangle$. Therefore, we take $p'_a \equiv p_a$, $q_a \equiv y_a \ ; z$, and $q'_a \equiv q_a$. Note that $(p'_a, q'_a) \in R$.

$\langle x, \sigma \rangle\downarrow \wedge \langle y, \sigma \rangle\downarrow \wedge \langle z, \sigma \rangle \xrightarrow{a} \langle z_a, \sigma' \rangle$: Note that $p_a \equiv z_a$. Using Rule 19 two times, we obtain $\langle q, \sigma \rangle \xrightarrow{a} \langle z_a, \sigma' \rangle$. Therefore, we take $p'_a \equiv p_a$, $q_a \equiv z_a$, and $q'_a \equiv q_a$. Note that $(p'_a, q'_a) \in R$.

Condition 3: The proof is similar to the proof of Condition 2.

Condition 4: We have to prove that $\forall d, p_d, \sigma, \sigma' : \langle p, \sigma \rangle \xmapsto{d} \langle p_d, \sigma' \rangle \Rightarrow \exists p'_d, q_d, q'_d :$ $\langle q, \sigma \rangle \xmapsto{d} \langle q_d, \sigma' \rangle$, $p_d \leftrightarrows p'_d$, $(p'_d, q'_d) \in R$, and $q'_d \leftrightarrows q_d$. So, assume $\langle p, \sigma \rangle \xmapsto{d} \langle p_d, \sigma' \rangle$. We distinguish three cases.

$\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma' \rangle \wedge \langle x, \sigma \rangle \not\downarrow$: Note that $p_d \equiv (x_d \ ; y) \ ; z$. According to Rule 20, we also have $\langle q, \sigma \rangle \xmapsto{d} \langle x_d \ ; (y \ ; z), \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv x_d \ ; (y \ ; z)$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.

$\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma' \rangle \wedge \langle x, \sigma \rangle\downarrow$: We distinguish three cases.

$\langle y, \sigma \rangle \xmapsto{d} \langle y_d, \sigma' \rangle \wedge (\langle y, \sigma \rangle \not\downarrow \vee \langle z, \sigma \rangle \mapsto)$: Note that $p_d \equiv (x_d \ ; y) \ ; z \parallel y_d \ ; z$. According to Rules 20, 21, and 23, we obtain $\langle q, \sigma \rangle \xmapsto{d} \langle x_d \ ; (y \ ; z) \parallel y_d \ ; z, \sigma' \rangle$. Therefore, we take $p'_d \equiv p_d$, $q_d \equiv x_d \ ; (y \ ; z) \parallel y_d \ ; z$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.

$\langle y, \sigma \rangle \xmapsto{d} \langle y_d, \sigma' \rangle \wedge \langle y, \sigma \rangle\downarrow \wedge \langle z, \sigma \rangle \xmapsto{d} \langle z_d, \sigma' \rangle$: Note that $p_d \equiv (x_d \ ; y \parallel y_d) \ ; z \parallel z_d$. According to Rule 23 we obtain $\langle q, \sigma \rangle \xmapsto{d} \langle x_d \ ; (y \ ; z) \parallel (y_d \ ; z \parallel z_d), \sigma' \rangle$. So, take $q_d \equiv x_d \ ; (y \ ; z) \parallel (y_d \ ; z \parallel z_d)$. Using Lemmas 4.17 and 4.23, we can make the following computation:

$$\begin{array}{ll} & p_d \\ \equiv & (x_d \ ; y \parallel y_d) \ ; z \parallel z_d \\ \leftrightarrows & ((x_d \ ; y) \ ; z \parallel y_d \ ; z) \parallel z_d \\ \leftrightarrows & (x_d \ ; y) \ ; z \parallel (y_d \ ; z \parallel z_d). \end{array}$$

So, we define $p'_d \equiv (x_d \ ; y) \ ; z \parallel (y_d \ ; z \parallel z_d)$ and take $q'_d \equiv q_d$. It is clear that $(p'_d, q'_d) \in R$. Note that here we actually see that R is a 'bisimulation up to $\leftrightarrows$' relation; it is *not* a bisimulation, since $(p_d, q_d) \notin R$.

$\langle y, \sigma \rangle \mapsto \wedge \langle y, \sigma \rangle\downarrow \wedge \langle z, \sigma \rangle \xmapsto{d} \langle z_d, \sigma' \rangle$: Note that $p_d \equiv (x_d \ ; y) \ ; z \parallel z_d$. According to Rules 22 and 23 we obtain $\langle q, \sigma \rangle \xmapsto{d} \langle x_d \ ; (y \ ; z) \parallel z_d, \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv x_d \ ; (y \ ; z) \parallel z_d$, and $q'_d \equiv q_d$. It is clear that $(p'_d, q'_d) \in R$.

$\langle x, \sigma \rangle \mapsto \wedge \langle x, \sigma \rangle \downarrow$: We distinguish three cases.

- $\langle y, \sigma \rangle \xrightarrow{d} \langle y_d, \sigma' \rangle \wedge (\langle y, \sigma \rangle \not\downarrow \vee \langle z, \sigma \rangle \mapsto)$: Note that $p_d \equiv y_d \ ; z$. According to Rules 20, 21, and 22 we obtain $\langle q, \sigma \rangle \xrightarrow{d} \langle y_d \ ; z, \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv y_d \ ; z$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.
- $\langle y, \sigma \rangle \xrightarrow{d} \langle y_d, \sigma' \rangle \wedge \langle y, \sigma \rangle \downarrow \wedge \langle z, \sigma \rangle \xrightarrow{d} \langle z_d, \sigma' \rangle$: Note that $p_d \equiv y_d \ ; z \parallel z_d$. According to Rules 22 and 23 we obtain $\langle q, \sigma \rangle \xrightarrow{d} \langle y_d \ ; z \parallel z_d, \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv y_d \ ; z \parallel z_d$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.
- $\langle y, \sigma \rangle \mapsto \wedge \langle y, \sigma \rangle \downarrow \wedge \langle z, \sigma \rangle \xrightarrow{d} \langle z_d, \sigma' \rangle$: Note that $p_d \equiv z_d$. According to Rule 22 we obtain $\langle q, \sigma \rangle \xrightarrow{d} \langle z_d, \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv z_d$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.

Condition 5: The proof is similar to the proof of Condition 4.

This concludes the proof of pairs $(p, q) \in R_1$.

Inductive step We prove the five 'bisimulation up to $\underline{\leftrightarrow}$' conditions for all $(p, q) \in R_2$. So, suppose $p \equiv x \parallel z$ and $q \equiv y \parallel z$ and $(x, y) \in R$ for some processes x, y, and z. Furthermore, the induction hypothesis says that the five 'bisimulation up to $\underline{\leftrightarrow}$' conditions hold for (x, y).

Condition 1: We have to prove $\forall \sigma : \langle p, \sigma \rangle \downarrow \Leftrightarrow \langle q, \sigma \rangle \downarrow$. Using the induction hypothesis and Rule 13, this can be proven as follows:

$$\begin{array}{ll} & \langle p, \sigma \rangle \downarrow \\ \equiv & \langle x \parallel z, \sigma \rangle \downarrow \\ \Leftrightarrow & \langle x, \sigma \rangle \downarrow \vee \langle z, \sigma \rangle \downarrow \\ \Leftrightarrow & \langle y, \sigma \rangle \downarrow \vee \langle z, \sigma \rangle \downarrow \\ \Leftrightarrow & \langle y \parallel z, \sigma \rangle \downarrow \\ \equiv & \langle q, \sigma \rangle \downarrow. \end{array}$$

Condition 2: We have to prove $\forall a, p_a, \sigma, \sigma' : \langle p, \sigma \rangle \xrightarrow{a} \langle p_a, \sigma' \rangle \Rightarrow \exists p'_a, q_a, q'_a : \langle q, \sigma \rangle \xrightarrow{a} \langle q_a, \sigma' \rangle$, $p_a \underline{\leftrightarrow} p'_a$, $(p'_a, q'_a) \in R$, and $q'_a \underline{\leftrightarrow} q_a$. Suppose $\langle p, \sigma \rangle \xrightarrow{a} \langle p_a, \sigma' \rangle$. Then Rule 14 applies and we distinguish two cases.

- $\langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma' \rangle$: Note that $p_a \equiv x_a$. Using Condition 2 of the induction hypothesis on x we obtain $\exists y_a : \langle y, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma' \rangle$. According to Rule 14, we obtain $\langle q, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma' \rangle$. So, we take $p'_a \equiv p_a$, $q_a \equiv y_a$, and $q'_a \equiv q_a$. Note that $(p'_a, q'_a) \in R$.

$\langle z, \sigma \rangle \xrightarrow{a} \langle z_a, \sigma' \rangle$: Note that $p_a \equiv z_a$. According to Rule 14, we obtain $\langle q, \sigma \rangle \xrightarrow{a} \langle z_a, \sigma' \rangle$. So, we take $p'_a \equiv p_a$, $q_a \equiv z_a$, and $q'_a \equiv q_a$. Note that $(p'_a, q'_a) \in R$.

Condition 3: The proof is similar to the proof of Condition 2.

Condition 4: We have to prove $\forall d, p_d, \sigma, \sigma' : \langle p, \sigma \rangle \xmapsto{d} \langle p_d, \sigma' \rangle \Rightarrow \exists p'_d, q_d, q'_d : \langle q, \sigma \rangle \xmapsto{d} \langle q_d, \sigma' \rangle$, $p_d \leftrightarrows p'_d$, $(p'_d, q'_d) \in R$, and $q'_d \leftrightarrows q_d$. Suppose $\langle p, \sigma \rangle \xmapsto{d} \langle p_d, \sigma' \rangle$. We distinguish three cases.

- $\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma' \rangle \wedge \langle z, \sigma \rangle \not\mapsto$: In this case, Rule 15 applies to $\langle p, \sigma \rangle$. Note that $p_d \equiv x_d$. Using Condition 4 of the induction hypothesis on x, we obtain $\exists y_d : \langle y, \sigma \rangle \xmapsto{d} \langle y_d, \sigma' \rangle$. According to Rule 15, we have $\langle q, \sigma \rangle \xmapsto{d} \langle y_d, \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv y_d$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.
- $\langle z, \sigma \rangle \xmapsto{d} \langle z_d, \sigma' \rangle \wedge \langle x, \sigma \rangle \not\mapsto$: In this case, Rule 15 applies to $\langle p, \sigma \rangle$. Note that $p_d \equiv z_d$. Rule 15, we have $\langle q, \sigma \rangle \xmapsto{d} \langle z_d, \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv z_d$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.
- $\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma' \rangle \wedge \langle z, \sigma \rangle \xmapsto{d} \langle z_d, \sigma' \rangle$: Note that $p_d \equiv x_d \parallel z_d$. Using Condition 4 of the induction hypothesis on x, we obtain $\exists y_d : \langle y, \sigma \rangle \xmapsto{d} \langle y_d, \sigma' \rangle$. According to Rule 16 we obtain $\langle q, \sigma \rangle \xmapsto{d} \langle y_d \parallel z_d, \sigma' \rangle$. So, we take $p'_d \equiv p_d$, $q_d \equiv y_d \parallel z_d$, and $q'_d \equiv q_d$. Note that $(p'_d, q'_d) \in R$.

Condition 5: The proof is similar to the proof of Condition 4. □

Lemma 4.23 · *Let p, q, and r be processes, then*

$$(p \parallel q) \;;\; r \leftrightarrows p \;;\; r \parallel q \;;\; r.$$

Proof · (Lemma 4.23) We have to prove $(p \parallel q) \;;\; r \leftrightarrows p \;;\; r \parallel q \;;\; r$ for all processes p, q, and r. In this case, we define a relation $R \subseteq P \times P$ such that $((p \parallel q) \;;\; r, p \;;\; r \parallel q \;;\; r) \in R$ and R is a 'bisimulation up to $\leftrightarrows$' (see Definition 4.6). We define R as the least set satisfying

$$\begin{aligned} R &= R_0 \cup R_1 \cup R_2, \\ R_0 &= \{(p, p)\}, \\ R_1 &= \{((p \parallel q) \;;\; r, p \;;\; r \parallel q \;;\; r)\}, \\ R_2 &= \{(p \parallel r, q \parallel r) \mid (p, q) \in R\} \end{aligned}$$

and show that all pairs $(p, q) \in R$ satisfy the five conditions of Definition 4.6 During the proof it will become clear why R includes R_2. Since R is defined recursively,

we will use structural induction on the elements of R. The proof consists of two parts.

For the basis, we prove that the five 'bisimulation up to $\underline{\leftrightarrow}$' conditions hold for pairs $(p, q) \in R_0 \cup R_1$. For the inductive step, we assume the five 'bisimulation up to $\underline{\leftrightarrow}$' conditions hold for $(p, q) \in R$ and show they hold for pairs $(p \parallel r, q \parallel r) \in R_2$.

Basis Since the proof of the pairs $(p, q) \in R_0$ is trivial, we will only consider the pairs $(p, q) \in R_1$. So, suppose $(p, q) \in R_1$. Therefore, there are processes x, y, and z such that $p \equiv (x \parallel y)\ ;\ z$ and $q \equiv x\ ;\ z \parallel y\ ;\ z$.

Condition 1: We have to prove $\forall\sigma : \langle\, p, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\downarrow$. The following computation shows that this holds (we use Rules 13 and 17):

$$\begin{array}{ll} & \langle\, p, \sigma \,\rangle\downarrow \\ \Leftrightarrow & \langle\, x \parallel y, \sigma \,\rangle\downarrow \wedge \langle\, z, \sigma \,\rangle\downarrow \\ \Leftrightarrow & (\langle\, x, \sigma \,\rangle\downarrow \vee \langle\, y, \sigma \,\rangle\downarrow) \wedge \langle\, z, \sigma \,\rangle\downarrow \\ \Leftrightarrow & (\langle\, x, \sigma \,\rangle\downarrow \wedge \langle\, z, \sigma \,\rangle\downarrow) \vee (\langle\, y, \sigma \,\rangle\downarrow \wedge \langle\, z, \sigma \,\rangle\downarrow) \\ \Leftrightarrow & \langle\, x\ ;\ z, \sigma \,\rangle\downarrow \vee \langle\, y\ ;\ z, \sigma \,\rangle\downarrow \\ \Leftrightarrow & \langle\, q, \sigma \,\rangle\downarrow. \end{array}$$

Condition 2: We have to prove $\forall p', a, \sigma, \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow \exists p'', q', q'' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$, $p' \underline{\leftrightarrow} p''$, $(p'', q'') \in R$, and $q'' \underline{\leftrightarrow} q'$. So, assume $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. We distinguish the following cases.

- $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x', \sigma' \,\rangle$: Note that $p' \equiv x'\ ;\ z$. According to Rule 14 we have $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, x'\ ;\ z, \sigma' \,\rangle$. So, we take $p'' \equiv p'$, $q' \equiv x'\ ;\ z$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.
- $\langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, y', \sigma' \,\rangle$: Note that $p' \equiv y'\ ;\ z$. According to Rule 14 we have $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, y'\ ;\ z, \sigma' \,\rangle$. So, we take $p'' \equiv p'$, $q' \equiv y'\ ;\ z$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.
- $\langle\, x \parallel y, \sigma \,\rangle\downarrow \wedge \langle\, z, \sigma \,\rangle \xrightarrow{a} \langle\, z', \sigma' \,\rangle$: Note that $p' \equiv z'$. Based on Rule 13 we have $\langle\, x, \sigma \,\rangle\downarrow$ or $\langle\, y, \sigma \,\rangle\downarrow$. According to Rules 19 and 14 we have $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, z', \sigma' \,\rangle$. So, take $p'' \equiv p'$, $q' \equiv z'$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

Condition 3: We have to prove $\forall q', a, \sigma, \sigma' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p', p'', q'' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, $p' \underline{\leftrightarrow} p''$, $(p'', q'') \in R$, and $q'' \underline{\leftrightarrow} q'$. So, assume $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$. Based on Rule 14 we can distinguish two cases.

- $\langle\, x\ ;\ z, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$: Based on Rules 18 and 19 we can distinguish two cases.

$\langle x, \sigma \rangle \xrightarrow{a} \langle x', \sigma' \rangle$: Note that $q' \equiv x' \ ; \ z$. According to Rules 14 and 18 we have $\langle p, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle$. So, take $p' \equiv q'$, $p'' \equiv p'$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle\downarrow \wedge \langle z, \sigma \rangle \xrightarrow{a} \langle z', \sigma' \rangle$: Note that $q' \equiv z'$. According to Rules 13 and 19 we have $\langle p, \sigma \rangle \xrightarrow{a} \langle z', \sigma' \rangle$. So, take $p' \equiv z'$, $p'' \equiv p'$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

$\langle y \ ; \ z, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle$: The proof is similar to the proof of the previous case.

Condition 4: We have to prove $\forall p', d, \sigma, \sigma' : \langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle \Rightarrow \exists p'', q', q'' : \langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle$, $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$. So, assume $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$. We distinguish two cases.

$\langle x, \sigma \rangle \overset{d}{\longmapsto} \langle x', \sigma' \rangle$: We distinguish two cases.

$\langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y', \sigma' \rangle$: We distinguish the following cases.

$(\langle x, \sigma \rangle\downarrow \vee \langle y, \sigma \rangle\downarrow) \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $p' \equiv (x' \parallel y') \ ; \ z \parallel z'$. According to Rules 23 and 15 we have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle (x' \ ; \ z \parallel z') \parallel (y' \ ; \ z \parallel z'), \sigma' \rangle$. So, take $q' \equiv (x' \ ; \ z \parallel z') \parallel (y' \ ; \ z \parallel z')$. According to Lemmas 4.15, 4.16, and 4.17 we can make the following computation: $q' \equiv (x' \ ; \ z \parallel z') \parallel (y' \ ; \ z \parallel z') \leftrightarrows (x' \ ; \ z \parallel y' \ ; \ z) \parallel z'$ So, we define $q'' \equiv (x' \ ; \ z \parallel y' \ ; \ z) \parallel z'$ and take $p'' \equiv p'$. Given the definition of R_1 and R_2, it is clear that $(p'', q'') \in R$.

$(\langle x, \sigma \rangle\not\downarrow \wedge \langle y, \sigma \rangle\not\downarrow) \vee \langle z, \sigma \rangle\mapsto$: Note that $p' \equiv (x' \parallel y') \ ; \ z$. According to Rules 20, 21, and 16 we have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle x' \ ; \ z \parallel y' \ ; \ z, \sigma' \rangle$. So, we take $p'' \equiv p'$, $q' \equiv x' \ ; \ z \parallel y' \ ; \ z$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

$\langle y, \sigma \rangle\mapsto$: We distinguish the following cases.

$(\langle x, \sigma \rangle\downarrow \vee \langle y, \sigma \rangle\downarrow) \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $p' \equiv x' \ ; \ z \parallel z'$. According to Rules 22, 23, and 16 we have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle x' \ ; \ z \parallel z', \sigma' \rangle$. So, we take $p'' \equiv p'$, $q' \equiv x' \ ; \ z \parallel z'$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

$(\langle x, \sigma \rangle\not\downarrow \wedge \langle y, \sigma \rangle\not\downarrow) \vee \langle z, \sigma \rangle\mapsto$: Note that $p' \equiv x' \ ; \ z$. According to Rules 20, 21, and 15 we have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle x' \ ; \ z, \sigma' \rangle$. So, we take $p'' \equiv p'$, $q' \equiv x' \ ; \ z$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle\mapsto$: We distinguish two cases.

$\langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y', \sigma' \rangle$: We distinguish the following cases.

$(\langle x, \sigma \rangle\downarrow \vee \langle y, \sigma \rangle\downarrow) \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $p' \equiv y' \ ; \ z \parallel z'$. According to Rules 22, 23, and 16 we have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle y' \ ; \ z \parallel z', \sigma \rangle$. So, we take $p'' \equiv p'$, $q' \equiv y' \ ; \ z \parallel z'$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

$(\langle x, \sigma \rangle \not\downarrow \wedge \langle y, \sigma \rangle \not\downarrow) \vee \langle z, \sigma \rangle \mapsto$: Note that $p' \equiv y' \ ; \ z$. According to Rules 20, 21, and 15 we have $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle y' \ ; \ z, \sigma' \rangle$. So, we take $p'' \equiv p'$, $q' \equiv y' \ ; \ z$, and $q'' \equiv q'$. Note that $(p'', q'') \in R$.

$\langle y, \sigma \rangle \mapsto$: Note that we must have $(\langle x, \sigma \rangle \downarrow \vee \langle y, \sigma \rangle \downarrow) \wedge \langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$, otherwise $\langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle p', \sigma \rangle$ is not possible. So, we have $p' \equiv z'$. According to Rules 22 and 16 we also have $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle z' \parallel z', \sigma' \rangle$. So, we take $p'' \equiv p'$ and $q' \equiv z' \parallel z'$. Using Lemma 4.15, we can make the following computation: $q' \equiv z' \parallel z' \underline{\leftrightarrow} z'$. So, we define $q'' \equiv z'$ and note that $(p'', q'') \in R$.

Condition 5: We have to prove $\forall q', d, \sigma, \sigma' : \langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle q', \sigma' \rangle \Rightarrow \exists p', p'', q'' : \langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle p', \sigma' \rangle$, $p' \underline{\leftrightarrow} p''$, $(p'', q'') \in R$, and $q'' \underline{\leftrightarrow} q'$. So, assume $\langle q, \sigma \rangle \stackrel{d}{\longmapsto} \langle q', \sigma' \rangle$. Since $q \equiv x \ ; \ z \parallel y \ ; \ z$ the relevant Rules are 15 and 16. Therefore, we distinguish the following cases.

$\langle x \ ; \ z, \sigma \rangle \stackrel{d}{\longmapsto} \langle q_0', \sigma' \rangle \wedge \langle y \ ; \ z \rangle \mapsto$: Note that $q' \equiv q_0'$. The relevant Rules are 20, 21, 22, and 23. Therefore, we distinguish the following cases.

$\langle x, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle \wedge (\langle x, \sigma \rangle \not\downarrow \vee \langle z, \sigma \rangle \mapsto)$: Note that $q' \equiv x' \ ; \ z$. According to Rules 20, 21, and 15 we have $\langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle x' \ ; \ z, \sigma' \rangle$. So, we take $p' \equiv x' \ ; \ z$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle \mapsto \wedge \langle x, \sigma \rangle \downarrow \wedge \langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $q' \equiv z'$. According to Rules 13 and 22 we have $\langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$. So, we take $p' \equiv z'$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle \wedge \langle x, \sigma \rangle \downarrow \wedge \langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$: Since we know that $\langle y \ ; \ z, \sigma \rangle \mapsto$ we can conclude that $\langle y, \sigma \rangle \mapsto$. Therefore, we have $q' \equiv x' \ ; z \| z'$. According to Rules 13 and 23 we get $\langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle x' \ ; \ z \parallel z', \sigma' \rangle$. So, we take $p' \equiv x' \ ; z \| z'$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$\langle x \ ; \ z, \sigma \rangle \mapsto \wedge \langle y \ ; \ z \rangle \stackrel{d}{\longmapsto} \langle q_1', \sigma' \rangle$: Note that $q' \equiv q_1'$. The proof is similar to the proof of the previous case.

$\langle x \ ; \ z, \sigma \rangle \stackrel{d}{\longmapsto} \langle q_0', \sigma' \rangle \wedge \langle y \ ; \ z \rangle \stackrel{d}{\longmapsto} \langle q_1', \sigma' \rangle$: Note that $q' \equiv q_0' \parallel q_1'$. We distinguish the following cases.

$\langle x, \sigma \rangle \stackrel{d}{\longmapsto} \langle x', \sigma' \rangle$:

$\langle y, \sigma \rangle \stackrel{d}{\longmapsto} \langle y', \sigma' \rangle$:

$\langle x, \sigma \rangle \downarrow \wedge \langle y, \sigma \rangle \downarrow \wedge \langle z, \sigma \rangle \stackrel{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $q' \equiv (x' \ ; \ z \parallel z') \parallel (y' \ ; \ z \parallel z')$. According to Rules 13, 16, and 23 we have $\langle p, \sigma \rangle \stackrel{d}{\longmapsto} \langle (x' \parallel y') \ ; \ z \parallel z', \sigma' \rangle$. So, we take $p' \equiv (x' \parallel y') \ ; \ z \parallel z'$. Now we can make the same computation as in the corresponding case of the proof of Condition 4: $q' \equiv (x' \ ; z \| z') \| (y' \ ; z \| z') \underline{\leftrightarrow} (x' \ ; z \| y' \ ; z) \| z'$.

So, we define $q'' \equiv (x' \ ; \ z \parallel y' \ ; \ z) \parallel z'$ and take $p'' \equiv p'$. Given the definition of R_1 and R_2, it is clear that $(p'', q'') \in R$.

$\langle x, \sigma \rangle\!\downarrow \wedge \langle y, \sigma \rangle \not\downarrow \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $q' \equiv (x' \ ; \ z \parallel z') \parallel y' \ ; \ z$. According to Rules 13, 16, and 23 we have $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle (x' \parallel y') \ ; \ z \parallel z', \sigma' \rangle$. So, take $p' \equiv (x' \parallel y') \ ; \ z \parallel z'$ and $p'' \equiv p'$. Further, consider the following computation: $q' \equiv (x' \ ; \ z \parallel z') \parallel y' \ ; \ z \underline{\leftrightarrow} x' \ ; \ z \parallel (z' \parallel y' \ ; \ z) \underline{\leftrightarrow} x' \ ; \ z \parallel (y' \ ; \ z \parallel z') \underline{\leftrightarrow} (x' \ ; \ z \parallel y' \ ; \ z) \parallel z'$. So, we take $q'' \equiv (x' \ ; \ z \parallel y' \ ; \ z) \parallel z'$ and note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle \not\downarrow \wedge \langle y, \sigma \rangle\!\downarrow \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $q' \equiv x' \ ; \ z \parallel (y' \ ; \ z \parallel z')$. The proof continues in the same way as in the previous case.

$(\langle x, \sigma \rangle \not\downarrow \wedge \langle y, \sigma \rangle \not\downarrow) \vee \langle z, \sigma \rangle \mapsto\!\!\!\!\!\!\rightarrow$: Note that $q' \equiv x' \ ; \ z \parallel y' \ ; \ z$. According to Rules 16, 20, and 21 we have $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle (x' \parallel y') \ ; \ z, \sigma' \rangle$. So, we take $p' \equiv (x' \parallel y') \ ; \ z$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$\langle y, \sigma \rangle \mapsto\!\!\!\!\!\!\rightarrow$: We distinguish the following cases.

$\langle x, \sigma \rangle\!\downarrow \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $q' \equiv x' \ ; \ z \parallel z'$. According to Rules 15 and 23 we have $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle x' \ ; \ z \parallel z', \sigma' \rangle$. So, we take $p' \equiv x' \ ; \ z \parallel z'$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle \not\downarrow \vee \langle z, \sigma \rangle \mapsto\!\!\!\!\!\!\rightarrow$: Note that $q' \equiv x' \ ; \ z$. According to Rules 15, 20, and 21 we have $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle x' \ ; \ z, \sigma' \rangle$. So, we take $p' \equiv x' \ ; \ z$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle \mapsto\!\!\!\!\!\!\rightarrow$: We distinguish the following cases.

$\langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y', \sigma' \rangle$: We distinguish the following cases.

$\langle x, \sigma \rangle\!\downarrow \wedge \langle y, \sigma \rangle\!\downarrow \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $q' \equiv z' \parallel (y' \ ; z \parallel z')$. According to Rules 13, 16, 22, and 23 we have $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle y' \ ; \ z \parallel z', \sigma' \rangle$. So, take $p' \equiv y' \ ; \ z \parallel z'$ and $p'' \equiv p'$. Further, consider the following computation: $q' \equiv z' \parallel (y' \ ; \ z \parallel z') \underline{\leftrightarrow} (\parallel y' \ ; \ z \parallel z') \parallel z' \underline{\leftrightarrow} y' \ ; \ z \parallel (z' \parallel z') \underline{\leftrightarrow} y' \ ; \ z \parallel z'$. So, take $q'' \equiv y' \ ; \ z \parallel z'$ and note that $(p'', q'') \in R$.

$\langle x, \sigma \rangle\!\downarrow \wedge \langle y, \sigma \rangle \not\downarrow \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that we have $q' \equiv z' \parallel y' \ ; z$. Now the proof continues in the same way as in the previous case.

$\langle x, \sigma \rangle \not\downarrow \wedge \langle y, \sigma \rangle\!\downarrow \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z', \sigma' \rangle$: Note that $q' \equiv y' \ ; \ z \parallel z'$. Now the proof continues in the same way as in the previous case.

$(\langle\, x, \sigma \,\rangle\not\downarrow \wedge \langle\, y, \sigma \,\rangle\not\downarrow) \vee \langle\, z, \sigma \,\rangle\mapsto\!\!\!\!\!\rightarrow$: Note that $q' \equiv y' \;;\; z$. According to Rules 20, 21, and 15 we have $\langle\, p, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, y' \;;\; z, \sigma' \,\rangle$. So, take $p' \equiv y' \;;\; z$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$\langle\, y, \sigma \,\rangle\mapsto\!\!\!\!\!\rightarrow$: Note that we must have $(\langle\, x, \sigma \,\rangle\downarrow \vee \langle\, y, \sigma \,\rangle\downarrow) \wedge \langle\, z, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, z', \sigma' \,\rangle$, since otherwise $\langle\, q, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$ cannot hold. So, we have $q' \equiv z'$ (if either $\langle\, x, \sigma \,\rangle\downarrow$ or $\langle\, y, \sigma \,\rangle\downarrow$) or $q' \equiv z' \parallel z'$ (if both $\langle\, x, \sigma \,\rangle\downarrow$ and $\langle\, y, \sigma \,\rangle\downarrow$).

$q' \equiv z'$: According to Rules 13 and 22 we have $\langle\, p, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, z', \sigma' \,\rangle$. So, we take $p' \equiv z'$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $(p'', q'') \in R$.

$q' \equiv z' \parallel z'$: According to Rules 13 and 22 we have $\langle\, p, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, z', \sigma' \,\rangle$. So, we take $p' \equiv z'$ and $p'' \equiv p'$. Further, using Lemma 4.15 we can make the following computation: $q' \equiv z' \parallel z' \leftrightarrows z'$. So, we take $q'' \equiv z'$ and note that $(p'', q'') \in R$.

Inductive step Suppose $(p, q) \in R_2$. According to definition of R_2, there are processes x, y, and z such that $p \equiv x \parallel z$ and $q \equiv y \parallel z$ and $(x, y) \in R$. Furthermore, the induction hypothesis says the five bisimulation conditions hold for the pair (x, y).

Condition 1: We have to prove $\forall \sigma : \langle\, p, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\downarrow$. Using the induction hypothesis and Rule 13 we can make the following computation:

$$\begin{array}{ll} & \langle\, p, \sigma \,\rangle\downarrow \\ \Leftrightarrow & \langle\, x, \sigma \,\rangle\downarrow \vee \langle\, z, \sigma \,\rangle\downarrow \\ \Leftrightarrow & \langle\, y, \sigma \,\rangle\downarrow \vee \langle\, z, \sigma \,\rangle\downarrow \\ \Leftrightarrow & \langle\, q, \sigma \,\rangle\downarrow. \end{array}$$

Condition 2: We have to prove $\forall a, p_a, \sigma, \sigma' : \langle\, p, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, p_a, \sigma' \,\rangle \Rightarrow \exists q_a : \langle\, q, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, q_a, \sigma' \,\rangle$ and $(p_a, q_a) \in R$. Consider the following computation:

$$\begin{array}{ll} & \langle\, p, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, p_a, \sigma' \,\rangle \\ \Leftrightarrow & (\langle\, x, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, p_a, \sigma' \,\rangle) \vee (\langle\, z, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, p_a, \sigma' \,\rangle) \\ \Leftrightarrow & (\langle\, y, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, p_a, \sigma' \,\rangle) \vee (\langle\, z, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, p_a, \sigma' \,\rangle) \\ \Leftrightarrow & \langle\, q, \sigma \,\rangle \stackrel{a}{\longrightarrow} \langle\, p_a, \sigma' \,\rangle. \end{array}$$

So, take $q_a \equiv p_a$ and note that $(p_a, q_a) \in R$.

Condition 3: The proof is similar to the proof of Condition 2.

Condition 4: We have to prove $\forall d, p_d, \sigma, \sigma' : \langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma' \rangle \Rightarrow \exists q_d : \langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q_d, \sigma' \rangle$. So, assume $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma' \rangle$. We distinguish three cases.

- $\langle x, \sigma \rangle \overset{d}{\longmapsto} \langle x_d, \sigma' \rangle \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z_d, \sigma' \rangle$: Note that $p_d \equiv x_d \parallel z_d$. Using Condition 4 on x, we obtain $\langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y_d, \sigma' \rangle$ for some y_d such that $(x_d, y_d) \in R$. According to Rule 16 we therefore have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q_d, \sigma' \rangle$ where $q_d \equiv y_d \parallel z_d$. Note that $(p_d, q_d) \in R$.
- $\langle x, \sigma \rangle \overset{d}{\longmapsto} \langle x_d, \sigma' \rangle \wedge \langle z, \sigma \rangle \not\longmapsto$: Note that $p_d \equiv x_d$. Using Condition 4 on x, we obtain $\langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y_d, \sigma' \rangle$ for some y_d such that $(x_d, y_d) \in R$. According to Rule 15 we therefore have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle y_d, \sigma' \rangle$. So, take $q_d \equiv y_d$. Note that $(p_d, q_d) \in R$.
- $\langle x, \sigma \rangle \not\longmapsto \wedge \langle z, \sigma \rangle \overset{d}{\longmapsto} \langle z_d, \sigma' \rangle$: Note that $p_d \equiv z_d$. According to Rule 15 we have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle z_d, \sigma' \rangle$. So, take $q_d \equiv z_d$ and note that $(p_d, q_d) \in R$.

Condition 5: The proof is similar to the proof of Condition 4. □

4.9 · Repetition operator

In Definition 4.24, we define the repetition operator '*'. A process p^* executes p zero or more times. This operator is often referred to as the (unary) Kleene star.

Definition 4.24 · (Repetition operator) *The repetition operator has the following signature:*

$$\_^* \quad : \quad P \to P.$$

The deduction rules for the repetition operator are listed in Table 4.5.

$$\frac{}{\langle p^*, \sigma \rangle \downarrow}\,24 \qquad \frac{\langle p, \sigma \rangle \overset{a}{\longrightarrow} \langle p', \sigma' \rangle}{\langle p^*, \sigma \rangle \overset{a}{\longrightarrow} \langle p' \ ; \ p^*, \sigma' \rangle}\,25 \qquad \frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle}{\langle p^*, \sigma \rangle \overset{d}{\longmapsto} \langle p' \ ; \ p^*, \sigma' \rangle}\,26$$

Table 4.5 · Deduction rules for the repetition operator.

Rule 24 states that a process p^* can always terminate. Rule 25 states that if a process p can perform an action, then the repetition p^* can also perform the

action. Finally, Rule 26 states that if a process p can perform a delay, then the repetition p^* can also perform that delay.

Lemma 4.25 · $\delta^* \leftrightarrows \varepsilon$.

Proof · (Lemma 4.25) We have to prove that $\delta^* \leftrightarrows \varepsilon$. In this case, we define a relation $R \subseteq P \times P$ such that $(\delta^*, \varepsilon) \in R$ and R is a bisimulation.

We define R as

$$R = \{(\delta^*, \varepsilon)\}$$

and show that pair $(\delta^*, \varepsilon) \in R$ satisfies the five bisimulation conditions of Definition 4.5.

Condition 1: We have to prove $\forall \sigma : \langle\, \delta^*, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, \varepsilon, \sigma \,\rangle\downarrow$. According to Rules 1 and 24 both $\langle\, \delta^*, \sigma \,\rangle\downarrow$ and $\langle\, \varepsilon, \sigma \,\rangle\downarrow$ hold. Therefore, the condition holds.

Condition 2–5: The proof is trivial since the left-hand side of the implication does not hold. □

Lemma 4.26 · *Let p be a process, then*

$$p^* \leftrightarrows p \;;\; p^* \parallel \varepsilon.$$

Proof · (Lemma 4.26) We have to prove $p^* \leftrightarrows p \;;\; p^* \parallel \varepsilon$ for all processes p. In this case, we define a relation $R \subseteq P \times P$ such that $(p^*, p \;;\; p^* \parallel \varepsilon) \in R$ and R is a 'bisimulation up to $\leftrightarrows$ relation (see Definition 4.6). We define R as

$$R = \{(p, p)\} \cup \{(p^*, p \;;\; p^* \parallel \varepsilon)\}.$$

Now we will show that each pair $(p, q) \in R$ satisfies the five 'bisimulation up to $\leftrightarrows$' conditions of Definition 4.6.Since the proofs for the pairs of the form (p, p) are trivial, we will only consider the pairs of the form $(p^*, p \;;\; p^* \parallel \varepsilon)$. So, assume $p \equiv x^*$ and $q \equiv x \;;\; x^* \parallel \varepsilon$, for some process x.

Condition 1: We have to prove $\forall \sigma : \langle\, p, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\downarrow$. This is easily proved using Rules 1, 17, 24, and 13: $\langle\, p, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, x^*, \sigma \,\rangle \Leftrightarrow \mathit{true} \Leftrightarrow \langle\, \varepsilon, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\downarrow$.

Condition 2: We have to prove $\forall \sigma, a, p', \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \;\Rightarrow\; \exists p'', q', q'' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$, $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$. So, assume $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. Since Rule 25 is the only rule that applies, we can derive $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x', \sigma' \,\rangle$ and $p' \equiv x' \;;\; x^*$. According to Rules 18 and 14, we also have $\langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, x' \;;\; x^*, \sigma' \,\rangle$. So, take $p'' \equiv p'$, $q' \equiv p'$, and $q'' \equiv q'$. Finally, note that $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$.

Condition 3: We have to prove $\forall \sigma, a, q', \sigma' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \;\Rightarrow\; \exists p', p'', q'' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$. So, assume $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$. Both Rule 18 and Rule 19 apply. However, in both cases we get $\langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, x' \;;\; x^*, \sigma' \,\rangle$ and $q' \equiv x' \;;\; x^*$. That is, if we derive $\langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, x' \;;\; x^*, \sigma' \,\rangle$ according to Rule 18 we can also derive (for the same x') $\langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle \xrightarrow{a} \langle\, x' \;;\; x^*, \sigma' \,\rangle$ Rule 19, and vice versa. Furthermore, we have $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x', \sigma' \,\rangle$. Therefore, using Rule 25, we find $\langle\, x^*, \sigma \,\rangle \xrightarrow{a} \langle\, x' \;;\; x^*, \sigma \,\rangle$. So, take $p' \equiv q'$, $p'' \equiv p'$, and $q'' \equiv q'$. Finally, note that $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$.

Condition 4: We have to prove $\forall \sigma, d, p', \sigma' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \;\Rightarrow\; \exists p'', q', q'' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$, $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$. So, assume $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$. Since Rule 26 is the only rule that applies, we have $\langle\, x^*, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \;;\; x^*, \sigma' \,\rangle$ for some x' such that $p' \equiv x' \;;\; x^*$ and $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x', \sigma' \,\rangle$. Depending on the termination behaviour of x, we distinguish two cases.

- $\langle\, x, \sigma \,\rangle\!\downarrow$: Using Rules 23, 15 and 26, we find $\langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \;;\; x^* \parallel x' \;;\; x^*, \sigma' \,\rangle$. According to Lemma 4.15, we have $x' \;;\; x^* \parallel x' \;;\; x \leftrightarrows x' \;;\; x^*$. So, take $p'' \equiv p$, $q' \equiv x' \;;\; x^* \parallel x' \;;\; x$, and $q'' \equiv p'$. Finally, note that $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$.
- $\langle\, x, \sigma \,\rangle\!\not\downarrow$: Using Rules 20 and 15 we find $\langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \;;\; x^*, \sigma \,\rangle$. So, take $p'' \equiv p'$, $q' \equiv p'$, and $q'' \equiv p'$. Finally, note that $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$.

Condition 5: We have to prove $\forall \sigma, d, q', \sigma' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \;\Rightarrow\; \exists p', p'', q'' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$, $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$. Assume $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$. Depending on the delay behaviour of x we distinguish two cases.

- $\langle\, x, \sigma \,\rangle\!\downarrow$: Now, Rules 23 and 15 apply and, consequently, we have $\langle\, x \;;\; x^* \parallel \varepsilon, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \;;\; x^* \parallel x' \;;\; x^*, \sigma \,\rangle$ for some process x such that $q' \equiv x' \;;\; x^* \parallel x' \;;\; x^*$ and $\langle\, x, \sigma \,\rangle \xrightarrow{d} \langle\, x', \sigma' \,\rangle$. According to Lemma 4.15, we have $x' \;;\; x^* \parallel x' \;;\; x \leftrightarrows x' \;;\; x^*$. Furthermore, using Rule 26, we obtain $\langle\, x^*, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \;;\; x^*, \sigma' \,\rangle$. So, take $p' \equiv x' \;;\; x^*$, $p'' \equiv p'$, and $q'' \equiv p'$. Finally, note that $p' \leftrightarrows p''$, $(p'', q'') \in R$, and $q'' \leftrightarrows q'$.

$\langle x, \sigma \rangle \not\downarrow$: Now, Rules 20 and 15 apply and, consequently, we have $\langle x \ ; \ x^* \parallel \varepsilon, \sigma \rangle \overset{d}{\longmapsto} \langle x' \ ; \ x^*, \sigma \rangle$ for some process x such that $q' \equiv x' \ ; \ x^*$ and $\langle x, \sigma \rangle \xrightarrow{d} \langle x', \sigma' \rangle$. According to Rule 26 we obtain $\langle x^*, \sigma \rangle \overset{d}{\longmapsto} \langle x' \ ; \ x^*, \sigma' \rangle$. So, take $p' \equiv q'$, $p'' \equiv q'$, and $q'' \equiv q'$. Finally, note that $p' \underline{\leftrightarrow} p''$, $(p'', q'') \in R$, and $q'' \underline{\leftrightarrow} q'$. □

4.10 · Parallel composition operator

In Definition 4.27, we define the parallel composition operator '$\parallel$'. A process $p \parallel q$ executes p and q concurrently in an interleaved fashion. That is, the actions of p and q are executed in arbitrary order.

Definition 4.27 · (Parallel composition operator) *The parallel composition operator has the following signature:*

$$\_ \parallel \_ \ : \ P \times P \to P.$$

The deduction rules for the parallel composition operator are listed in Table 4.6.

$$\frac{\langle p, \sigma \rangle\downarrow, \ \langle q, \sigma \rangle\downarrow}{\langle p \parallel q, \sigma \rangle\downarrow}\ 27 \qquad \frac{\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{a} \langle p' \parallel q, \sigma' \rangle, \ \langle q \parallel p, \sigma \rangle \xrightarrow{a} \langle q \parallel p', \sigma' \rangle}\ 28$$

$$\frac{\langle p, \sigma \rangle \xrightarrow{sa(m,c)} \langle p', \sigma' \rangle, \ \langle q, \sigma' \rangle \xrightarrow{ra(m,x)} \langle q', \sigma'' \rangle}{\langle p \parallel q, \sigma \rangle \xrightarrow{ca(m,x,c)} \langle p' \parallel q', \sigma'' \rangle, \ \langle q \parallel p, \sigma \rangle \xrightarrow{ca(m,x,c)} \langle q' \parallel p', \sigma'' \rangle}\ 29$$

$$\frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle, \ \langle q, \sigma \rangle\downarrow, \ \langle q, \sigma \rangle\not\mapsto}{\langle p \parallel q, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle, \ \langle q \parallel p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle}\ 30$$

$$\frac{\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle, \ \langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle}{\langle p \parallel q, \sigma \rangle \overset{d}{\longmapsto} \langle p' \parallel q', \sigma' \rangle}\ 31$$

Table 4.6 · Deduction rules for the parallel composition operator.

If both process p and q can terminate, then the parallel composition $p \parallel q$ can also terminate, as defined by Rule 27. If process p can perform an action, then both the parallel compositions $p \parallel q$ and $q \parallel p$ can perform the action as well, as defined by Rule 28. This expresses interleaving. Furthermore, if processes p and q can perform matching send and receive actions, then the parallel compositions $p \parallel q$ and $q \parallel p$ can perform a communication action, as defined by Rule 29. Notice that there is an order dependency. The stack is first updated by the send action which changes the value of a channel. Next, the stack is updated by the receive action. The delay behaviour of the parallel composition is defined by the Rules 30 and 31. If only one argument of a parallel composition $p \parallel q$ can delay, say p, then the parallel composition can delay only if the other argument, q, can terminate. This is defined by Rule 30. Note that in this case, argument q is lost. If both arguments can delay, then they delay together, as defined by Rule 31.

Lemma 4.28 · *Let p be a process, then*

$$\varepsilon \parallel p \leftrightarrows p.$$

Proof · (Lemma 4.28) We have to prove that $\varepsilon \parallel p \leftrightarrows p$ for all processes p. In this case, we define a bisimulation relation $R \subseteq P \times P$ such that $(\varepsilon \parallel p, p) \in R$ and R is a bisimulation. We define R as

$$R = \{(p, p)\} \cup \{(\varepsilon \parallel p, p)\}$$

and show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs are trivial for the pairs (p, p), we only give the proofs for the pairs $(\varepsilon \parallel p, p)$.

Condition 1: We have to prove $\forall \sigma : \langle\, \varepsilon \parallel p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, \sigma \,\rangle\!\downarrow$. Let us first prove the right implication of Condition 1. Suppose $\langle\, \varepsilon \parallel p, \sigma \,\rangle\!\downarrow$. Then Rule 27 should apply and we obtain $\langle\, p, \sigma \,\rangle\!\downarrow$. For the left implication we find the following. Suppose $\langle\, p, \sigma \,\rangle\!\downarrow$. Rule 1 combined with Rule 27 then gives $\langle\, \varepsilon \parallel p, \sigma \,\rangle\!\downarrow$.

Condition 2: We have to prove $\forall \sigma, \sigma', q, a : \langle\, \varepsilon \parallel p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (q, r) \in R$. Suppose $\langle\, \varepsilon \parallel p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$, then also $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$ since there are no action transitions defined for ε. In that case, we also have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and $(q, r) \in R$.

Condition 3: We have to prove $\forall \sigma, \sigma', q, a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, \varepsilon \parallel p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (r, q) \in R$. Suppose $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$, then, by Rule 28 we also have $\langle\, \varepsilon \parallel p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and $(r, q) \in R$.

Condition 4: We have to prove $\forall \sigma, \sigma', q, d : \langle\, \varepsilon \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle \wedge (q, r) \in R$. Suppose $\langle\, \varepsilon \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle$, then also $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle$ since there are no delay transitions defined for ε. In that case, we also have $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and $(q, r) \in R$.

Condition 5: We have to prove $\forall \sigma, \sigma', q, d : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, \varepsilon \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle \wedge (r, q) \in R$. Suppose $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q, \sigma' \,\rangle$, then, by Rule 30 we also have $\langle\, \varepsilon \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, where $r \equiv q$ and $(r, q) \in R$. □

Lemma 4.29 · *Let p and q be processes, then*

$$p \parallel q \leftrightarrows q \parallel p.$$

Proof · (Lemma 4.29) We have to prove $p \parallel q \leftrightarrows q \parallel p$ for all processes p and q. The relevant deduction rules are Rules 27, 28, 29, 30, and 31. Observing these rules, we see that whenever we can derive $\langle\, p \parallel q, \sigma \,\rangle\!\downarrow$, $\langle\, p \parallel q, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, or $\langle\, p \parallel q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, we can also derive $\langle\, q \parallel p, \sigma \,\rangle\!\downarrow$, $\langle\, q \parallel p, \sigma' \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle$, and $\langle\, q \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, r, \sigma' \,\rangle$, respectively. □

Lemma 4.30 · *Let p, q, and r be processes, then*

$$(p \parallel q) \parallel r \leftrightarrows p \parallel (q \parallel r).$$

Proof · (Lemma 4.30) We have to prove $(p \parallel q) \parallel r \leftrightarrows p \parallel (q \parallel r)$ for all processes p, q, and r. In this case, we define a relation $R \subseteq P \times P$ such that $((p \| q) \| r, p \| (q \| r)) \in R$ and R is a bisimulation. We define R as

$$R = \{(p, p)\} \cup \{((p \parallel q) \parallel r, p \parallel (q \parallel r))\}$$

and show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proof for pairs of the form (p, p) is trivial, we will only consider pairs of the form $((p \parallel q) \parallel r, p \parallel (q \parallel r))$. Suppose $p \equiv (x \parallel y) \parallel z$ and $q \equiv x \parallel (y \parallel z)$ for some processes x, y, and z.

Condition 1: We have to prove $\forall \sigma : \langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\!\downarrow$. The following computation shows that this holds (we use Rule 27 and associativity of '$\wedge$'):

$$\begin{aligned} & \langle\, p, \sigma \,\rangle\!\downarrow \\ \Leftrightarrow\ & (\langle\, x, \sigma \,\rangle\!\downarrow \wedge \langle\, y, \sigma \,\rangle\!\downarrow) \wedge \langle\, z, \sigma \,\rangle\!\downarrow \\ \Leftrightarrow\ & \langle\, x, \sigma \,\rangle\!\downarrow \wedge (\langle\, y, \sigma \,\rangle\!\downarrow \wedge \langle\, z, \sigma \,\rangle\!\downarrow) \\ \Leftrightarrow\ & \langle\, q, \sigma \,\rangle\!\downarrow. \end{aligned}$$

Condition 2: We have to prove $\forall a, p', \sigma, \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$ and $(p', q) \in R$. So, assume $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. This means that Rule 28 or 29 applies. So, we distinguish two cases.

- $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$ *According to Rule 28*: Recall that $p \equiv (x \parallel y) \parallel z$. Therefore we have $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x', \sigma' \,\rangle$, $\langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, y', \sigma' \,\rangle$, or $\langle\, z, \sigma \,\rangle \xrightarrow{a} \langle\, z', \sigma' \,\rangle$. Furthermore, we have $p' \equiv (x' \parallel y) \parallel z$, $p' \equiv (x \parallel y') \parallel z$, or $p' \equiv (x \parallel y) \parallel z'$, respectively. Also, we can use Rule 28 to obtain $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, x' \parallel (y \,[\!]\, z), \sigma' \,\rangle$, $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, x \parallel (y' \,[\!]\, z), \sigma' \,\rangle$, or $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, x \parallel (y \,[\!]\, z'), \sigma' \,\rangle$, respectively. Finally, note that in all cases we have $(p', q') \in R$.
- $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$ *According to Rule 29*: Note that $a \equiv ca(m, x, c)$ for some channel m, programming variable x, and value c. Recall that $p \equiv (x \parallel y) \parallel z$. Since any process can send to any other process, there are six possibilities. The proofs for these six cases are similar, and therefore, we will only consider the case that x sends to z. So, according to Rule 29 there is a σ'' such that $\langle\, x, \sigma \,\rangle \xrightarrow{sa(m,c)} \langle\, x', \sigma'' \,\rangle$ and $\langle\, z, \sigma'' \,\rangle \xrightarrow{ra(m,x)} \langle\, z', \sigma' \,\rangle$ and, therefore, $p' \equiv (x' \parallel y) \parallel z'$. By the same rule, we can also derive that $\langle\, q, \sigma \,\rangle \xrightarrow{ca(m,x,c)} \langle\, x' \parallel (y \parallel z'), \sigma' \,\rangle$. So, we take $q' \equiv x' \parallel (y \parallel z')$ and note that $(p', q') \in R$.

Condition 3: The proof is similar to proof of Condition 2.

Condition 4: We have to prove $\forall p', d, \sigma, \sigma' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$ and $(p', q') \in R$. So, assume $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$. We distinguish the following cases.

- $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x', \sigma' \,\rangle$, $\langle\, y, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, y', \sigma' \,\rangle$, $\langle\, z, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, z', \sigma' \,\rangle$: Note that $p' \equiv (x' \parallel y') \parallel z'$. Using Rule 31 we obtain $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \parallel (y' \parallel z'), \sigma' \,\rangle$. So, take $q' \equiv x' \parallel (y' \parallel z')$ and note that $(p', q') \in R$.
- $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x', \sigma' \,\rangle$, $\langle\, y, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, y', \sigma' \,\rangle$, $\langle\, z, \sigma \,\rangle \mapsto$, $\langle\, z, \sigma \,\rangle \downarrow$: Note that $p' \equiv x' \parallel y'$. Using Rules 30 and 31 we obtain $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \parallel y', \sigma' \,\rangle$. So, take $q' \equiv x' \parallel y'$ and note that $(p', q') \in R$.
- $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \sigma' \,\rangle$, $\langle\, y, \sigma \,\rangle \mapsto$, $\langle\, y, \sigma \,\rangle \downarrow$, $\langle\, z, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, z', \sigma' \,\rangle$: Note that $p' \equiv x' \| z'$. Using Rules 30 and 31 we obtain $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \parallel z', \sigma' \,\rangle$. So, take $q' \equiv x' \parallel z'$ and note that $(p', q') \in R$.
- $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x', \sigma' \,\rangle$, $\langle\, y, \sigma \,\rangle \mapsto$, $\langle\, y, \sigma \,\rangle \downarrow$, $\langle\, z, \sigma \,\rangle \mapsto$, $\langle\, z, \sigma \,\rangle \downarrow$: Note that $p' \equiv x'$. Using Rule 30 we obtain $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x', \sigma' \,\rangle$. So, take $q' \equiv x'$ and note that $(p', q') \in R$.

$\langle x, \sigma \rangle \mapsto$, $\langle x, \sigma \rangle \downarrow$, $\langle y, \sigma \rangle \xrightarrow{d} \langle y', \sigma' \rangle$, $\langle z, \sigma \rangle \xrightarrow{d} \langle z', \sigma' \rangle$: Note that $p' \equiv y' \parallel z'$. Using Rules 30 and 31 we obtain $\langle q, \sigma \rangle \xrightarrow{d} \langle y' \parallel z', \sigma' \rangle$. So, take $q' \equiv y' \parallel z'$ and note that $(p', q') \in R$.

$\langle x, \sigma \rangle \mapsto$, $\langle x, \sigma \rangle \downarrow$, $\langle y, \sigma \rangle \xrightarrow{d} \langle y', \sigma' \rangle$, $\langle z, \sigma \rangle \mapsto$, $\langle z, \sigma \rangle \downarrow$: Note that $p' \equiv y'$. Using Rule 30 we obtain $\langle q, \sigma \rangle \xrightarrow{d} \langle y', \sigma' \rangle$. So, take $q' \equiv y'$ and note that $(p', q') \in R$.

$\langle x, \sigma \rangle \mapsto$, $\langle x, \sigma \rangle \downarrow$, $\langle y, \sigma \rangle \mapsto$, $\langle y, \sigma \rangle \downarrow$, $\langle z, \sigma \rangle \xrightarrow{d} \langle z', \sigma' \rangle$: Note that $p' \equiv z'$. Using Rule 30 we obtain $\langle q, \sigma \rangle \xrightarrow{d} \langle z', \sigma' \rangle$. So, take $q' \equiv z'$ and note that $(p', q') \in R$.

Condition 5: The proof is similar to the proof of Condition 4. □

4.11 · State operator

In Definition 4.31, we define the state operator '$[\![\mid]\!]$'. A process $[\![s \mid p]\!]$, where s is a state (see Section 4.1) and p a process, behaves like p in the (local) state s. This state s can be used to define (local) programming variables or channels.

Definition 4.31 · (State operator) *The state operator has the following signature:*

$$[\![\_ \mid \_]\!] \quad : \quad State \times P \to P.$$

The deduction rules for the state operator are listed in Table 4.7.

$$\frac{\langle p, s :: \sigma \rangle \downarrow}{\langle [\![s \mid p]\!], \sigma \rangle \downarrow}\,32 \qquad \frac{\langle p, s :: \sigma \rangle \xrightarrow{a} \langle p', s' :: \sigma' \rangle}{\langle [\![s \mid p]\!], \sigma \rangle \xrightarrow{a} \langle [\![s' \mid p']\!], \sigma' \rangle}\,33$$

$$\frac{\langle p, s :: \sigma \rangle \xrightarrow{d} \langle p', s' :: \sigma' \rangle}{\langle [\![s \mid p]\!], \sigma \rangle \xrightarrow{d} \langle [\![s' \mid p']\!], \sigma' \rangle}\,34$$

Table 4.7 · Deduction rules for the state operator.

The semantics of a state operator process $[\![s \mid p]\!]$ under a stack σ is similar to the semantics of its process argument p under the stack $s :: \sigma$. So, in order to determine if $[\![s \mid p]\!]$ can terminate under a stack σ, we have to determine if p

terminates under stack $s :: \sigma$. This is defined by Rule 32. The same 'push s on σ' approach determines the action and delay behaviour of the state operator, as defined by Rules 33 and 34, respectively.

The state operator as defined here, is similar to the state operator of the process algebra ACP [11, 12]. The ACP state operator is written as $\lambda_s(p)$, where s is an element of some domain S of states (not necessarily the same as the set *State* of χ_σ states) and p an ACP process. The process algebra ACP is parameterised by a set of (atomic) actions called A. The operational semantics of $\lambda_s(p)$ is defined by

$$\frac{p \xrightarrow{a} p'}{\lambda_s(p) \xrightarrow{action(a,s)} \lambda_{effect(a,s)}(p')} \ ,$$

where $a \in A$, the function $action : A \times S \to A$ returns an action, and the function $effect : A \times S \to S$ returns a state. The similarity between both state operators is illustrated by taking $S = State$, $A = Stack \times (Action \cup R_{>0}) \times Stack$, and by giving the partial definitions

$$\begin{aligned} action((s :: \sigma, a, s' :: \sigma'), s) &= (\sigma, a, \sigma'), \\ action((s :: \sigma, d, s' :: \sigma'), s) &= (\sigma, d, \sigma'), \\ effect((s :: \sigma, a, s' :: \sigma'), s) &= s', \\ effect((s :: \sigma, d, s' :: \sigma'), s) &= s', \end{aligned}$$

where $s, s' \in State$; $\sigma, \sigma' \in Stack$; $a \in Action$; and $d \in R_{>0}$. In order to see the χ_σ state operator as an instantiation of the ACP state operator, some technicalities have to be addressed. For example, the ACP state operator is usually not considered in a setting with a termination predicate depending on the state, and *action* and *effect* should be defined as total functions.

Lemma 4.32 · *Let p be a process, then*

$$[\![\lambda_s \mid p]\!] \leftrightarrows p.$$

Proof · (Lemma 4.32) We have to prove that $[\![\lambda_s \mid p]\!] \leftrightarrows p$ for all processes p. In this case, we define a relation $R \subseteq P \times P$ such that $([\![\lambda_s \mid p]\!], p) \in R$ and R is a bisimulation. We define R as

$$R = \{(p, p)\} \cup \{([\![\lambda_s \mid p]\!], p)\}$$

and show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs are trivial for the pairs (p, p), we only give the proofs for the pairs $(\llbracket\, \lambda_s \mid p \,\rrbracket, p)$.

Condition 1: We have to prove $\forall \sigma : \langle\, \llbracket\, \lambda_s \mid p \,\rrbracket, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, \sigma \,\rangle\!\downarrow$. Using Rule 32 this means we have to prove $\forall \sigma : \langle\, p, \lambda_s :: \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, \sigma \,\rangle\!\downarrow$. Using Definition A.11 we derive that $(\lambda_s :: \sigma)(i) = \sigma(i)$ for any identifier i. According to Lemma 4.52, we can derive $\langle\, p, \lambda_s :: \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, \sigma \,\rangle\!\downarrow$.

Condition 2: We have to prove $\forall \sigma, \sigma', q, a : \langle\, \llbracket\, \lambda_s \mid p \,\rrbracket, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (q, r) \in R$. So, assume $\langle\, \llbracket\, \lambda_s \mid p \,\rrbracket, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$. Using Rule 33 we obtain $\exists s', \sigma' : \langle\, p, \lambda_s :: \sigma \,\rangle \xrightarrow{a} \langle\, q, s' :: \sigma' \,\rangle$. Using Definition A.11 we derive that $(\lambda_s :: \sigma)(i) = \sigma(i)$ for any identifier i, in other words, $\lambda_s :: \sigma \stackrel{\circ}{=} \sigma$. According to Lemma 4.52, we can derive $\exists \sigma_q : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma_q \,\rangle$. Remains to prove that $\sigma' = \sigma_q$. According to Lemma 4.50, we have $(\lambda_s :: \sigma) = (s' :: \sigma')$ and $\sigma_q = \sigma$, $(\lambda_s :: \sigma)[c/i] = (s' :: \sigma')$ and $\sigma_q = \sigma[c/i]$, or $(\lambda_s :: \sigma)[c/i][c'/i'] = (s' :: \sigma')$ and $\sigma_q = \sigma[c/i][c'/i']$. Definitions A.10 and A.12 give us $\sigma = \sigma'$, $\sigma[c/i] = \sigma'$, or $\sigma[c/i][c'/i'] = \sigma'$. Therefore, in all three cases we have $\sigma_q = \sigma'$. So, we take $r \equiv q$ and note $(q, r) \in R$.

Condition 3: We have to prove $\forall \sigma, \sigma', q, a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle \Rightarrow \exists r : \langle\, \llbracket\, \lambda_s \mid p \,\rrbracket, \sigma \,\rangle \xrightarrow{a} \langle\, r, \sigma' \,\rangle \wedge (q, r) \in R$. So, suppose $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma' \,\rangle$. Using Definition A.11 we derive that $(\lambda_s :: \sigma)(i) = \sigma(i)$ for any identifier i, so $\lambda_s :: \sigma \stackrel{\circ}{=} \sigma$. According to Lemma 4.52 we know that $\exists \sigma_q : \langle\, p, \lambda_s :: \sigma \,\rangle \xrightarrow{a} \langle\, q, \sigma_q \,\rangle$. Furthermore, Lemma 4.50 can be used to derive $\sigma' = \sigma \wedge \sigma_q = \lambda_s :: \sigma$ or $(\exists c, i : \sigma' = \sigma[c/i] \wedge \sigma_q = (\lambda_s :: \sigma)[c/i])$, or $(\exists c, c', i, i' : \sigma' = \sigma[c/i][c'/i'] \wedge \sigma_q = (\lambda_s :: \sigma)[c/i]\sigma[c'/i'])$. Using Definition A.10, we find $\sigma' = \sigma \wedge \sigma_q = \lambda_s :: \sigma$ or $(\exists c, i : \sigma' = \sigma[c/i] \wedge \sigma_q = \lambda_s :: \sigma[c/i])$, or $(\exists c, c', i, i' : \sigma' = \sigma[c/i][c'/i'] \wedge \sigma_q = \lambda_s :: \sigma[c/i]\sigma[c'/i'])$. So, in all three cases we find $\sigma_q = \lambda_s :: \sigma'$ and therefore we have $\langle\, p, \lambda_s :: \sigma \,\rangle \xrightarrow{a} \langle\, q, \lambda_s :: \sigma' \,\rangle$. According to Rule 33 we obtain $\langle\, \llbracket\, \lambda_s \mid p \,\rrbracket, \sigma \,\rangle \xrightarrow{a} \langle\, \llbracket\, \lambda_s \mid q \,\rrbracket, \sigma' \,\rangle$. So, take $r \equiv \llbracket\, \lambda_s \mid q \,\rrbracket$ and note that $(r, q) \in R$.

Condition 4: The proof is similar to the proof of Condition 2.

Condition 5: The proof is similar to the proof of Condition 3. □

Lemma 4.33 · *Let s_0 and s_1 be states and let p be a process, then*

$$\llbracket\, s_0 \mid \llbracket\, s_1 \mid p \,\rrbracket \,\rrbracket \leftrightarrows \llbracket\, set(s_0, s_1) \mid p \,\rrbracket.$$

Proof · (Lemma 4.33) We have to prove that $[\![\, s \mid [\![\, s' \mid p \,]\!] \,]\!] \leftrightarrows [\![\, set(s,s') \mid p \,]\!]$ for all processes p and states s and s'. In this case, we define a relation $R \subseteq P \times P$ such that $([\![\, s \mid [\![\, s' \mid p \,]\!] \,]\!], [\![\, set(s,s') \mid p \,]\!]) \in R$ and R is a bisimulation. We define R as

$$R = \{(p,p)\} \cup \{([\![\, s \mid [\![\, s' \mid p \,]\!] \,]\!], [\![\, set(s,s') \mid p \,]\!])\}$$

and show that all pairs $(p,q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs are trivial for the pairs (p,p), we only give the proofs for the pairs $([\![\, s \mid [\![\, s' \mid p \,]\!] \,]\!], [\![\, set(s,s') \mid p \,]\!])$.

Condition 1: We have to prove $\forall \sigma : \langle\, [\![\, s \mid [\![\, s' \mid p \,]\!] \,]\!], \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, [\![\, set(s,s') \mid p \,]\!], \sigma \,\rangle\!\downarrow$. Using Rule 32 twice we obtain $\forall \sigma : \langle\, p, s' :: s :: \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, set(s,s') :: \sigma \,\rangle\!\downarrow$. If we can prove that $s' :: s :: \sigma \mathrel{\hat{=}} set(s,s') :: \sigma$, then we can use Lemma 4.52 to get $\forall \sigma : \langle\, p, s' :: s :: \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, set(s,s') :: \sigma \,\rangle\!\downarrow$.

So, we have to prove $s' :: s :: \sigma \mathrel{\hat{=}} set(s,s') :: \sigma$. According to Definition A.13 this means we have to prove $\forall i : (s' :: s :: \sigma)(i) = (set(s,s') :: \sigma)(i)$. We can distinguish three cases: $i \in dom(s')$, $i \notin dom(s') \wedge i \in dom(s)$, and $i \notin dom(s') \wedge i \notin dom(s)$.

$i \in dom(s')$: Using Lemma A.21 we find that also $i \in dom(set(s,s'))$. Consider the following computation:

$$\begin{array}{ll} & (s' :: s :: \sigma)(i) = (set(s,s') :: \sigma)(i) \\ \Leftrightarrow & \{\text{Definition A.11}\} \\ & s'(i) = set(s,s')(i) \\ \Leftrightarrow & \{\text{Lemma A.23}\} \\ & s'(i) = s'(i). \end{array}$$

So, for $i \in dom(s')$ we have $(s' :: s :: \sigma)(i) = (set(s,s') :: \sigma)(i)$.

$i \notin dom(s') \wedge i \in dom(s)$: Using Lemma A.21 we obtain $i \in dom(set(s,s'))$. Consider the following computation:

$$\begin{array}{ll} & (s' :: s :: \sigma)(i) = (set(s,s') :: \sigma)(i) \\ \Leftrightarrow & \{\text{Definition A.11}\} \\ & (s :: \sigma)(i) = set(s,s')(i) \\ \Leftrightarrow & \{\text{Definition A.11 and Lemma A.23}\} \\ & s(i) = s(i). \end{array}$$

So, for $i \notin dom(s') \wedge i \in dom(s)$ we have $(s' :: s :: \sigma)(i) = (set(s,s') :: \sigma)(i)$.

$i \notin dom(s') \wedge i \notin dom(s)$: Using Lemma A.21 we obtain $i \notin dom(set(s, s'))$. Consider the following computation:

$$\begin{array}{ll} & (s' :: s :: \sigma)(i) = (set(s, s') :: \sigma)(i) \\ \Leftrightarrow & \{\text{Definition A.11}\} \\ & (s :: \sigma)(i) = \sigma(i) \\ \Leftrightarrow & \{\text{Definition A.11}\} \\ & \sigma(i) = \sigma(i). \end{array}$$

So, we have derived $(s' :: s :: \sigma)(i) = (set(s, s') :: \sigma)(i)$.

As mentioned above, we can now see that Condition 1 holds.

Condition 2: We have to prove $\forall \sigma, \sigma', q, a : \langle [\![s \mid [\![s' \mid p]\!]]\!], \sigma \rangle \xrightarrow{a} \langle q, \sigma' \rangle \Rightarrow \exists r : \langle [\![set(s, s') \mid p]\!], \sigma \rangle \xrightarrow{a} \langle r, \sigma' \rangle \wedge (q, r) \in R$. So, assume $\langle [\![s \mid [\![s' \mid p]\!]]\!], \sigma \rangle \xrightarrow{a} \langle q, \sigma' \rangle$ holds. Using Rule 33 twice we obtain $\langle p, s' :: s :: \sigma \rangle \xrightarrow{a} \langle p_a, \sigma'_q \rangle$. Furthermore, the same rule also gives us $\exists s_a, s'_a : q \equiv [\![s_a \mid [\![s'_a \mid p_a]\!]]\!] \wedge \sigma'_q \equiv s'_a :: s_a :: \sigma'$. Using the same argument as in the proof of Condition 1, we can prove $s' :: s :: \sigma \stackrel{\circ}{=} set(s, s') :: \sigma$. Therefore, we can use Lemma 4.52 to derive $\langle p, set(s, s') :: \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_0 \rangle$ for some σ_0. Assume we have $\sigma_0 = set(s_a, s'_a) :: \sigma'$, that is, we have $\langle p, set(s, s') :: \sigma \rangle \xrightarrow{a} \langle p_a, set(s_a, s'_a) :: \sigma' \rangle$. According to Rule 33 we obtain $\langle [\![set(s, s') \mid p]\!], \sigma \rangle \xrightarrow{a} \langle [\![set(s_a, s'_a) \mid p_a]\!], \sigma' \rangle$. So, take $r \equiv [\![set(s_a, s'_a) \mid p_a]\!]$ and note that $(q, r) \in R$.

Remains to prove $\sigma_0 = set(s_a, s'_a) :: \sigma'$. Recall that we derived the transitions $\langle p, s' :: s :: \sigma \rangle \xrightarrow{a} \langle p_a, \sigma'_q \rangle$ where $\sigma'_q \equiv s'_a :: s_a :: \sigma'$ and $\langle p, set(s, s') :: \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_0 \rangle$. Therefore, based on Lemma 4.50 we distinguish the following cases.

$\sigma'_q = s' :: s :: \sigma \wedge \sigma_0 = set(s, s') :: \sigma$: Now we obtain $s' :: s :: \sigma = \sigma'_q \equiv s'_a :: s_a :: \sigma'$. Therefore, using Definition A.12, we have $s' = s'_a$, $s = s_a$, and $\sigma = \sigma'$. Consequently, we have $\sigma_0 = set(s, s') :: \sigma = set(s_a, s'_a) :: \sigma'$, which completes the proof of this case.

$\exists c, i : \sigma'_q = (s' :: s :: \sigma)[c/i] \wedge \sigma_0 = (set(s, s') :: \sigma)[c/i]$: We can distinguish the following cases: $i \in dom(s')$, $i \notin dom(s') \wedge i \in dom(s)$, and $i \notin (dom(s') \cup dom(s)) \wedge i \in dom(\sigma)$. In the first case we find, using Definition A.10, $\sigma'_q = s'[c/i] :: s :: \sigma$. Therefore, $s'_a = s'[c/i]$, $s = s_a$, and $\sigma = \sigma'$. Furthermore, we can use Definition A.10 and Lemma A.22 to rewrite $(set(s, s') :: \sigma)[c/i]$ to $set(s, s'[c/i]) :: \sigma$. So, $\sigma_0 = (set(s, s') :: \sigma)[c/i] = set(s_a, s'_a) :: \sigma'$.
In the second case, $i \notin dom(s') \wedge i \in dom(s)$, we find $\sigma'_q = s' :: (s[c/i]) :: \sigma$. Therefore, $s'_a = s'$, $s_a = s[c/i]$, and $\sigma = \sigma'$. Furthermore, we can use Definition A.10 and Lemma A.22 to rewrite $(set(s, s') :: \sigma)[c/i]$ to $set(s[c/i], s') :: \sigma$. So, $\sigma_0 = (set(s, s') :: \sigma)[c/i] = set(s_a, s'_a) :: \sigma'$.

In the third case, $i \notin (dom(s') \cup dom(s)) \wedge i \in dom(\sigma)$, we find $\sigma'_q = s' :: s :: (\sigma[c/i])$. Therefore, $s'_a = s'$, $s_a = s$, and $\sigma' = \sigma[c/i]$. Furthermore, we can use Definition A.10 and Lemma A.21 to rewrite $(set(s, s') :: \sigma)[c/i]$ to $set(s, s') :: (\sigma[c/i])$. So, $\sigma_0 = (set(s, s') :: \sigma)[c/i] = set(s_a, s'_a) :: \sigma'$.

$\exists c, c', i, i' : \sigma'_q = (s' :: s :: \sigma)[c/i][c'/i'] \wedge \sigma_0 = (set(s, s') :: \sigma)[c/i][c'/i']$: Luckily, the proof is similar to the previous case.

This concludes the proof for Condition 2.

Condition 3: We have to prove $\forall \sigma, \sigma', q, a : \langle [\![set(s, s') \mid p]\!], \sigma \rangle \xrightarrow{a} \langle q, \sigma' \rangle \Rightarrow \exists r : \langle [\![s \mid [\![s' \mid p]\!]]\!], \sigma \rangle \xrightarrow{a} \langle r, \sigma' \rangle \wedge (r, q) \in R$. So, assume $\langle [\![set(s, s') \mid p]\!], \sigma \rangle \xrightarrow{a} \langle q, \sigma' \rangle$ holds. Since Rule 33 is the only rule that applies, we can derive $\exists \sigma_0 : \langle p, set(s, s') :: \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_0 \rangle$. Using the same argument as in the proof of Condition 1, we can prove $s' :: s :: \sigma \stackrel{\circ}{=} set(s, s') :: \sigma$. Therefore, we can use Lemma 4.52 to derive $\exists \sigma'_q : \langle p, s' :: s :: \sigma \rangle \xrightarrow{a} \langle p_a, \sigma'_q \rangle$. Furthermore, by making the same case distinction, based on Lemma 4.50, as in the proof of Condition 2, the proof is completed.

Condition 4: The proof is similar to the proof of Condition 2, except that the case distinction based on Lemma 4.50 reduces to one case only.

Condition 5: The proof is similar to the proof of Condition 3, except that the case distinction based on Lemma 4.50 reduces to one case only. □

During the lifetime of a state operator process, the domain of its state cannot change. This is formalized by Lemma 4.34. Another property of the state operator is that it respects intuitive scoping rules for identifiers. That is, if the semantics of a process $[\![s \mid p]\!]$ is computed in a stack σ, then if an identifier $i \in s$ also occurs in σ, its value in σ is irrelevant. Lemma 4.35 formalizes the scoping behaviour of the state operator.

Lemma 4.34 · *Let s and s' be states, p and p' be processes, σ and σ' be stacks, a be an action, and d be a positive real number. Then*

$$\langle [\![s \mid p]\!], \sigma \rangle \xrightarrow{a} \langle [\![s' \mid p']\!], \sigma' \rangle \Rightarrow dom(s) = dom(s'),$$
$$\langle [\![s \mid p]\!], \sigma \rangle \xmapsto{d} \langle [\![s' \mid p']\!], \sigma' \rangle \Rightarrow dom(s) = dom(s').$$

Proof · (Lemma 4.34) The proof of this lemma is quite simple once we have proven Lemma 4.51 of Section 4.17. First, we have to prove $\langle [\![s \mid p]\!], \sigma \rangle \xrightarrow{a} \langle [\![s' \mid p']\!], \sigma' \rangle \Rightarrow dom(s) = dom(s')$. So, suppose $\langle [\![s \mid p]\!], \sigma \rangle \xrightarrow{a} \langle [\![s' \mid p']\!], \sigma' \rangle$. Since

Rule 33 is the only rule that applies, we know that $\langle\, p, s :: \sigma \,\rangle \xrightarrow{a} \langle\, p', s' :: \sigma' \,\rangle$. Using this transition and Lemma 4.49, we know that $s' :: \sigma' = s :: \sigma$, $s' :: \sigma' = (s :: \sigma)[c/i]$, or $s' :: \sigma' = (s :: \sigma)[c/i][c'/i']$. Using Definitions A.10 and A.12 and Lemma A.16, we have $dom(s') = dom(s)$.

The proof for the case $\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle \overset{d}{\longmapsto} \langle\, [\![\, s' \mid p' \,]\!], \sigma' \,\rangle \Rightarrow dom(s) = dom(s')$ is similar. □

Lemma 4.35 · *Let p and p' be a processes, s and s' be states, σ and σ' be stacks, i be an identifier, c be a value, a be an action, and d be a positive real value. Then for $i \in dom(s)$ we have*

$$\begin{aligned}
\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle\!\downarrow &\Leftrightarrow \langle\, [\![\, s \mid p \,]\!], \sigma[c/i] \,\rangle\!\downarrow,\\
\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle \xrightarrow{a} \langle\, [\![\, s' \mid p' \,]\!], \sigma' \,\rangle &\Leftrightarrow \langle\, [\![\, s \mid p \,]\!], \sigma[c/i] \,\rangle \xrightarrow{a} \langle\, [\![\, s' \mid p' \,]\!], \sigma'[c/i] \,\rangle,\\
\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle \overset{d}{\longmapsto} \langle\, [\![\, s' \mid p' \,]\!], \sigma' \,\rangle &\Leftrightarrow \langle\, [\![\, s \mid p \,]\!], \sigma[c/i] \,\rangle \overset{d}{\longmapsto} \langle\, [\![\, s' \mid p' \,]\!], \sigma'[c/i] \,\rangle.
\end{aligned}$$

Proof · (Lemma 4.35) The important observations to prove this lemma, are that the semantics of $\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle$ is defined in terms of the semantics of $\langle\, p, s :: \sigma \,\rangle$ (see Rules 32, 33, and 34), and that the hypotheses of each deduction rule cannot depend on the values of *invisible* identifiers of the stack σ in $s :: \sigma$. So, suppose $i \in dom(\sigma)$ (if not, the proof is trivial). Since we also know that $i \in s$, the identifier i of σ is invisible in the stack $s :: \sigma$ (see Definition A.11). Therefore, its value in σ cannot influence the set of deduction rules that apply to $\langle\, p, s :: \sigma \,\rangle$ and, similarly, it cannot influence the set of deduction rules that apply to $\langle\, p, s :: \sigma[c/i] \,\rangle$. So, the proof for the termination clause of the lemma is finished. To finish the proof for the action transition and the delay transition of the lemma, we use Lemma 4.50 to derive that the resulting stack σ' equals σ, $\sigma[c_0/i_0]$, or $\sigma[c_0/i_0][c_0'/i_0']$. Therefore, a transition can only change the values of visible identifiers of a stack. Since i is invisible in σ of $s :: \sigma$, as well as in $\sigma[c/i]$ of $s :: \sigma[c/i]$, its value can neither be changed in σ' of $s' :: \sigma'$ nor in $\sigma'[c/i]$ of $s' :: \sigma'[c/i]$. □

4.12 · Encapsulation operator

In Definition 4.36, we define the encapsulation operator '∂'. A process $\partial_A(p)$ encapsulates all actions that p can perform and occur in the set A by disabling them.

Definition 4.36 · (Encapsulation operator) *The encapsulation operator has the following signature:*

$$\partial\_ \quad : \quad \mathcal{P}(Action) \times P \to P.$$

The deduction rules for the encapsulation operator are listed in Table 4.8.

$$\frac{\langle\, p, \sigma \,\rangle\downarrow}{\langle\, \partial_A(p), \sigma \,\rangle\downarrow}\,35 \qquad \frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle,\ a \notin A}{\langle\, \partial_A(p), \sigma \,\rangle \xrightarrow{a} \langle\, \partial_A(p'), \sigma' \,\rangle}\,36$$

$$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}{\langle\, \partial_A(p), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \partial_A(p'), \sigma' \,\rangle}\,37$$

Table 4.8 · Deduction rules for the encapsulation operator.

Rule 35 states that an encapsulation process can terminate, if its process argument can. Rule 36 states that an encapsulation process can perform an action if its process argument can perform that action and if that action is not in A, the set of actions to be encapsulated. With respect to delay steps, encapsulation has no effect. Rule 37 states that an encapsulation process can perform a delay, if its process argument can perform that delay too.

Lemma 4.37 · *Let A and A' be sets of actions and let p be a process, then*

$$\partial_A(\partial_{A'}(p)) \leftrightarrows \partial_{A \cup A'}(p).$$

Proof · (Lemma 4.37) We have to prove $\partial_A(\partial_{A'}(p)) \leftrightarrows \partial_{A\cup A'}(p)$ for all processes p and sets of actions A and A'. In this case, we define a relation $R \subseteq P \times P$ such that $(\partial_A(\partial_{A'}(p)), \partial_{A\cup A'}(p)) \in R$ and show that R is a bisimulation. We define R as

$$R = \{(p, p)\} \cup \{(\partial_A(\partial_{A'}(p)), \partial_{A\cup A'}(p))\}.$$

We will now show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs for the pairs of the form (p, p) are trivial, we will only consider pairs of the form $(\partial_A(\partial_{A'}(p)), \partial_{A\cup A'}(p))$. So, assume $p \equiv \partial_A(\partial_{A'}(x))$ and $q \equiv \partial_{A\cup A'}(x)$, for some process x.

Condition 1: We have to prove $\forall\sigma{:}\langle\, p,\sigma\,\rangle\!\downarrow \Leftrightarrow \langle\, q,\sigma\,\rangle\!\downarrow$. Using Rule 35, the following computation proves that it holds: $\langle\, p,\sigma\,\rangle\!\downarrow \Leftrightarrow \langle\, \partial_A(\partial_{A'}(x)),\sigma\,\rangle\!\downarrow \Leftrightarrow \langle\, x,\sigma\,\rangle\!\downarrow \Leftrightarrow \partial_{A\cup A'}(x) \Leftrightarrow \langle\, q,\sigma\,\rangle\!\downarrow$.

Condition 2: We have to prove $\forall\sigma,a,p',\sigma' : \langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle \Rightarrow \exists q' : \langle\, q,\sigma\,\rangle \xrightarrow{a} \langle\, q',\sigma'\,\rangle$ and $(p',q') \in R$. So, assume $\langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle$ holds. Since Rule 36 is the only rule that applies, we have $\langle\, x,\sigma\,\rangle \xrightarrow{a} \langle\, x',\sigma'\,\rangle$ such that $a \notin A$, $a \notin A'$, and $p' \equiv \partial_A(\partial_{A'}(x'))$. Therefore, $a \notin A \cup A'$ and we can use Rule 36 to obtain $\langle\, \partial_{A\cup A'}(x),\sigma\,\rangle \xrightarrow{a} \langle\, \partial_{A\cup A'}(x'),\sigma'\,\rangle$. So, take $q' \equiv \partial_{A\cup A'}(x')$ and note that $(p',q') \in R$.

Condition 3: We have to prove $\forall\sigma,a,q',\sigma' : \langle\, q,\sigma\,\rangle \xrightarrow{a} \langle\, q',\sigma'\,\rangle \Rightarrow \exists p' : \langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle$ and $(p',q') \in R$. So, assume $\langle\, q,\sigma\,\rangle \xrightarrow{a} \langle\, q',\sigma'\,\rangle$ holds. Since Rule 36 is the only rule that applies, we have $\langle\, x,\sigma\,\rangle \xrightarrow{a} \langle\, x',\sigma'\,\rangle$ such that $q' \equiv \partial_{A\cup A}(x')$ and $a \notin A\cup A'$. Therefore, we have $a \notin A$ as well as $a \notin A'$, and we use Rule 36 to derive $\langle\, \partial_A(\partial_{A'}(x)),\sigma\,\rangle \xrightarrow{a} \langle\, \partial_A(\partial_{A'}(x')),\sigma'\,\rangle$. So, take $p' \equiv \partial_A(\partial_{A'}(x'))$ and note that $(p',q') \in R$.

Condition 4: We have to prove $\forall\sigma,d,p',\sigma' : \langle\, p,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, p',\sigma'\,\rangle \Rightarrow \exists q' : \langle\, q,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, q',\sigma'\,\rangle$ and $(p',q') \in R$. Since the proof is almost similar to the proof of Condition 2, we will not work out the details. In fact, the proof is slightly simpler, since the sets A and A' do not influence delay behaviour of p and q.

Condition 5: As Condition 4.

4.13 · Maximal progress operator

In Definition 4.38, we define the maximal progress operator 'π'. A process $\pi(p)$ can delay only if p can delay and p cannot execute an action. We need this operator in order to establish a desired communicational behaviour. That is, both the send and the receive process must be able to delay, but if two of these processes can communicate, they should not delay.

The maximal progress operator is a kind of priority operator [14] that assigns action transitions a higher priority than delay transitions. Consequently, it makes actions undelayable or *urgent*. In fact, the maximal progress operator of χ_σ is similar to a particular instantiation of the *urgency* operator $\mathcal{U}_U(p)$ described in [115] (the particular instantiation results from taking $U = Action$). The operator $\mathcal{U}_U(p)$

makes actions in the set U urgent: if p can, at some time, execute actions from U, then it cannot delay at that time.

Definition 4.38 · (Maximal progress operator) *The maximal progress operator has the following signature:*

$$\pi \quad : \quad P \to P.$$

The deduction rules for the maximal progress operator are listed in Table 4.9.

$$\frac{\langle\, p, \sigma \,\rangle\!\downarrow}{\langle\, \pi(p), \sigma \,\rangle\!\downarrow}\,38 \qquad \frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, \pi(p), \sigma \,\rangle \xrightarrow{a} \langle\, \pi(p'), \sigma' \,\rangle}\,39 \qquad \frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, p, \sigma \,\rangle \not\overset{}{\mapsto}}{\langle\, \pi(p), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \pi(p'), \sigma' \,\rangle}\,40$$

Table 4.9 · Deduction rules for the maximal progress operator.

Rule 38 states that the maximal progress operator can terminate, if its process argument can terminate. Concerning action behaviour, the maximal progress operator has no effect; Rule 39 states that if a process p can perform an action, then also the maximal progress process $\pi(p)$ can perform that action. The delay behaviour of the maximal progress operator is more interesting; the maximal progress operator postpones delay behaviour as long as its process argument can perform actions. So, a maximal progress operator only performs a delay if its process argument p can perform that delay *and* if p cannot perform actions as defined by Rule 40.

Lemma 4.39 · *Let p be a process, then*

$$\pi(\pi(p)) \leftrightarrows \pi(p).$$

Proof · (Lemma 4.39) We have to prove that $\pi(\pi(p)) \leftrightarrows \pi(p)$ for all processes p. In this case, we define a relation $R \subseteq P \times P$ such that $(\pi(\pi(p)), \pi(p)) \in R$ and R is a bisimulation. We define R as

$$R = \{(p, p)\} \cup \{(\pi(\pi(p)), \pi(p))\}$$

and show that all pairs $(p, q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs are trivial for the pairs (p, p), we only give the proofs for the pairs $(\pi(\pi(p)), \pi(p))$.

Suppose $p \equiv \pi(\pi(x))$ and $q \equiv \pi(x)$. So, note that $p \equiv \pi(q)$.

Condition 1: We have to prove that $\langle p, \sigma \rangle\downarrow \Leftrightarrow \langle q, \sigma \rangle\downarrow$. Using Rule 38 we obtain that if $\langle p, \sigma \rangle\downarrow$ then also $\langle q, \sigma \rangle\downarrow$, and that if $\langle q, \sigma \rangle\downarrow$ then also $\langle p, \sigma \rangle\downarrow$. So, $\langle p, \sigma \rangle\downarrow \Leftrightarrow \langle q, \sigma \rangle\downarrow$.

Condition 2: We have to prove that $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle \Rightarrow \exists q' : \langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle$ and $(p', q') \in R$. Suppose $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$. Then, according to Rule 39, we have $\langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle$ and $p' \equiv \pi(q')$. So, $(p', q') \in R$.

Condition 3: The proof is similar to the proof of Condition 2.

Condition 4: We have to prove that $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle \Rightarrow \exists q' : \langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle$ and $(p', q') \in R$. Suppose $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$. Then, according to Rule 40, we have $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle$ and $p' \equiv \pi(q')$. So, $(p', q') \in R$.

Condition 5: We have to prove that $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle \Rightarrow \exists p' : \langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$ and $(p', q') \in R$. Suppose $\langle q, \sigma \rangle \overset{d}{\longmapsto} \langle q', \sigma' \rangle$. Using Rule 40 we obtain that $\langle x, \sigma \rangle \not\rightarrow$. According to Rule 39 we then also have $\langle \pi(x), \sigma \rangle \not\rightarrow$. So, $\langle q, \sigma \rangle \not\rightarrow$. Knowing this, Rule 40 gives $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$, where $p' \equiv \pi(q')$. Consequently, also $(p', q') \in R$. □

4.14 · Abstraction operator

In Definition 4.40, we define the abstraction operator 'τ'. A process $\tau_A(p)$ 'hides' all actions that p can perform and occur in the set A by renaming those actions to the internal action τ.

Definition 4.40 · (Abstraction operator) *The abstraction operator has the following signature:*

$$\tau\_ \quad : \quad \mathcal{P}(Action) \times P \rightarrow P.$$

The deduction rules for the abstraction operator are listed in Table 4.10.

Rule 41 states that an abstraction process can terminate if its process argument can. Action steps for abstraction processes are described by Rule 42 and 43. Rule 42 states that if the process argument can perform an action and this action is not in A, the set of actions to be abstracted from, then the abstraction process can perform that action too. On the other hand, if the process argument can perform

$$\frac{\langle\, p,\sigma\,\rangle\downarrow}{\langle\, \tau_A(p),\sigma\,\rangle\downarrow}\,41 \qquad \frac{\langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle,\ a\notin A}{\langle\, \tau_A(p),\sigma\,\rangle \xrightarrow{a} \langle\, \tau_A(p'),\sigma'\,\rangle}\,42 \qquad \frac{\langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle,\ a\in A}{\langle\, \tau_A(p),\sigma\,\rangle \xrightarrow{\tau} \langle\, \tau_A(p'),\sigma'\,\rangle}\,43$$

$$\frac{\langle\, p,\sigma\,\rangle \overset{d}{\longmapsto} \langle\, p',\sigma'\,\rangle}{\langle\, \tau_A(p),\sigma\,\rangle \overset{d}{\longmapsto} \langle\, \tau_A(p'),\sigma'\,\rangle}\,44$$

Table 4.10 · Deduction rules for the abstraction operator.

an action that is in A, then the abstraction process can perform the 'invisible' action τ. This is stated in Rule 43. With respect to delay steps, encapsulation has no effect. Rule 44 states that an abstraction process can perform a delay if its process argument can perform that delay too.

Lemma 4.41 · *Let A and A' be sets of actions and let p be a process, then*

$$\tau_A(\tau_{A'}(p)) \leftrightarrows \tau_{A\cup A'}(p).$$

Proof · (Lemma 4.41) We have to prove $\tau_A(\tau_{A'}(p)) \leftrightarrows \tau_{A\cup A'}(p)$ for all processes p and sets of actions A and A'. In this case, we define a relation $R \subseteq P \times P$ such that $(\tau_A(\tau_{A'}(p)), \tau_{A\cup A'}(p)) \in R$ and show R is a bisimulation. We define R as

$$R = \{(p,p)\} \cup \{(\tau_A(\tau_{A'}(p)), \tau_{A\cup A'}(p))\}.$$

We will now show that all pairs $(p,q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proofs for the pairs of the form (p,p) are trivial, we will only consider pairs of the form $(\tau_A(\tau_{A'}(p)), \tau_{A\cup A'}(p))$. So, assume $p \equiv \tau_A(\tau_{A'}(x))$ and $q \equiv \tau_{A\cup A'}(x)$, for some process x.

Condition 1: We have to prove $\forall\sigma{:}\langle\, p,\sigma\,\rangle\downarrow \Leftrightarrow \langle\, q,\sigma\,\rangle\downarrow$. Using Rule 41, the following computation proves that it holds: $\langle\, p,\sigma\,\rangle\downarrow \Leftrightarrow \langle\, \tau_A(\tau_{A'}(x)),\sigma\,\rangle\downarrow \Leftrightarrow \langle\, x,\sigma\,\rangle\downarrow \Leftrightarrow \tau_{A\cup A'}(x) \Leftrightarrow \langle\, q,\sigma\,\rangle\downarrow$.

Condition 2: We have to prove $\forall\sigma,a,p',\sigma' : \langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle \Rightarrow \exists q' : \langle\, q,\sigma\,\rangle \xrightarrow{a} \langle\, q',\sigma'\,\rangle$ and $(p',q') \in R$. So, assume $\langle\, p,\sigma\,\rangle \xrightarrow{a} \langle\, p',\sigma'\,\rangle$ holds. There are two possibilities, since Rule 42 or Rule 43 applies.

Rule 42 applies to $\tau_A(\tau_{A'}(x))$: Now, we can derive $\langle\, \tau_{A'}(x), \sigma \,\rangle \xrightarrow{a} \langle\, \tau_{A'}(x'), \sigma' \,\rangle$ and $a \notin A$. Since the rules that apply are Rule 42 and Rule 43, we can make the same distinction again.

Rule 42 applies to $\tau_{A'}(x)$: Now we can derive $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x', \sigma' \,\rangle$ and $a \notin A'$. Since $a \notin A$ and $a \notin A'$ we have $a \notin A \cup A'$. Therefore, using Rule 42, we can derive $\langle\, \tau_{A\cup A'}(x), \sigma \,\rangle \overset{a}{\longmapsto} \langle\, \tau_{A\cup A'}(x'), \sigma \,\rangle$. So, take $q' \equiv \tau_{A\cup A'}(x')$ and note that $(p', q') \in R$.

Rule 43 applies to $\tau_{A'}(x)$: Now we can derive $\langle\, x, \sigma \,\rangle \xrightarrow{a'} \langle\, x', \sigma' \,\rangle$ for some a' such that $a' \in A'$. Furthermore, we have $a \equiv \tau$. Since $a' \in A'$, we also have $a' \in A \cup A'$ and therefore, using Rule 43, we obtain $\langle\, \tau_{A\cup A'}(x), \sigma \,\rangle \xrightarrow{\tau} \langle\, \tau_{A\cup A'}(x'), \sigma' \,\rangle$. So, take $q' \equiv \tau_{A\cup A'}(x')$ and note that $(p', q') \in R$.

Rule 43 applies to $\tau_A(\tau_{A'}(x))$: Now, we can derive $\langle\, \tau_{A'}(x), \sigma \,\rangle \xrightarrow{a'} \langle\, \tau_{A'}(x'), \sigma' \,\rangle$ such that $a' \in A$ and $p' \equiv \tau_{A'}(x')$. Furthermore, we have $a \equiv \tau$. Since the rules that apply are Rule 42 and Rule 43, we can make the same distinction again.

Rule 42 applies to $\tau_{A'}(x)$: Now, we can derive that $\langle\, x, \sigma \,\rangle \xrightarrow{a'} \langle\, x', \sigma' \,\rangle$ such that $a' \notin A'$. However, since $a' \in A$ we have $a' \in A \cup A'$ and therefore, using Rule 43, we derive $\langle\, \tau_{A\cup A'}(x), \sigma \,\rangle \xrightarrow{\tau} \langle\, \tau_{A\cup A'}(x'), \sigma' \,\rangle$. Therefore, take $q' \equiv \tau_{A\cup A'}(x')$ and note that $(p', q') \in R$.

Rule 43 applies to $\tau_{A'}(x)$: Now, we can derive that $\langle\, x, \sigma \,\rangle \xrightarrow{a''} \langle\, x', \sigma' \,\rangle$ such that $a'' \in A'$. Therefore, we have $a'' \in A \cup A'$. So, using Rule 43, we find $\langle\, \tau_{A\cup A'}(x), \sigma \,\rangle \xrightarrow{\tau} \langle\, \tau_{A\cup A'}(x'), \sigma' \,\rangle$. Therefore, take $q' \equiv \tau_{A\cup A'}(x')$ and note that $(p', q') \in R$.

Condition 3: We have to prove $\forall \sigma, a, q', \sigma' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$ and $(p', q') \in R$. So, assume $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$ holds. There are two possibilities, since Rule 42 or Rule 43 applies.

Rule 42 applies to $\tau_{A\cup A'}(x)$: Now, we have $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x', \sigma' \,\rangle$ and $a \notin A \cup A'$. Therefore, we have $a \notin A$ and $a \notin A'$. According to Rule 42 we can derive $\langle\, \tau_A(\tau_{A'}(x)), \sigma \,\rangle \xrightarrow{a} \langle\, \tau_A(\tau_{A'}(x')), \sigma' \,\rangle$. So, take $p' \equiv \tau_A(\tau_{A'}(x'))$ and note that $(p', q') \in R$.

Rule 43 applies to $\tau_{A\cup A'}(x)$: Now, we have $\langle\, x, \sigma \,\rangle \xrightarrow{a'} \langle\, x', \sigma' \,\rangle$ for some a', such that $a' \in A \cup A'$. Furthermore, we have $a \equiv \tau$. We distinguish the following cases.

$a' \in A$: Now we can derive $\langle\, \tau_A(\tau_{A'}(x)), \sigma \,\rangle \xrightarrow{\tau} \langle\, \tau_A(\tau_{A'}(x')), \sigma' \,\rangle$, no matter if Rule 42 or Rule 43 applies to $\langle\, \tau_{A'}(x), \sigma \,\rangle$. So, take $p' \equiv \tau_A(\tau_{A'}(x'))$ and note that $(p', q') \in R$.

$a' \notin A$: Now, we have $a' \in A'$. So, $\langle\, \tau_A(\tau_{A'}(x)), \sigma \,\rangle \xrightarrow{\tau} \langle\, \tau_A(\tau_{A'}(x')), \sigma' \,\rangle$ holds. So, take $p' \equiv \tau_A(\tau_{A'}(x'))$ and note that $(p', q') \in R$.

Condition 4: We have to prove $\forall \sigma, d, p', \sigma' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$ and $(p', q') \in R$. Since the proof is almost similar to the proof of Condition 4 of Lemma 4.37, details are omitted.

Condition 5: We have to prove $\forall \sigma, d, q', \sigma' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$ and $(p', q') \in R$. Since the proof is almost similar to the proof of Condition 4 of Lemma 4.37, details are omitted. □

4.15 · Stratification of the deduction rules

As mentioned in Section 4.2, we have to show that the TSS constituted by the deduction rules given in the previous sections is meaningful. A TSS is meaningful if there exists an LTS with relations that coincide exactly with the positive formulas that can be proven from the TSS. To show that such an LTS exists for a given TSS, it is sufficient to show that the deduction rules of the TSS are *stratifiable*. The following definition stems from [1] and is adapted to the TSS of χ_σ.

Definition 4.42 · (Stratification) *A mapping S from positive formulas to natural numbers is a* stratification *for the SOS of χ_σ if for every deduction rule $\frac{H}{c}$ and every closed substitution θ,*

- *for $h \in H$ of the forms 1–4 of Definition 4.3 (the positive hypotheses), $S(\theta(h)) \leq S(\theta(c))$; and*
- *for $h \in H$ of the forms 5–7 of Definition 4.3 (the negative hypotheses):*

  form 5: if $h = \neg\exists p' \in C(P), \sigma', a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, then for all closed terms p': $S(\langle\, \theta(p), \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle) < S(\theta(c))$; and

  form 6: if $h = \neg\exists p' \in C(P), \sigma', a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, then for all closed terms p': $S(\langle\, \theta(p), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle) < S(\theta(c))$; and

  form 7: if $h = \neg\langle\, p, \sigma \,\rangle\!\downarrow$, $S(\langle\, \theta(p), \sigma \,\rangle\!\downarrow) < S(\theta(c))$; respectively.

A TSS with a stratification is stratifiable.

We will now define a stratification S for the TSS of χ_σ. In the definition of S, the function *ops* returns the number of process operators in a given closed process term.

Definition 4.43 · *Let p and p' be processes. The function $ops : P \to N$ is defined recursively by*

- $ops(p) = 0$, *if* $p \in \{\delta, \varepsilon, \mathsf{skip}, x := e, m\,!\,e, m\,?\,e, \Delta e\}$,
- $ops(e :\rightarrow p) = 1 + ops(p)$,
- $ops(p \between p') = 1 + ops(p) + ops(p')$,
- $ops(p \;;\; p') = 1 + ops(p) + ops(p')$,
- $ops(p^*) = 1 + ops(p)$,
- $ops(p \parallel p') = 1 + ops(p) + ops(p')$,
- $ops([\![\, s \mid p \,]\!]) = 1 + ops(p)$,
- $ops(\partial_A(p)) = 1 + ops(p)$,
- $ops(\pi(p) = 1 + ops(p)$,
- $ops(\tau_A(p)) = 1 + ops(p)$.

Definition 4.44 · *Let p and p' be closed process terms, σ and σ' be stacks, a an action, and d a positive real number. The function S from positive formulas to natural numbers is defined by*

1. $S(\textsc{true}(e)) = 0$,
2. $S(\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle) = ops(p)$,
3. $S(\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle) = ops(p)$,
4. $S(\langle\, p, \sigma \,\rangle\!\downarrow) = ops(p)$.

Lemma 4.45 · *The TSS of χ_σ is stratifiable.*

Proof · (Lemma 4.45) This follows from the fact that S of Definition 4.44 is a stratification for the TSS of χ_σ. □

4.16 · Bisimulation as a congruence

In order to make calculations with bisimulation equivalence more practical, it would be nice to know that substitution of processes by bisimilar processes is a valid operation. For instance, if we know that $p \leftrightarrows q$, then it would be nice that by substituting q for p in the right hand side of $p \mathbin{;} r \leftrightarrows p \mathbin{;} r$, we derive $p \mathbin{;} r \leftrightarrows q \mathbin{;} r$. This property is called *congruence*. A binary equivalence relation $R \subseteq S \times S$ on some set S is a congruence if for all n-ary operators $\otimes$ to construct elements of S and for all terms $t_1, \ldots, t_n, t'_1, \ldots, t'_n$, with $t_1 R t'_1, \ldots, t_n R t'_n$, we have $\otimes(t_1, \ldots, t_n) R \otimes (t'_1, \ldots, t'_n)$. Bisimulation equivalence is a congruence is a powerful tool to perform calculations on processes. In fact, in some of the proofs given in previous sections, we implicitly assumed that bisimulation equivalence is a congruence. For example, see the proof of Lemma 4.23 and in particular the proof of Condition 5.

The standard approach to show that strong bisimulation on processes is a congruence is to use *congruence theorems* for specific syntactic formats of the deduction rules [1, 192, 17, 89, 96, 36, 187]. However, these theorems are usually presented in a one-sorted setting where each term denotes a process. Since we have many-sorted terms denoting expressions, stacks, states, real numbers, sets of actions, and processes, it is unclear whether this approach can be applied to the TSS of χ_σ. Moreover, the semantics of these terms is defined in different styles: AS for data types, SOS for processes, and ordinary mathematical definitions for states and stacks. To make the situation even worse, χ_σ has *variable binding* operators, which are also not accounted for in the standard congruence theorems. For example, the implicit operators to evaluate expressions in states and stacks, denote by $s(e)$ and $\sigma(e)$, binds the identifiers occurring in e that are in the domains of s and σ, respectively. Notice that χ_σ does not have variable binding *process* operators.

Fortunately, the problems of many-sortedness and variable binding operators have been addressed by others [139, 72]. Furthermore, in [139], also a solution to the problem of different definition styles is given. The author introduces the notion of *given sorts*: sorts for which a well established semantics exists. He argues that it is impractical and unnecessary to redefine the semantics of given sorts if the sole purpose of this exercise would be to apply a particular congruence theorem. Instead, one only has to distinguish given sorts as such and show that their semantics defines an equivalence relation on terms of these sorts. After that, one can use the congruence theorem based on the so-called *relaxed panth* format

defined in [139]. This congruence theorem says that if a TSS is stratifiable and it is in relaxed panth format, bisimulation equivalence is a congruence. An application of the relaxed panth format in the area of timed process algebras is given in [16].

The TSS of χ_σ should define the operational semantics of processes. Therefore, we consider the remaining sorts as given sorts. These sorts contain expressions, states, stacks, real numbers, and sets (of actions). By the definition of derivation (of equalities between expressions) in MEL, Definition C.11, it is clear that there is a well defined equivalence relation on expressions. For states and stacks, equivalence relations are defined in Definition A.6 and A.12, respectively. It is well known that conventional equality on real numbers is an equivalence relation. Similarly equality on sets is an equivalence relation, too. So, we can consider expressions, states, stacks, and sets of actions as given sorts. Next, the deduction rules have to be checked for conformance to the relaxed panth format. Concretely, each deduction rule should satisfy the following conditions (these conditions stem from [139] and are adapted for χ_σ).

1. For each positive hypothesis of the form $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$ or $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$, process p' should be a variable.

2. For each conclusion of the form $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$ or $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$, process p should contain at most one process operator and each process argument of this operator should be a variable.

3. The variables p' of positive hypotheses of the form $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$ and $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$ and the variables occurring in p of conclusions of the form $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$ or $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$ are mutually distinct.

Inspection of the deduction rules shows that they satisfy these conditions. Therefore, the SOS of χ_σ is in relaxed panth format. Together with the stratification of the SOS of χ_σ, see Section 4.15, this means that bisimulation equivalence is a congruence.

4.17 · Properties

In this section we discuss properties of χ_σ processes. Amongst others, we reflect on the timing aspects relevant for χ_σ as discussed in Section 4.3: continuous time, time factorisation, and time determinism.

In χ_σ, we can specify that a process is able to delay a certain number of time units. At the same time, that process is then also able to delay less. Because we have continuous time, this implies that if a process can perform a delay d greater than zero, than it is always able to also perform a delay d' smaller than d. This is expressed in Lemma 4.46 below.

Lemma 4.46 · *Let p and p_d be processes; σ and σ_d be stacks; and d and d' be positive real numbers such that $d' < d$. Then*

$$\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle \Rightarrow \exists p_{d'}, \sigma_{d'} : \langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle.$$

Proof · (Lemma 4.46) We have to prove that $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle \Rightarrow \exists p_{d'}, \sigma_{p_{d'}} : \langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$ for all p and p_d processes; σ and σ_d stacks; and d and d' positive real numbers such that $d' < d$.

First, we prove that this holds for the defined atomic processes. Then we prove by induction that it also holds for every defined process operator.

Let p be an atomic process. Then we end up with the following cases: $p \equiv \varepsilon$, $p \equiv \delta$, $p \equiv \mathsf{skip}$, $p \equiv x := e$, $p \equiv m\,!\,e$, $p \equiv m\,?\,x$, and $p \equiv \Delta e$. In case $p \equiv \varepsilon$, $p \equiv \delta$, $p \equiv \mathsf{skip}$, or $p \equiv x := e$, the hypothesis does not hold, which then concludes the proof. For the cases $p \equiv m\,!\,e$ and $p \equiv m\,?\,x$, Rules 7 and 8 show that if $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$, then for $d' < d$ we have $\langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$ with $p_{d'} \equiv p_d$. For the case $p \equiv \Delta e$, Rule 9 shows that if $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$ with $p_d \equiv \Delta e - d$, then for $d' < d$ we have $\langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$ with $p_{d'} \equiv \Delta e - d'$. This concludes the proof for the defined atomic processes.

In case p is a non-atomic process and if $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$, then we can apply Rule 12, 20, 21, 22, 23, 15, 16, 26, 30, 31, 34, 37, 40, or 44 and find by induction that $\langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$. □

The fact that χ_σ has time factorisation is expressed by Lemma 4.47 below. As can be seen, the smallest delay can be factored out of the alternative composition.

Lemma 4.47 · (Time factorisation) *Let e and e' be expressions, such that $e \geq 0$ and $e' \geq 0$, and let p be a process, then*

$$\Delta e \;;\; p \between \Delta e + e' \leftrightarrows \Delta e \;;\; (p \between \Delta e').$$

Proof · (Lemma 4.47) We have to prove $\Delta e \;;\; p \between \Delta e + e' \leftrightarrows \Delta e \;;\; (p \between \Delta e')$ for all processes p and expressions e and e', such that $e \geq 0$ and $e' \geq 0$. In this case, we

define a relation R such that $(\Delta e \ ; \ p \parallel \Delta e + e', \Delta e \ ; \ (p \parallel \Delta e') \in R$ and show that R is a bisimulation. We define R as

$$R = \{(p,p)\} \cup \{(\Delta e \ ; \ p \parallel \Delta e + e', \Delta e \ ; \ (p \parallel \Delta e') \mid e' \geq 0\}.$$

We will now show that all pairs $(p,q) \in R$ satisfy the five bisimulation conditions of Definition 4.5. Since the proof for the pairs of the form (p,p) is trivial, we will only consider pairs of the form $(\Delta e \ ; \ p \parallel \Delta e + e', \Delta e \ ; \ (p \parallel \Delta e')$.

So, suppose $p \equiv \Delta e \ ; \ x \parallel \Delta e + e'$ and $q \equiv \Delta e \ ; \ (x \parallel \Delta e')$ for some process x.

Condition 1: We have to prove $\forall \sigma : \langle\, p, \sigma \,\rangle\downarrow \Leftrightarrow \langle\, q, \sigma' \,\rangle\downarrow$. The following computation completes the proof of Condition 1. In the computation, deduction Rules 2, 17, and 13 are used.

$$\begin{array}{ll}
 & \langle\, p, \sigma \,\rangle\downarrow \\
\Leftrightarrow & \langle\, \Delta e \ ; \ x \parallel \Delta e + e', \sigma \,\rangle\downarrow \\
\Leftrightarrow & \langle\, \Delta e \ ; \ x, \sigma \,\rangle\downarrow \vee \langle\, \Delta e + e', \sigma \,\rangle\downarrow \\
\Leftrightarrow & (\langle\, \Delta e, \sigma \,\rangle\downarrow \wedge \langle\, x, \sigma \,\rangle\downarrow) \vee \sigma(e + e') = 0 \\
\Leftrightarrow & (\sigma(e) = 0 \wedge \langle\, x, \sigma \,\rangle\downarrow) \vee \sigma(e + e') = 0 \\
\Leftrightarrow & (\sigma(e) = 0 \wedge \langle\, x, \sigma \,\rangle\downarrow) \vee \sigma(e) + \sigma(e') = 0 \\
\Leftrightarrow & \{\text{Expressions } e \text{ and } e' \text{ satisfy } e \geq 0 \text{ and } e' \geq 0\} \\
 & (\sigma(e) = 0 \wedge \langle\, x, \sigma \,\rangle\downarrow) \vee (\sigma(e) = 0 \wedge \sigma(e') = 0) \\
\Leftrightarrow & \sigma(e) = 0 \wedge (\langle\, x, \sigma \,\rangle\downarrow \vee \sigma(e') = 0) \\
\Leftrightarrow & \sigma(e) = 0 \wedge (\langle\, x, \sigma \,\rangle\downarrow \vee \langle\, \Delta e', \sigma \,\rangle\downarrow) \\
\Leftrightarrow & \langle\, \Delta e, \sigma \,\rangle\downarrow \wedge \langle\, x \parallel \Delta e', \sigma \,\rangle\downarrow \\
\Leftrightarrow & \langle\, \Delta e \ ; \ (x \parallel \Delta e'), \sigma \,\rangle\downarrow \\
\Leftrightarrow & \langle\, q, \sigma \,\rangle\downarrow.
\end{array}$$

Condition 2: We have to prove $\forall a, \sigma', p' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$. So, suppose we have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, where $p \equiv \Delta e \,;\, x \parallel \Delta e + e'$. Since delay processes cannot execute actions, we know that x performs the action. So, Rules 19 and 14 apply and we obtain $\langle\, \Delta e, \sigma \,\rangle\downarrow$ and $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. Note that according to Rule 2, we can derive $\sigma(e) = 0$. Finally, using Rules 19 and 14 again, we derive $\langle\, \Delta e \ ; \ (x \parallel \Delta e'), \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. So take $q' \equiv p'$ and note that $(p', q') \in R$.

Condition 3: We have to prove $\forall a, \sigma', q' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. So, assume $\langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$ where $q \equiv \Delta e \ ; \ (x \parallel \Delta e')$. Since delay processes cannot execute actions, we know that x performs the action. So,

Rules 19 and 14 apply and we obtain $\langle\, \Delta e, \sigma \,\rangle\!\downarrow$ and $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$. Note that according to Rule 2, we can derive $\sigma(e) = 0$. Finally, using Rules 19 and 14 again, we derive $\langle\, \Delta e \;;\; x \parallel \Delta e + e', \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle$. So, take $p' \equiv q'$ and note that $(p', q') \in R$.

Condition 4: We have to prove $\forall d, \sigma', p' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \;\Rightarrow\; \exists q' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$. So, suppose we have $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$, where $p \equiv \Delta e \;;\; x \parallel \Delta e + e'$. Since $e \geq 0$ we can distinguish the following cases.

$\sigma(e) = 0$: Since $e' \geq 0$ we can distinguish two cases.

$\sigma(e') = 0$: Using Rules 22 and 15 we can derive $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$. Using the same rules again, we can derive $\langle\, \Delta e \;;\; (x \parallel \Delta e'), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$. So, take $q' \equiv p'$ and note that $(p', q') \in R$.

$\sigma(e') > 0$: Note that we can derive $\sigma(e') \geq d$, otherwise the assumption $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$ cannot be true. Since x can or cannot delay, there are two cases to distinguish. If x cannot delay, we derive $\langle\, \Delta e \;;\; x \parallel \Delta e + e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e + e' - d, \sigma' \,\rangle$. So, $p' \equiv \Delta e + e' - d$. Using Rules 22 and 15, we can now derive $\langle\, \Delta e \;;\; (x \parallel \Delta e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e' - d, \sigma' \,\rangle$. Furthermore, since $e = 0$, we have $e + e' = e'$ and therefore we have $p' \equiv \Delta e + e' - d = \Delta e' - d$. So, take $q' \equiv p'$ and note that $(p', q') \in R$. On the other hand, if x can delay, we can derive that it can delay at least d time units. So, we get $\langle\, \Delta e \;;\; x \parallel \Delta e + e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \parallel \Delta e + e' - d, \sigma' \,\rangle$ for some process x' such that $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x', \sigma' \,\rangle$ and $p' \equiv x' \parallel \Delta e + e' - d$. Using Rules 22 and 15, we can now derive $\langle\, \Delta e \;;\; (x \parallel \Delta e'), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x' \parallel \Delta e' - d, \sigma' \,\rangle$. Furthermore, since $e = 0$, we have $e + e' = e'$ and therefore we have $p' \equiv x' \parallel \Delta e + e' - d = x' \parallel \Delta e' - d$. So, take $q' \equiv p'$ and note that $(p', q') \in R$.

$\sigma(e) > 0$: Note that we can derive $\sigma(e) \geq d$, otherwise $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$ cannot be true. Furthermore, since $e' \geq 0$, we also have $\sigma(e + e') \geq d$. So, the delay transition of p is actually $\langle\, \Delta e \;;\; x \parallel \Delta e + e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e - d \;;\; x \parallel \Delta e + e' - d, \sigma' \,\rangle$ and we find $p' \equiv \Delta e - d \;;\; x \parallel \Delta e + e' - d$. Using arithmetic, we can rewrite p' to $\Delta e - d \;;\; x \parallel \Delta e - d + e'$. Furthermore, for process $q \equiv \Delta e \;;\; (x \parallel \Delta e')$ we can derive $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e - d \;;\; (x \parallel \Delta e'), \sigma' \,\rangle$ using Rule 20. So, take $q' \equiv \Delta e - d \;;\; (x \parallel \Delta e')$ and note that $(p', q') \in R$.

Condition 5: We have to prove $\forall d, q', \sigma' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \;\Rightarrow\; \exists p' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$. So, assume $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$ where $q \equiv \Delta e \;;\; (x \parallel \Delta e')$. Since $e \geq 0$ we can distinguish two cases.

$\sigma(e) = 0$: Now, using Rules 2 and 22, we can derive $\langle\, x \parallel \Delta e' \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$. We distinguish two cases.

$\sigma(e') = 0$: Now, using Rule 15, we can derive $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$. Furthermore, using Rules 2, 22, and 15, we obtain $\langle\, \Delta e \,;\, x \parallel \Delta e + e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$. So, take $p' \equiv q'$ and note that $(p', q') \in R$.

$\sigma(e') > 0$: Note that we can derive $\sigma(e') \geq d$, otherwise the assumption $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$ cannot be true. Since x can or cannot delay, there are two cases to distinguish. If x cannot delay, we find $\langle\, \Delta e \,;\, (x \parallel \Delta e'), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e' - d, \sigma' \,\rangle$. So, $q' \equiv \Delta e' - d$. According to Rule 15, we can derive $\langle\, \Delta e \;;\; x \parallel \Delta e + e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e + e' - d, \sigma' \,\rangle$. Furthermore, since $e = 0$, we have $e + e' = e'$ and therefore we also have $\langle\, \Delta e \;;\; x \parallel \Delta e + e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e' - d, \sigma' \,\rangle$. So, take $p' \equiv q'$ and note that $(p', q') \in R$.

$\sigma(e) > 0$: Note that we can derive $\sigma(e) \geq d$, otherwise $\langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle$ cannot be true. Furthermore, since $e' \geq 0$, we also have $e + e' \geq d$. So, the delay transition of q is actually $\langle\, \Delta e \;;\; (x \parallel \Delta e'), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e - d \;;\; (x \parallel \Delta e'), \sigma' \,\rangle$ and we find $q' \equiv \Delta e - d \;;\; (x \parallel \Delta e')$. According to Rules 9, 20, and 15, we find $\langle\, \Delta e \;;\; x \parallel \Delta e + e', \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e - d \;;\; x \parallel \Delta e + e' - d, \sigma' \,\rangle$. Furthermore, we can rewrite $\Delta e - d \;;\; x \parallel \Delta e + e' - d$ to $\Delta e - d \;;\; x \parallel \Delta e - d + e'$. So, take $p' \equiv \Delta e - d \;;\; x \parallel \Delta e - d + e'$ and note that $(p', q') \in R$. □

Lemma 4.48 expresses that χ_σ has time determinism. This lemma states that if a process can evolve into two or more processes by performing a single delay step, then these processes must be syntactically equivalent.

Lemma 4.48 · (Time determinism) *Let p, p', and p'' be processes; σ, σ', and σ'' be stacks; and d be a positive real number. Then*

$$(\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \wedge \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p'', \sigma'' \,\rangle) \Rightarrow (p' \equiv p'' \wedge \sigma' \equiv \sigma'').$$

Proof · (Lemma 4.48) So, we need to show that a delay step always leads to a unique result. First, we prove that this holds for the defined atomic processes. Then we prove by induction that it also holds for the defined process operators.

Let p be an atomic process. Then we end up with the following cases: $p \equiv \varepsilon$, $p \equiv \delta$, $p \equiv \mathsf{skip}$, $p \equiv x := e$, $p \equiv m\,!\,e$, $p \equiv m\,?\,x$, and $p \equiv \Delta e$. In case $p \equiv \varepsilon$, $p \equiv \delta$, $p \equiv \mathsf{skip}$, or $p \equiv x := e$, the hypothesis does not hold, which then concludes the proof. For the cases $p \equiv m\,!\,e$, $p \equiv m\,?\,x$, and $p \equiv \Delta e$ only one rule per process operator exists that defines the delay step for p, Rule 7, 8, and 9 respectively. Hence, the resulting process must be unique.

In case p is a non-atomic process, we end up with the following cases (x and y are processes): $p \equiv e :\rightarrow x$, $p \equiv x \mathbin{;} y$, $p \equiv x \between y$, $p \equiv x^*$, $p \equiv x \parallel y$, $p \equiv [\![s \mid x]\!]$, $p \equiv \partial_A(x)$, $p \equiv \pi(x)$, $p \equiv \tau_A(x)$. For the cases $p \equiv e :\rightarrow x$, $p \equiv x^*$, $p \equiv [\![s \mid x]\!]$, $p \equiv \partial_A(x)$, $p \equiv \pi(x)$, and $p \equiv \tau_A(x)$, only one rule per process operator exists that defines the delay step for p, Rule 12, 26, 34, 37, 40, and 44, respectively. Hence the resulting process must be unique.

For the cases $p \equiv x \mathbin{;} y$, $p \equiv x \between y$, and $p \equiv x \parallel y$, two or more rules define possible delay steps. We prove that also for these processes a delay step leads to a unique result by showing that these rules have mutually exclusive hypotheses or by showing that their conclusions are syntactically equivalent.

In case $p \equiv x \mathbin{;} y$, Rule 20, 21, 22, and 23 apply. The hypotheses of Rule 20 and 21 are not mutually exclusive. However, their conclusions are syntactically equivalent. Furthermore, the hypothesis of Rule 22 excludes the hypotheses of Rule 20, 21, and 23, and the hypothesis of Rule 23 excludes the hypotheses of Rule 20, 21, and 22.

In case $p \equiv x \between y$, Rule 15 and 16 apply. The hypotheses of these rules are mutually exclusive.

In case $p \equiv x \parallel y$, Rule 30 and 31 apply. The hypotheses of these rules are mutually exclusive. □

Action and delay transitions can affect the stack that is part of the transition. Lemma 4.49 below describes how they are effected. As can be seen, τ and delay transitions leave the stack untouched. The other possible transitions either also leave the stack untouched (p can be a state process so that the transition effects the state of that process instead of the stack), or their effect on the stack is the result of a substitution. Further, Lemma 4.50 below strengthens Lemma 4.49. It was used to prove the properties of the state operator in Section 4.11.

Lemma 4.49 · *Let p and p' be processes; σ and σ' be stacks such that $m \in dom(\sigma)$, then*

$$\begin{aligned}
\langle p, \sigma \rangle &\xrightarrow{\tau} \langle p', \sigma' \rangle &&\Rightarrow \quad \sigma' = \sigma,\\
\langle p, \sigma \rangle &\xrightarrow{aa(x,c)} \langle p', \sigma' \rangle &&\Rightarrow \quad \sigma' = \sigma \vee \sigma' = \sigma[c/x],\\
\langle p, \sigma \rangle &\xrightarrow{sa(m,c)} \langle p', \sigma' \rangle &&\Rightarrow \quad \sigma' = \sigma \vee \sigma' = \sigma[c/m],\\
\langle p, \sigma \rangle &\xrightarrow{ra(m,x)} \langle p', \sigma' \rangle &&\Rightarrow \quad \sigma' = \sigma \vee \sigma' = \sigma[\sigma(m)/x],\\
\langle p, \sigma \rangle &\xrightarrow{ca(m,x,c)} \langle p', \sigma' \rangle &&\Rightarrow \quad \sigma' = \sigma \vee \sigma' = \sigma[c/m][c/x],\\
\langle p, \sigma \rangle &\xmapsto{d} \langle p', \sigma' \rangle &&\Rightarrow \quad \sigma' = \sigma.
\end{aligned}$$

Recall that due to the state operator, it can be the case that even though an action like $aa(x, c)$ is performed, we have $\sigma = \sigma'$.

Proof · (Lemma 4.49) We will prove this lemma by structural induction on p.

Basis Let p be an atomic process. The deduction rules for atomic processes are given in Table 4.1. From these deduction rules it follows immediately that the stack σ changes as stated in Lemma 4.49.

Inductive step Let p be a compound process with a unary process operator $\otimes$ and process argument p_0 or with a binary process operator $\otimes$ and process argument p_0 and p_1. Furthermore, the induction hypothesis says that the lemma holds for the process arguments p_0 (and p_1). The deduction rules for compound processes are given in Tables 4.2–4.10. In most deduction rules, σ' is defined by a transition of a process argument of p. Therefore, by using the induction hypothesis these cases are easily proved. The only interesting rule is Rule 29 according to which communication actions can be derived. So, if this rule applies, we have $p \equiv p_0 \parallel p_1$ and $\langle p_0 \parallel p_1, \sigma \rangle \xrightarrow{ca(m,x,c)} \langle p_0' \parallel p_1', \sigma'' \rangle$ such that $\langle p_0, \sigma \rangle \xrightarrow{sa(m,c)} \langle p_0', \sigma' \rangle$ and $\langle p_1, \sigma' \rangle \xrightarrow{ra(m,x)} \langle p_1', \sigma'' \rangle$ (the case that p_0 receives and p_1 sends is similar). Applying the induction hypothesis to p_0 and p_1, we find $\sigma' = \sigma[c/m]$ and $\sigma'' = \sigma'[c/x]$. Therefore, we have $\sigma'' = \sigma[c/m][c/x]$. □

Lemma 4.50 · *Let p and p' be processes, σ_0, σ_0', σ_1, and σ_1' be stacks, a be an action, and d be a positive real number, then*

$$\begin{gathered}(\langle p, \sigma_0 \rangle \xrightarrow{a} \langle p', \sigma_0' \rangle \wedge \langle p, \sigma_1 \rangle \xrightarrow{a} \langle p', \sigma_1' \rangle) \vee \\ (\langle p, \sigma_0 \rangle \overset{d}{\longmapsto} \langle p', \sigma_0' \rangle \wedge \langle p, \sigma_1 \rangle \overset{d}{\longmapsto} \langle p', \sigma_1' \rangle) \\ \Rightarrow \\ (\sigma_0' = \sigma_0 \wedge \sigma_1' = \sigma_1) \vee (\exists c, i : \sigma_0' = \sigma_0[c/i] \wedge \sigma_1' = \sigma_1[c/i]) \vee \\ (\exists c, c', i, i' : \sigma_0' = \sigma_0[c/i][c'/i'] \wedge \sigma_1' = \sigma_1[c/i][c'/i']).\end{gathered}$$

Proof · (Lemma 4.50) We distinguish two cases: action transitions and delay transitions.

Action transition: We have $\langle p, \sigma_0 \rangle \xrightarrow{a} \langle p', \sigma_0' \rangle$ and $\langle p, \sigma_1 \rangle \xrightarrow{a} \langle p', \sigma_1' \rangle$. We will only consider the case where a is an assignment action, since the other possibilities (skip action, send action, receive action, and communication action)

are similar. So, $a \equiv aa(x, c)$ for some programming variable x and value c. We distinguish the following cases.

$\sigma_0 = \sigma_0'$ *and* $\sigma_1 = \sigma_1'$: Trivial.

$\sigma_0 = \sigma_0'$ *and* $\sigma_1 \neq \sigma_1'$: According to Lemma 4.49, we have $\sigma_1' = \sigma_1[c/x]$. There are two possibilities for σ_0: $x \in dom(\sigma_0)$ and $x \notin dom(\sigma_0)$.

- $x \in dom(\sigma_0)$: Now, we either have $p \equiv [\![s \mid p_0]\!]$ for some state s with $x \in dom(s)$ and process p_0 or we have $\sigma_0(x) = c$. In the first case, we find $\langle p_0, s{::}\sigma_0 \rangle \xrightarrow{aa(x,c)} \langle p_0', s'{::}\sigma_0' \rangle$ for some state s' and process p_0', such that $p' \equiv [\![s' \mid p_0']\!]$. But then we also find $\langle p_0, s{::}\sigma_1 \rangle \xrightarrow{aa(x,c)} \langle p_0', s'{::}\sigma_1' \rangle$ and since $x \in dom(s)$ we have $\sigma_1 = \sigma_1'$ which is a contradiction. So, we must have $\sigma_0(x) = c$. Now, we have $\sigma_0' = \sigma_0[c/x] = \sigma_0$ and $\sigma_1' = \sigma_1[c/x]$.
- $x \notin dom(\sigma_0)$: Now, according to Lemma A.26, we have $\sigma_0' = \sigma_0[c/x] = \sigma_0$ and $\sigma_1' = \sigma_1[c/x]$.

$\sigma_0 \neq \sigma_0'$ *and* $\sigma_1 = \sigma_1'$: The proof is similar to the proof of the previous case.

$\sigma_0 \neq \sigma_0'$ *and* $\sigma_1 \neq \sigma_1'$: According to Lemma 4.49, we have $\sigma_0' = \sigma_0[c/x]$ and $\sigma_1' = \sigma_1[c/x]$.

Delay transition: We have $\langle p, \sigma_0 \rangle \overset{d}{\longmapsto} \langle p', \sigma_0' \rangle$ and $\langle p, \sigma_1 \rangle \overset{d}{\longmapsto} \langle p', \sigma_1' \rangle$. According to Lemma 4.49, we have $\sigma_0' = \sigma_0$ and $\sigma_1' = \sigma_1$. □

Transitions from one process to another do not change the domain of the stack. This is expressed by Lemma 4.51 below.

Lemma 4.51 · *Let p and p' be processes, σ and σ' be stacks, a an action and d a positive real number, then*

$$\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle \Rightarrow dom(\sigma) = dom(\sigma'),$$
$$\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle \Rightarrow dom(\sigma) = dom(\sigma').$$

Proof · (Lemma 4.51) Lemma 4.49 showed that for a transition $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$ or $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$ we have $\sigma' = \sigma$, $\sigma' = \sigma[c/i]$, or $\sigma' = \sigma[c/i][c'/i']$. Furthermore, Lemma A.25 gives us $\forall c, i : dom(\sigma[c/i]) = dom(\sigma)$. □

If a process can terminate, perform an action, or delay under a certain stack, then it can also terminate, perform that action, or delay for the same amount of time under another stack that is observationally equivalent to that stack. This is expressed by Lemma 4.52 below.

Lemma 4.52 · *Let p and p' be processes and let σ and σ' be stacks such that $\sigma \stackrel{\circ}{=} \sigma'$. Then we have*

$$\langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, \sigma' \,\rangle\!\downarrow.$$

Furthermore, let a be an action and d a positive real number, then

$$\exists\sigma : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma \,\rangle \;\Leftrightarrow\; \exists\sigma' : \langle\, p, \sigma' \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle,$$
$$\exists\sigma : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma \,\rangle \;\Leftrightarrow\; \exists\sigma' : \langle\, p, \sigma' \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle.$$

Proof · (Lemma 4.52) The proof of this lemma is based on two observations. The first observation is that for any expression e, and channel m we have $\sigma \stackrel{\circ}{=} \sigma' \Rightarrow (\sigma(e) = \sigma'(e) \wedge \sigma(m) = \sigma'(m))$, according to Definition A.13.

The second observation is that if a hypothesis of a deduction rule uses a stack σ, it is to evaluate an expression, $\sigma(e)$ or a channel, $\sigma(m)$. So, if a deduction rule applies to $\langle\, p, \sigma \,\rangle$, it also applies to $\langle\, p, \sigma' \,\rangle$ and vice versa. □

We will now present four lemmas: *Time confluence*, *Preservation of terminations*, *Preservation of action transitions*, and *Undelayability of terminations*. The proofs of these lemmas are merged into one.

Lemma 4.53 · (Time confluence) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle \wedge \langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, then $\langle\, q, \sigma' \,\rangle \overset{d-d'}{\longmapsto} \langle\, r, \sigma'' \,\rangle$.*

This lemma says that dividing delay transitions in smaller delay transitions, which is possible according to Lemma 4.46, does not influence the final process.

Lemma 4.54 · (Preservation of terminations) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle \wedge \langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, then $\langle\, p_{d'}, \sigma_{d'} \,\rangle\!\downarrow \Rightarrow \langle\, p, \sigma \,\rangle\!\downarrow \wedge \langle\, p_d, \sigma_d \,\rangle\!\downarrow$.*

This lemma says that if a process can terminate after a small delay, then this termination option was possible before the delay an it is still possible after the big delay. Therefore, termination behaviour is preserved while delaying. Note that the inverse is not true. That is, a termination option after a small delay cannot be derived from a termination option before the small delay or from a termination option after the big delay.

Lemma 4.55 · (Preservation of action transitions) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle \wedge \langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, then $\forall p_{d',a}, \sigma_{d',a}, a : \langle p_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists p_a, \sigma_a, p_{d,a}, \sigma_{d,a} : \langle p, \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_a \rangle \wedge \langle p_d, \sigma_d \rangle \xrightarrow{a} \langle p_{d,a}, \sigma_{d,a} \rangle$.*

Note that due to Lemma 4.46 we have that if $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle$, we also have $\langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$ for some $p_{d'}$, $\sigma_{d'}$, and d' with $d' < d$.

This lemma says for action transitions what the previous lemma said for terminations. So, action transitions are preserved while delaying. Again, the other way around is not true.

Lemma 4.56 · (Undelayability of terminations) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle \wedge \langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, then $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$.*

This lemma says that while delaying, a process can never terminate. This is a stronger result than Lemma 4.54 (Preservation of terminations). In fact, in the proof below, we will use the undelayability of terminations to prove preservation of terminations.

Proof · (Lemmas 4.53, 4.54, 4.55, and 4.56) As mentioned above, Lemmas 4.53, 4.54, 4.55, and 4.56 are proved together. We also mentioned that Lemma 4.54 follows from Lemma 4.56. Therefore, we will focus on Lemmas 4.53, 4.55, and 4.56. These lemmas are proved by structural induction on process p.

First, we prove that Lemmas 4.53, 4.55, and 4.56 hold for the atomic processes. Then we prove by induction that they also hold for every process operator.

Let p be an atomic process. Then we end up with the following cases: $p \equiv \varepsilon$, $p \equiv \delta$, $p \equiv \mathsf{skip}$, $p \equiv x := e$, $p \equiv m\,!\,e$, $p \equiv m\,?\,x$, and $p \equiv \Delta e$. For the processes $p \equiv \varepsilon$, $p \equiv \delta$, $p \equiv \mathsf{skip}$, and $p \equiv x := e$, the hypothesis does not hold, which then concludes the proof for these processes. The cases that still need proof thus are $p \equiv m\,!\,e$, $p \equiv m\,?\,x$, and $p \equiv \Delta e$.

$p \equiv m\,!\,e$: Using Rule 7 we obtain $\langle m\,!\,e, \sigma \rangle \overset{d}{\longmapsto} \langle m\,!\,e, \sigma_d \rangle$ and $\langle m\,!\,e, \sigma \rangle \overset{d'}{\longmapsto} \langle m\,!\,e, \sigma_{d'} \rangle$ with $d' < d$ and $\sigma \equiv \sigma_d \equiv \sigma_{d'}$.

The following items prove Lemmas 4.53, 4.55, and 4.56, respectively.

1. We apply Rule 7 and obtain $\langle m\,!\,e, \sigma_{d'} \rangle \overset{d-d'}{\longmapsto} \langle m\,!\,e, \sigma_d \rangle$.

2. Suppose $\langle\, m\,!\,e, \sigma_{d'}\,\rangle \xrightarrow{sa(m,c)} \langle\, \varepsilon, \sigma_{d', sa(m,c)}\,\rangle$ for $\sigma(e) = c$. Then we also have $\langle\, m!e, \sigma\,\rangle \xrightarrow{sa(m,c)} \langle\, \varepsilon, \sigma_{sa(m,c)}\,\rangle$ and $\langle\, m!e, \sigma_d\,\rangle \xrightarrow{sa(m,c)} \langle\, \varepsilon, \sigma_{d,sa(m,c)}\,\rangle$ and $\sigma \equiv \sigma_{d', sa(m,c)} \equiv \sigma_{d,sa(m,c)}$.
3. Since termination is not defined for send processes, we obtain $\langle\, m!e, \sigma_{d'}\,\rangle \not\downarrow$.

$p \equiv m\,?\,x$: Like the proof of case $p \equiv m\,!\,e$ using Rule 8 instead of Rule 7.

$p \equiv \Delta e$: We can use Rule 9 and obtain $\langle\, \Delta e, \sigma\,\rangle \overset{d}{\longmapsto} \langle\, \Delta e - d, \sigma\,\rangle$ and $\langle\, \Delta e, \sigma\,\rangle \overset{d'}{\longmapsto} \langle\, \Delta e - d', \sigma\,\rangle$, with $d' < d$.

The following items prove Lemmas 4.53, 4.55, and 4.56, respectively.

1. We can use Rule 9 and obtain $\langle\, \Delta e - d', \sigma\,\rangle \overset{d-d'}{\longmapsto} \langle\, \Delta e - d, \sigma\,\rangle$.
2. The hypothesis does not hold since $\Delta e - d'$ cannot perform an action. So, we are done.
3. From Item 1, we know that $\langle\, \Delta e - d', \sigma\,\rangle \overset{d-d'}{\longmapsto} \langle\, \Delta e - d, \sigma\,\rangle$. According to Rule 9 this gives $0 < d - d' \leq \sigma(e - d')$ and since we already knew that $d' < d$, we can derive that $\sigma(e - d') > 0$. Therefore, Rule 2 does not apply and we have $\langle\, \Delta e - d', \sigma\,\rangle \not\downarrow$.

We proved that Lemmas 4.53, 4.55, and 4.56 hold for all atomic processes. Next, we prove that they also hold for all process operators.

Guarded processes: Suppose $\langle\, p, \sigma\,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d\,\rangle$ and $\langle\, p, \sigma\,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'}\,\rangle$, where $p \equiv e :\!\rightarrow x$ and $d' < d$. Applying Rule 12 gives $\langle\, x, \sigma\,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d\,\rangle$, $\langle\, x, \sigma\,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'}\,\rangle$, and $\sigma(e) = \mathit{true}$, where $d' < d$. Induction on x we now gives us

1. $\langle\, p_{d'}, \sigma_{d'}\,\rangle \overset{d-d'}{\longmapsto} \langle\, p_d, \sigma_d\,\rangle$,
2. $\forall p_{d',a}, \sigma_{d',a}, a : \langle\, p_{d'}, \sigma_{d'}\,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a}\,\rangle \Rightarrow \exists x_a, \sigma_a, p_{d,a}, \sigma_{d,a} : \langle\, x, \sigma\,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a\,\rangle \wedge \langle\, p_d, \sigma_d\,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a}\,\rangle$,
3. $\langle\, p_{d'}, \sigma_{d'}\,\rangle \not\downarrow$.

This makes the proof of Lemma 4.53 trivial.

To prove Lemma 4.55, assume $\langle\, p_{d'}, \sigma_{d'}\,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a}\,\rangle$. Then, we know that $\exists x_a, \sigma_a, p_{d,a}, \sigma_{d,a} : \langle\, x, \sigma\,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a\,\rangle \wedge \langle\, p_d, \sigma_d\,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a}\,\rangle$. Since $\sigma(e) = \mathit{true}$, Rule 11 applies and we obtain $\langle\, p, \sigma\,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a\,\rangle \wedge \langle\, p_d, \sigma_d\,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a}\,\rangle$.

The proof of 4.56 is trivial.

Alternative composition: Suppose $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle$ and $\langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, where $p \equiv x \parallel y$ and $d' < d$. Then at least one of the Rules 15 and 16 applies. Looking at the hypotheses of these rules we can distinguish three cases regarding the delay behaviour of x and y:

1. $\langle x, \sigma \rangle \not\longmapsto \wedge \langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y_d, \sigma_d \rangle$,
2. $\langle x, \sigma \rangle \overset{d}{\longmapsto} \langle x_d, \sigma_d \rangle \wedge \langle y, \sigma \rangle \not\longmapsto$,
3. $\langle x, \sigma \rangle \overset{d}{\longmapsto} \langle x_d, \sigma_d \rangle \wedge \langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y_d, \sigma_d \rangle$.

For case 1, Rule 15 applies, so we can conclude that $p_d \equiv y_d$. From Lemma 4.46 it follows that also $\langle y, \sigma \rangle \overset{d'}{\longmapsto} \langle y_{d'}, \sigma_{d'} \rangle$. Again, Rule 15 applies and we can conclude that $p_{d'} \equiv y_{d'}$.

Induction on y gives us

1. $\langle y_{d'}, \sigma_{d'} \rangle \overset{d-d'}{\longmapsto} \langle y_d, \sigma_d \rangle$,
2. $\forall y_{d',a}, \sigma_{d',a}, a : \langle y_{d'}, \sigma_{d'} \rangle \overset{a}{\longrightarrow} \langle y_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle y, \sigma \rangle \overset{a}{\longrightarrow} \langle y_a, \sigma_a \rangle \wedge \langle y_d, \sigma_d \rangle \overset{a}{\longrightarrow} \langle y_{d,a}, \sigma_{d,a} \rangle$,
3. $\langle y_{d'}, \sigma_{d'} \rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv y_d$ and $p_{d'} \equiv y_{d'}$. Result 1 of the induction on y then gives us $\langle p_{d'}, \sigma_{d'} \rangle \overset{d-d'}{\longmapsto} \langle p_d, \sigma_d \rangle$.

To prove Lemma 4.55, we assume $\langle p_{d'}, \sigma_{d'} \rangle \overset{a}{\longrightarrow} \langle p_{d',a}, \sigma_{d',a} \rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv y_{d'}$, we find $p_{d',a} \equiv y_{d',a}$. Also, since $p_d \equiv y_d$ result 2 of the induction on y gives us $\exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle y, \sigma \rangle \overset{a}{\longrightarrow} \langle y_a, \sigma_a \rangle \wedge \langle p_d, \sigma_d \rangle \overset{a}{\longrightarrow} \langle y_{d,a}, \sigma_{d,a} \rangle$. Furthermore, Rule 14 gives us $\langle p, \sigma \rangle \overset{a}{\longrightarrow} \langle y_a, \sigma_a \rangle$. So, $\langle p, \sigma \rangle \overset{a}{\longrightarrow} \langle p_a, \sigma_a \rangle$ with $p_a \equiv y_a$ and $\langle p_d, \sigma_d \rangle \overset{a}{\longrightarrow} \langle p_{d,a}, \sigma_{d,a} \rangle$ with $p_{d,a} \equiv y_{d,a}$.

Lemma 4.56 is easily proved since we already derived that $p_{d'} \equiv y_{d'}$. Result 3 of the induction on y then gives $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$.

Case 2 is proven in a way similar to case 1 with x and y interchanged.

Finally, for case 3, Rule 16 applies and we obtain $\langle x \parallel y, \sigma \rangle \overset{d}{\longmapsto} \langle x_d \parallel y_d, \sigma_d \rangle$. Since $p \equiv x \parallel y$ and $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle$, we obtain $p_d \equiv x_d \parallel y_d, \sigma_d$. Also, since in this case we have $\langle x, \sigma \rangle \overset{d}{\longmapsto} \langle x_d, \sigma_d \rangle$ and $\langle y, \sigma \rangle \overset{d}{\longmapsto} \langle y_d, \sigma_d \rangle$, Lemma 4.46 gives us $\langle x, \sigma \rangle \overset{d'}{\longmapsto} \langle x_{d'}, \sigma_{d'} \rangle$ and $\langle y, \sigma \rangle \overset{d'}{\longmapsto} \langle y_{d'}, \sigma_{d'} \rangle$, and we obtain $\langle x \parallel y, \sigma \rangle \overset{d'}{\longmapsto} \langle x_{d'} \parallel y_{d'}, \sigma_{d'} \rangle$. Since $p \equiv x \parallel y$ and $\langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, we obtain $p_{d'} \equiv x_{d'} \parallel y_{d'}$.

Induction on both x and y gives us

1. $\langle x_{d'}, \sigma_{d'} \rangle \overset{d-d'}{\longmapsto} \langle x_d, \sigma_d \rangle$, and $\langle y_{d'}, \sigma_{d'} \rangle \overset{d-d'}{\longmapsto} \langle y_d, \sigma_d \rangle$,

2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle x_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle x_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma_a \rangle \wedge \langle x_d, \sigma_d \rangle \xrightarrow{a} \langle x_{d,a}, \sigma_{d,a} \rangle$,
   $\forall y_{d',a}, \sigma d', a, a : \langle y_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle y_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle y, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma_a \rangle \wedge \langle y_d, \sigma_d \rangle \xrightarrow{a} \langle y_{d,a}, \sigma_{d,a} \rangle$,
3. $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$, and $\langle y_{d'}, \sigma_{d'} \rangle \not\downarrow$.

To prove Lemma 4.53 we have to prove that $\langle p_{d'}, \sigma_{d'} \rangle \xmapsto{d-d'} \langle p_d, \sigma_d \rangle$. Since $p_{d'} \equiv x_{d'} \parallel y_{d'}$, we can use result 1 of the induction on x and y and Rule 16 to obtain $\langle p_{d'}, \sigma_{d'} \rangle \xmapsto{d-d'} \langle x_d \parallel y_d, \sigma_d \rangle$. Since $p_d \equiv x_d \parallel y_d$, we are done.

To prove Lemma 4.55 we assume $\langle p_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv x_{d'} \parallel y_{d'}$, Rule 14 should apply. Therefore, we have $\langle x_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle$ or $\langle y_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle$. Result 2 of the induction on x and y then gives us $\langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma_a \rangle$ and $\langle x_d, \sigma_d \rangle \xrightarrow{a} \langle x_{d,a}, \sigma_{d,a} \rangle$ or we obtain $\langle y, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma_a \rangle$ and $\langle y_d, \sigma_d \rangle \xrightarrow{a} \langle y_{d,a}, \sigma_{d,a} \rangle$. So, in the first case we take $p_a \equiv x_a$ and $p_{d,a} \equiv x_{d,a}$, and in the second case we take $p_a \equiv y_a$ and $p_{d,a} \equiv y_{d,a}$.

To prove Lemma 4.56 we use result 3 of the induction on x and y which tells us that $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$ and $\langle y_{d'}, \sigma_{d'} \rangle \not\downarrow$. Since $p_{d'} \equiv x_{d'} \parallel y_{d'}$, Rule 13 does not apply to $p_{d'}$ and we obtain $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$.

Sequential composition: Suppose that $\langle p, \sigma \rangle \xmapsto{d} \langle p_d, \sigma_d \rangle$ and $\langle p, \sigma \rangle \xmapsto{d'} \langle p_{d'}, \sigma_{d'} \rangle$, where $p \equiv x \ ; \ y$ and $d' < d$. Then at least one of the Rules 20, 21, 22, and 23 applies. Looking at the hypotheses of these rules we can distinguish three cases regarding the delay behaviour of x and y:

1. $\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma_d \rangle \wedge (\langle x, \sigma \rangle \not\downarrow \vee \langle y, \sigma \rangle \not\mapsto)$,
2. $\langle x, \sigma \rangle \not\mapsto \wedge \langle x, \sigma \rangle \downarrow \wedge \langle y, \sigma \rangle \xmapsto{d} \langle y_d, \sigma_d \rangle$,
3. $\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma_d \rangle \wedge \langle x, \sigma \rangle \downarrow \wedge \langle y, \sigma \rangle \xmapsto{d} \langle y_d, \sigma_d \rangle$.

For case 1, Rule 20 or 21 applies and we obtain $\langle x \ ; \ y, \sigma \rangle \xmapsto{d} \langle x_d \ ; \ y, \sigma_d \rangle$. Since $p \equiv x \ ; \ y$ and $\langle p, \sigma \rangle \xmapsto{d} \langle p_d, \sigma_d \rangle$, we obtain $p_d \equiv x_d \ ; \ y$. Also, since in this case we have $\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma_d \rangle$, Lemma 4.46 gives us $\langle x, \sigma \rangle \xmapsto{d'} \langle x_{d'}, \sigma_{d'} \rangle$, and we obtain $\langle x \ ; \ y, \sigma \rangle \xmapsto{d'} \langle x_{d'} \ ; \ y, \sigma_{d'} \rangle$. Since $p \equiv x \ ; \ y$ and $\langle p, \sigma \rangle \xmapsto{d'} \langle p_{d'}, \sigma_{d'} \rangle$, we obtain $p_{d'} \equiv x_{d'} \ ; \ y$.

Induction on x gives us

1. $\langle x_{d'}, \sigma_{d'} \rangle \xmapsto{d-d'} \langle x_d, \sigma_d \rangle$,
2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle x_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle x_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma_a \rangle \wedge \langle x_d, \sigma_d \rangle \xrightarrow{a} \langle x_{d,a}, \sigma_{d,a} \rangle$,
3. $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$.

Lemma 4.53 is proved using results 1 and 3 of the induction on x. Together with Rule 20 they give $\langle\, x_{d'} \;;\; y, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, x_d \;;\; y, \sigma_d \,\rangle$. Since $p_{d'} \equiv x_{d'} \;;\; y$ and $p_d \equiv x_d \;;\; y$, we have $\langle\, p_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$.

To prove Lemma 4.55, assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv x_{d'} \;;\; y$ either Rule 18 or 19 applies. However, result 3 of the induction on x gives us $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\downarrow$, so Rule 19 does not apply. As a consequence, Rule 18 applies and we obtain $\langle\, x_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, x_{d',a}, \sigma_{d',a} \,\rangle$ for some $x_{d',a}$, and $\sigma_{d',a}$ such that $p_{d',a} \equiv x_{d',a} \;;\; y$. Now, by result 2 of the induction on x we know that there are x_a, σ_a, $x_{d,a}$, and $\sigma_{d,a}$ such that $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle$ and $\langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$. Using Rule 18 we then obtain $\langle\, x \;;\; y, \sigma \,\rangle \xrightarrow{a} \langle\, x_a \;;\; y, \sigma_a \,\rangle$ and $\langle\, x_d \;;\; y, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a} \;;\; y, \sigma_{d,a} \,\rangle$. So, we have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma_a \,\rangle$ and $\langle\, p_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a} \,\rangle$, with $p_a \equiv x_a \;;\; y$ and $p_{d,a} \equiv x_{d,a} \;;\; y$.

To prove Lemma 4.56, we use result 3 of the induction on x, which tells us that $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\downarrow$. Since $p_{d'} \equiv x_{d'} \;;\; y$, Rule 17 does not apply to $p_{d'}$ and we obtain $\langle\, p_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

For case 2, Rule 22 applies, so we can conclude that $p_d \equiv y_d$. From Lemma 4.46 it follows that $\langle\, y, \sigma_p \,\rangle \overset{d'}{\longmapsto} \langle\, y_{d'}, \sigma_{d'} \,\rangle$. Again, Rule 22 applies and we can conclude that $p_{d'} \equiv y_{d'}$.

Induction on y gives us

1. $\langle\, y_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, y_d, \sigma_d \,\rangle$,
2. $\forall y_{d',a}, \sigma_{d'a,}, a : \langle\, y_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, y_{d',a}, \sigma_{d',a} \,\rangle \Rightarrow \exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, y_a, \sigma_a \,\rangle \wedge \langle\, y_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, y_{d,a}, \sigma_{d,a} \,\rangle$,
3. $\langle\, y_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv y_d$ and $p_{d'} \equiv y_{d'}$. Result 1 of the induction on y then gives us $\langle\, p_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$.

To prove Lemma 4.55, assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv y_{d'}$, we find $p_{d',a} \equiv y_{d',a}$. Also, since $p_d \equiv y_d$ result 2 of the induction on y gives us $\exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, y_a, \sigma_a \,\rangle \wedge \langle\, p_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, y_{d,a}, \sigma_{d,a} \,\rangle$. Furthermore, since we know that $\langle\, x, \sigma \,\rangle\!\downarrow$, Rule 19 gives us $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, y_a, \sigma_a \,\rangle$. So, $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma_a \,\rangle$ with $p_a \equiv y_a$ and $\langle\, p_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a} \,\rangle$ with $p_{d,a} \equiv y_{d,a}$.

Lemma 4.56 is easily proved since we already derived that $p_{d'} \equiv y_{d'}$. Result 3 of the induction on y then gives $\langle\, p_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Finally, for case 3, Rule 23 applies and we obtain $\langle\, x \;;\; y, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x_d \;;\; y \,[\!]\, y_d, \sigma_d \,\rangle$. Since $p \equiv x \;;\; y$ and $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$, we obtain $p_d \equiv x_d \;;\; y \,[\!]\, y_d$. Also, since

in this case we have $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$ and $\langle\, y, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, y_d, \sigma_d \,\rangle$, Lemma 4.46 gives us $\langle\, x, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, x_{d'}, \sigma_{d'} \,\rangle$ and $\langle\, y, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, y_{d'}, \sigma_{d'} \,\rangle$, and we obtain $\langle\, x \;;\; y, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, x_{d'} \;;\; y \,[\!]\, y_{d'}, \sigma_{d'} \,\rangle$. Since $p \equiv x \;;\; y$ and $\langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, we obtain $p_{d'} \equiv x_{d'} \;;\; y \,[\!]\, y_{d'}$.

Induction on both x and y gives us

1. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$, and $\langle\, y_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, y_d, \sigma_d \,\rangle$,
2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle\, x_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, x_{d',a}, \sigma_{d',a} \,\rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle \wedge \langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$, and
   $\forall y_{d',a}, \sigma_{d',a}, a : \langle\, y_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, y_{d',a}, \sigma_{d',a} \,\rangle \Rightarrow \exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, y_a, \sigma_a \,\rangle \wedge \langle\, y_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, y_{d,a}, \sigma_{d,a} \,\rangle$,
3. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\longmapsto$, and $\langle\, y_{d'}, \sigma_{d'} \,\rangle \not\longmapsto$.

To prove Lemma 4.53, we have $p_{d'} \equiv x_{d'} \;;\; y \,[\!]\, y_{d'}$ and result 4 of the induction on x and y gives us $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\longmapsto$. Consequently, Rule 20 and 16 apply and we obtain $\langle\, p_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, x_d \;;\; y \,[\!]\, y_d, \sigma_d \,\rangle$ using result 1 of the induction on x and y. Since $p_d \equiv x_d \;;\; y \,[\!]\, y_d$, we are done.

To prove Lemma 4.55, assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv x_{d'} \;;\; y \,[\!]\, y_{d'}$, Rule 14 should apply. Therefore, we have $\langle\, x_{d'} \;; y, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ or $\langle\, y_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$. Since result 3 of the induction on x and y gives us $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\longmapsto$, Rule 19 does not apply to $x_{d'} \;; y$ and therefore, we must have $\langle\, x_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$. Result 2 of the induction on x and y then gives us $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle$ and $\langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$ or we obtain $\langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, y_a, \sigma_a \,\rangle$ and $\langle\, y_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, y_{d,a}, \sigma_{d,a} \,\rangle$. So, in the first case we take $p_a \equiv x_a$ and $p_{d,a} \equiv x_{d,a}$, and in the second case we take $p_a \equiv y_a$ and $p_{d,a} \equiv y_{d,a}$.

To prove Lemma 4.56, we use result 3 of the induction on x and y which tells us that $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\longmapsto$ and $\langle\, y_{d'}, \sigma_{d'} \,\rangle \not\longmapsto$. Since $p_{d'} \equiv x_{d'} \;;\; y \,[\!]\, y_{d'}$, Rule 17 does not apply to $x_{d'} \;; y$ and we obtain $\langle\, x_{d'} \;; y, \sigma_{d'} \,\rangle \not\longmapsto$. Consequently, Rule 13 does not apply to $p_{d'}$ we obtain $\langle\, p_{d'}, \sigma_{d'} \,\rangle \not\longmapsto$.

Repetition: Suppose $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$ and $\langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, where $p \equiv x^*$ and $d' < d$. Using Rule 26 we obtain $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$ and we can conclude that $p_d \equiv x_d \;;\; x^*$. Also, $\langle\, x, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, x_{d'}, \sigma_{d'} \,\rangle$ and we can conclude that $p_{d'} \equiv x_{d'} \;;\; x^*$.

Induction on x gives us

1. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$,

2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle\, x_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, x_{d',a}, \sigma_{d',a} \,\rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle \wedge \langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$,
3. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv x_d \;;\; x^*$ and $p_{d'} \equiv x_{d'} \;;\; x^*$. Result 1 and 3 of the induction on x and Rule 20 then give $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xmapsto{d-d'} \langle\, p_d, \sigma_d \,\rangle$.

To prove Lemma 4.55, we assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv x_{d'} \;;\; x^*$ we find by result 2 and 3 of the induction on x that $p_{d',a} \equiv x_{d',a} \;;\; x^*$. Now, by result 2 of the induction on x we know that there are x_a and $x_{d,a}$, such that $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle$ and $\langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$. Using Rule 25 and 18 respectively, we obtain $\langle\, x^*, \sigma \,\rangle \xrightarrow{a} \langle\, x_a \;;\; x^*, \sigma_a \,\rangle$ and $\langle\, x_d \;;\; x^*, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a} \;;\; x^*, \sigma_{d,a} \,\rangle$. So, we have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma_a \,\rangle$ and $\langle\, p_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a} \,\rangle$ with $p_a \equiv x_a \;;\; x^*$ and $p_{d,a} \equiv x_{d,a} \;;\; x^*$.

To prove Lemma 4.56 we use result 3 of the induction on x, which tells us that $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\downarrow$. Since $p_{d'} \equiv x_{d'} \;;\; x^*$, Rule 24 does not apply to $p_{d'}$ and we obtain $\langle\, p_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Parallel composition: Suppose $\langle\, p, \sigma \,\rangle \xmapsto{d} \langle\, p_d, \sigma_d \,\rangle$ and $\langle\, p, \sigma \,\rangle \xmapsto{d'} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, where $p \equiv x \parallel y$ and $d' < d$. Then at least one of the Rules 30 and 31 applies. Looking at the hypotheses of these rules we can distinguish three cases regarding the delay behaviour of x and y:

1. $\langle\, x, \sigma \,\rangle \not\mapsto \wedge \langle\, x, \sigma \,\rangle \downarrow \wedge \langle\, y, \sigma \,\rangle \xmapsto{d} \langle\, y_d, \sigma_d \,\rangle$,
2. $\langle\, x, \sigma \,\rangle \xmapsto{d} \langle\, x_d, \sigma_d \,\rangle \wedge \langle\, y, \sigma \,\rangle \not\mapsto \wedge \langle\, y, \sigma \,\rangle \downarrow$,
3. $\langle\, x, \sigma \,\rangle \xmapsto{d} \langle\, x_d, \sigma_d \,\rangle \wedge \langle\, y, \sigma \,\rangle \xmapsto{d} \langle\, y_d, \sigma_d \,\rangle$.

For case 1, Rule 30 applies, so we can conclude that $p_d \equiv y_d$. From Lemma 4.46 it follows that also $\langle\, y, \sigma \,\rangle \xmapsto{d'} \langle\, y_{d'}, \sigma_{d'} \,\rangle$. Again, Rule 30 applies and we can conclude that $p_{d'} \equiv y_{d'}$.

Induction on y gives us

1. $\langle\, y_{d'}, \sigma_{d'} \,\rangle \xmapsto{d-d'} \langle\, y_d, \sigma_d \,\rangle$,
2. $\forall y_{d',a}, \sigma_{d',a}, a : \langle\, y_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, y_{d',a}, \sigma_{d',a} \,\rangle \Rightarrow \exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle\, y, \sigma \,\rangle \xrightarrow{a} \langle\, y_a, \sigma_a \,\rangle \wedge \langle\, y_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, y_{d,a}, \sigma_{d,a} \,\rangle$,
3. $\langle\, y_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv y_d$ and $p_{d'} \equiv y_{d'}$. Result 1 of the induction on y then gives us $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xmapsto{d-d'} \langle\, p_d, \sigma_d \,\rangle$.

To prove Lemma 4.55, we assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv y_{d'}$, we find $p_{d',a} \equiv y_{d',a}$. Also, since $p_d \equiv y_d$

result 2 of the induction on y gives us $\exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle y, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma_a \rangle \wedge \langle p_d, \sigma_d \rangle \xrightarrow{a} \langle y_{d,a}, \sigma_{d,a} \rangle$. Furthermore, Rule 28 gives us $\langle p, \sigma \rangle \xrightarrow{a} \langle x \parallel y_a, \sigma_a \rangle$. So, $\langle p, \sigma_p \rangle \xrightarrow{a} \langle p_a, \sigma_a \rangle$ with $\langle p_a, \sigma_a \rangle \equiv \langle x \| y_a, \sigma_a \rangle$ and $\langle p_d, \sigma_d \rangle \xrightarrow{a} \langle p_{d,a}, \sigma_{d,a} \rangle$ with $p_{d,a} \equiv y_{d,a}$.

Lemma 4.56 is easily proved since we already derived that $p_{d'} \equiv y_{d'}$. Result 3 of the induction on y then gives us $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$.

Case 2 is proven in a way similar to Case 1 with x and y interchanged.

Finally, for case 3, Rule 31 applies and we obtain $\langle x \parallel y, \sigma \rangle \xmapsto{d} \langle x_d \parallel y_d, \sigma_d \rangle$. Since $p \equiv x \parallel y$ and $\langle p, \sigma \rangle \xmapsto{d} \langle p_d, \sigma_d \rangle$, we obtain $p_d \equiv x_d \parallel y_d$. Also, since in this case we have $\langle x, \sigma \rangle \xmapsto{d} \langle x_d, \sigma_d \rangle$ and $\langle y, \sigma \rangle \xmapsto{d} \langle y_d, \sigma_d \rangle$, Lemma 4.46 gives us $\langle x, \sigma \rangle \xmapsto{d'} \langle x_{d'}, \sigma_{d'} \rangle$ and $\langle y, \sigma \rangle \xmapsto{d'} \langle y_{d'}, \sigma_{d'} \rangle$, and we obtain $\langle x \parallel y, \sigma \rangle \xmapsto{d'} \langle x_{d'} \parallel y_{d'}, \sigma_{d'} \rangle$. Since $p \equiv x \parallel y$ and $\langle p, \sigma \rangle \xmapsto{d'} \langle p_{d'}, \sigma_{d'} \rangle$, we obtain $p_{d'} \equiv x_{d'} \parallel y_{d'}$.

Induction on both x and y gives us

1. $\langle x_{d'}, \sigma_{d'} \rangle \xmapsto{d-d'} \langle x_d, \sigma_d \rangle$, and $\langle y_{d'}, \sigma_{d'} \rangle \xmapsto{d-d'} \langle y_d, \sigma_d \rangle$,
2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle x_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle x_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma_a \rangle \wedge \langle x_d, \sigma_d \rangle \xrightarrow{a} \langle x_{d,a}, \sigma_{d,a} \rangle$,
   $\forall y_{d',a}, \sigma_{d',a}, a : \langle y_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle y_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists y_a, \sigma_a, y_{d,a}, \sigma_{d,a} : \langle y, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma_a \rangle \wedge \langle y_d, \sigma_d \rangle \xrightarrow{a} \langle y_{d,a}, \sigma_{d,a} \rangle$,
3. $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$, and $\langle y_{d'}, \sigma_{d'} \rangle \not\downarrow$.

To prove Lemma 4.53, we can use result 1 of the induction on x and y and Rule 31 to obtain $\langle p_{d'}, \sigma_{d'} \rangle \xmapsto{d-d'} \langle x_d \parallel y_d, \sigma_d \rangle$ since $p_{d'} \equiv x_{d'} \parallel y_{d'}$. Because $p_d \equiv x_d \parallel y_d$, this means we are done.

To prove Lemma 4.55, we assume $\langle p_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv x_{d'} \| y_{d'}$, Rule 28 or 29 applies. Since if Rule 29 applies also Rule 28 applies, it suffices to only consider the case for Rule 28. In that case we have $\langle x_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle$ or $\langle y_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle$. Result 2 of the induction on x and y then gives us $\langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma_a \rangle$ and $\langle x_d, \sigma_d \rangle \xrightarrow{a} \langle x_{d,a}, \sigma_{d,a} \rangle$ or we obtain $\langle y, \sigma \rangle \xrightarrow{a} \langle y_a, \sigma_a \rangle$ and $\langle y_d, \sigma_d \rangle \xrightarrow{a} \langle y_{d,a}, \sigma_{d,a} \rangle$. In the first case we take $p_a \equiv x_a$ and $p_{d,a} \equiv x_{d,a}$, and in the second case we take $p_a \equiv y_a$ and $p_{d,a} \equiv y_{d,a}$.

To prove Lemma 4.56 we use result 3 of the induction on x and y, which tells us that $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$ and $\langle y_{d'}, \sigma_{d'} \rangle \not\downarrow$. Since $p_{d'} \equiv x_{d'} \parallel y_{d'}$, Rule 27 does not apply to $\langle p_{d'}, \sigma_{d'} \rangle$ and we obtain $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$.

State processes: Suppose $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$ and $\langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, where $p \equiv [\![\, s \mid x \,]\!]$ and $d' < d$. Then Rule 34 applies, which gives $\langle\, x, s :: \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x_d, s_d :: \sigma_d \,\rangle$ and $\langle\, x, s :: \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, x_{d'}, s_{d'} :: \sigma_{d'} \,\rangle$. Note that as a consequence, we find $p_d \equiv [\![\, s_d \mid x_d \,]\!]$ and $p_{d'} \equiv [\![\, s_{d'} \mid x_{d'} \,]\!]$.

Induction on x gives us

1. $\langle\, x_{d'}, s_{d'} :: \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, x_d, s_d :: \sigma_d \,\rangle$,
2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle\, x_{d'}, s_{d'} :: \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, x_{d',a}, s_{d',a} :: \sigma_{d',a} \,\rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_d :$ $\langle\, x, s :: \sigma \,\rangle \xrightarrow{a} \langle\, x_a, s_a :: \sigma_a \,\rangle \wedge \langle\, x_d, s_d :: \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, s_{d,a} :: \sigma_{d,a} \,\rangle$,
3. $\langle\, x_{d'}, s_{d'} :: \sigma_{d'} \,\rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv [\![\, s_d \mid x_d \,]\!]$ and $p_{d'} \equiv [\![\, s_{d'} \mid x_{d'} \,]\!]$. Rule 34 and result 1 of the induction on x then give $\langle\, p_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$.

To prove Lemma 4.55, we assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv [\![\, s_{d'} \mid x_{d'} \,]\!]$, using Rule 33 and result 2 of the induction on x we find that $p_{d',a} \equiv [\![\, s_{d',a} \mid x_{d',a} \,]\!]$. Now, by result 2 of the induction on x we know that there are x_a and $x_{d,a}$, such that $\langle\, x, s :: \sigma \,\rangle \xrightarrow{a} \langle\, x_a, s_a :: \sigma_a \,\rangle$ and $\langle\, x_d, s_d :: \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, s_{d,a} :: \sigma_{d,a} \,\rangle$. Using Rule 33, we obtain $\langle\, [\![\, s \mid x \,]\!], \sigma \,\rangle \xrightarrow{a} \langle\, [\![\, s_a \mid x_a \,]\!], \sigma_a \,\rangle$ and $\langle\, [\![\, s_d \mid x_d \,]\!], \sigma_d \,\rangle \xrightarrow{a} \langle\, [\![\, s_{d,a} \mid x_{d,a} \,]\!], \sigma_{d,a} \,\rangle$. So, we have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma_a \,\rangle$ and $\langle\, p_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a} \,\rangle$ with $p_a \equiv [\![\, s_a \mid x_a \,]\!]$ and $p_{d,a} \equiv [\![\, s_{d,a} \mid x_{d,a} \,]\!]$.

To prove Lemma 4.56, we use result 3 of the induction on x, which tells us that $\langle\, x_{d'}, s_{d'} :: \sigma_{d'} \,\rangle \not\downarrow$. Since $p_{d'} \equiv [\![\, s_{d'} \mid x_{d'} \,]\!]$, Rule 32 does not apply to $p_{d'}$ and we obtain $\langle\, p_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Encapsulation: Suppose $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$ and $\langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, where $p \equiv \partial_A(x)$ and $d' < d$. Using Rule 37 we obtain $\langle\, x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$ and we can conclude that $p_d \equiv \partial_A(x_d)$. Also, $\langle\, x, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, x_{d'}, \sigma_{d'} \,\rangle$ and we can conclude that $p_{d'} \equiv \partial_A(x_{d'})$.

Induction on x gives us

1. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \overset{d-d'}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$,
2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle\, x_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, x_{d',a}, \sigma_{d',a} \,\rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle\, x, \sigma \,\rangle \xrightarrow{a}$ $\langle\, x_a, \sigma_a \,\rangle \wedge \langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$,
3. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv \partial_A(x_d)$ and $p_{d'} \equiv \partial_A(x_{d'})$. Rule 37 and result 1 of the induction on x give $\langle\, p_{d'}, \sigma_{d'} \,\rangle \stackrel{d-d'}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$.

To prove Lemma 4.55, we assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $\langle\, p_{d'}, \sigma_{d'} \,\rangle \equiv \langle\, \partial_A(x_{d'}), \sigma_{d'} \,\rangle$ we have $a \notin A$ and using Rule 36 and result 2 of the induction on x we find that $p_{d',a} \equiv \partial_A(x_{d',a})$. Now, by result 2 of the induction on x we know that there are x_a and $x_{d,a}$, such that $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle$ and $\langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$. Using Rule 36 we obtain $\langle\, \partial_A(x), \sigma \,\rangle \xrightarrow{a} \langle\, \partial_A(x_a), \sigma_a \,\rangle$ and $\langle\, \partial_A(x_d), \sigma_d \,\rangle \xrightarrow{a} \langle\, \partial_A(x_{d,a}), \sigma_{d',a} \,\rangle$. So, we have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma_a \,\rangle$ and $\langle\, p_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a} \,\rangle$ with $p_a \equiv \partial_A(x_a)$ and $p_{d,a} \equiv \partial_A(x_{d,a})$.

To prove Lemma 4.56, we use result 3 of the induction on x, which tells us that $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\downarrow$. Since, $p_{d'} \equiv \partial_A(x_{d'})$, Rule 35 does not apply to $p_{d'}$ and we obtain $\langle\, p_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Maximal progress: Suppose $\langle\, p, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$ and $\langle\, p, \sigma \,\rangle \stackrel{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, where $p \equiv \pi(x)$ and $d' < d$. Using Rule 40 we obtain $\langle\, x, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$ and we can conclude that $p_d \equiv \pi(x_d)$. Also, $\langle\, x, \sigma \,\rangle \stackrel{d'}{\longmapsto} \langle\, x_{d'}, \sigma_{d'} \,\rangle$ and we can conclude that $p_{d'} \equiv \pi(x_{d'})$.

Induction on x gives us

1. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \stackrel{d-d'}{\longmapsto} \langle\, x_d, \sigma_d \,\rangle$,
2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle\, x_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, x_{d',a}, \sigma_{d',a} \,\rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle \wedge \langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$,
3. $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv \pi(x_d)$ and $p_{d'} \equiv \pi(x_{d'})$. Rule 40 and result 1 of the induction on x then give $\langle\, p_{d'}, \sigma_{d'} \,\rangle \stackrel{d-d'}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$ provided that $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\rightarrow$. This follows from the inverse of result 2 and the fact that $p \equiv \pi(x)$. We know that $\langle\, p, \sigma \,\rangle \stackrel{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle$ and Rule 40 then gives $\langle\, x, \sigma \,\rangle \not\rightarrow$. Consequently, $\langle\, x_{d'}, \sigma_{d'} \,\rangle \not\rightarrow$.

To prove Lemma 4.55, we assume $\langle\, p_{d'}, \sigma_{d'} \,\rangle \xrightarrow{a} \langle\, p_{d',a}, \sigma_{d',a} \,\rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv \pi(x_{d'})$ we find using Rule 39 and result 2 of the induction on x that $p_{d',a} \equiv \pi(x_{d',a})$. Now, by result 2 of the induction on x we know that there are x_a and $x_{d,a}$, such that $\langle\, x, \sigma \,\rangle \xrightarrow{a} \langle\, x_a, \sigma_a \,\rangle$ and $\langle\, x_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, x_{d,a}, \sigma_{d,a} \,\rangle$. Using Rule 39 we obtain $\langle\, \pi(x), \sigma \,\rangle \xrightarrow{a} \langle\, \pi(x_a), \sigma_a \,\rangle$ and $\langle\, \pi(x_d), \sigma_d \,\rangle \xrightarrow{a} \langle\, \pi(x_{d,a}), \sigma_{d,a} \,\rangle$. So, we have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_a, \sigma_a \,\rangle$ and $\langle\, p_d, \sigma_d \,\rangle \xrightarrow{a} \langle\, p_{d,a}, \sigma_{d,a} \,\rangle$ with $p_a \equiv \pi(x_a)$ and $p_{d,a} \equiv \pi(x_{d,a})$.

To prove Lemma 4.56, we use result 3 of the induction on x, which tells us that $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$. Since $p_{d'} \equiv \pi(x_{d'})$, Rule 38 does not apply to $p_{d'}$ and we obtain $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$.

Abstraction: Suppose $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle$ and $\langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, where $p \equiv \pi(x)$ and $d' < d$. Using Rule 44 we obtain $\langle x, \sigma \rangle \overset{d}{\longmapsto} \langle x_d, \sigma_d \rangle$ and we can conclude that $p_d \equiv \tau_A(x_d)$. Also, $\langle x, \sigma \rangle \overset{d'}{\longmapsto} \langle x_{d'}, \sigma_{d'} \rangle$ and we can conclude that $p_{d'} \equiv \tau_A(x_{d'})$.

Induction on x gives us

1. $\langle x_{d'}, \sigma_{d'} \rangle \overset{d-d'}{\longmapsto} \langle x_d, \sigma_d \rangle$,
2. $\forall x_{d',a}, \sigma_{d',a}, a : \langle x_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle x_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists x_a, \sigma_a, x_{d,a}, \sigma_{d,a} : \langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma_a \rangle \wedge \langle x_d, \sigma_d \rangle \xrightarrow{a} \langle x_{d,a}, \sigma_{d,a} \rangle$,
3. $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$.

Lemma 4.53 is easily proved since we already derived that $p_d \equiv \tau_A(x_d)$ and $p_{d'} \equiv \tau_A(x_{d'})$. Rule 40 and result 1 of the induction on x give $\langle p_{d'}, \sigma_{d'} \rangle \overset{d-d'}{\longmapsto} \langle p_d, \sigma_d \rangle$.

To prove Lemma 4.55, we assume $\langle p_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle$ for some $p_{d',a}$, $\sigma_{d',a}$, and a. Since $p_{d'} \equiv \tau_A(x_{d'})$, we find using Rule 42 or 43 and result 2 of the induction on x that $p_{d',a} \equiv \tau_A(x_{d',a})$. We can distinguish two cases: $a \not\equiv \tau$ or $a \equiv \tau$. Suppose $a \not\equiv \tau$, then we know by result 2 of the induction on x that there are x_a, σ_a, $x_{d,a}$, and $\sigma_{d,a}$ such that $\langle x, \sigma \rangle \xrightarrow{a} \langle x_a, \sigma_a \rangle$, $\langle x_d, \sigma_d \rangle \xrightarrow{a} \langle x_{d,a}, \sigma_{d,a} \rangle$, and $a \notin A$. Using Rule 42 we obtain $\langle \tau_A(x), \sigma \rangle \xrightarrow{a} \langle \tau_A(x_a), \sigma_a \rangle$ and $\langle \tau_A(x_d), \sigma_d \rangle \xrightarrow{a} \langle \tau_A(x_{d,a}), \sigma_{d,a} \rangle$. So, we have $\langle p, \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_a \rangle$ and $\langle p_d, \sigma_d \rangle \xrightarrow{a} \langle p_{d,a}, \sigma_{d,a} \rangle$ with $p_a \equiv \tau_A(x_a)$ and $p_{d,a} \equiv \tau_A(x_{d,a})$. In case $a \equiv \tau$, we know by result 2 of the induction on x that there are $x_{a'}$ and $x_{d,a'}$, such that $\langle x, \sigma \rangle \xrightarrow{a'} \langle x_{a'}, \sigma_{a'} \rangle$, $\langle x_d, \sigma_d \rangle \xrightarrow{a'} \langle x_{d,a'}, \sigma_{d,a'} \rangle$, and $a' \in A$. Using Rule 43 we obtain $\langle \tau_A(x), \sigma \rangle \xrightarrow{\tau} \langle \tau_A(x_\tau), \sigma_\tau \rangle$ and $\langle \tau_A(x_d), \sigma_d \rangle \xrightarrow{\tau} \langle \tau_A(x_{d,\tau}), \sigma_{d,\tau} \rangle$. So, we have $\langle p, \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_a \rangle$ and $\langle p_d, \sigma_d \rangle \xrightarrow{a} \langle p_{d,a}, \sigma_{d,a} \rangle$ with $p_a \equiv \tau_A(x_a)$, $p_{d,a} \equiv \tau_A(x_{d,a})$, and $a \equiv \tau$.

To prove Lemma 4.56 we use result 3 of the induction on x, which tells us that $\langle x_{d'}, \sigma_{d'} \rangle \not\downarrow$. Since $p_{d'} \equiv \tau_A(x_{d'})$, Rule 41 does not apply to $p_{d'}$ and we obtain $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$. □

4.18 · Process specifications in χ_σ

In this section, we describe how process specifications can be written in χ_σ. It is important to know that the process specification mechanism of χ_σ is based on syntactic replacement. So, formal parameters are replaced by actual parameters. Furthermore, it is assumed that instantiation is finite and therefore, recursive process specifications are not allowed. However, infinite behaviour can be specified by the repetition operator as discussed in Section 4.9.

In χ_σ, process specifications are equations. The general form is $P(x_1, \ldots, x_n) = p$, where P is an identifier, $x_1, \ldots, x_n$ are programming variables, and p is a χ_σ process possibly containing the programming variables $x_1, \ldots, x_n$. An example is presented in Chapter 8.

4.19 · Discussion

We defined the formal syntax and semantics of χ_σ processes. This is done in an SOS style. Consequently, the behaviour of χ_σ processes is defined by deduction rules. In addition, an equivalence relation, called strong bisimulation, has been defined on processes. This is an improvement with respect to χ, where a notion of equivalence is absent. Furthermore, we showed that process operators have desired properties, the deduction rules are meaningful, and strong bisimulation is a congruence for all process operators. Together this embodies a mathematical framework for χ_σ; a necessary ingredient for a formal method.

We conclude that SOS theory is well suited to define the operational semantics of χ_σ. Furthermore, the mathematical framework of χ_σ could be set up in a way similar to the frameworks of other languages defined using SOS theory. This confirms what was suggested in our motivation for the third alternative in Section 1.3. However, as became apparent in Section 4.16, combining SOS with AS could not be done as formal as we would like.

Relation between χ and χ_σ · 5

This chapter discusses the relation between χ and χ_σ. Its purpose is twofold: describe improvements and shortcomings of χ_σ with respect to χ, and provide evidence to judge whether the formalisation has been successful. Therefore, similarities are discussed and differences are explained.

This chapter is organised as follows. Firstly, Sections 5.1 through 5.3 discuss fundamental concepts of both languages: concurrent processes, communication, and imperative programming. Secondly, Sections 5.4 through 5.11 discuss features of χ that χ_σ misses. Thirdly, Section 5.12 explains differences between the selection constructs of χ and χ_σ, and Section 5.13 describes the abstraction mechanism of χ_σ, which is not present in χ. This chapter is concluded by a discussion in Section 5.14.

5.1 · Concurrent processes

One of the most important concepts of χ is that of a process. Usually, a single entity of a production system is modelled by one process, for instance, a buffer or a machine. In order to support structural design of production systems, groups of related entities can be modelled by systems, which are aggregates of concurrent processes or other systems. To this end, χ incorporates the parallel composition operator '$\|$'. Note that the '$\|$' operator has a special status with respect to other operators, since it is the only operator that can group χ processes; the other operators group χ statements. Since the process concept of χ is very effective to model industrial systems, as motivated in [8], χ_σ has processes, too. Also, χ_σ processes can be grouped in the same way as in χ, that is, by means of the '$\|$' operator. The only difference is that χ_σ does not have the notion of systems. Everything is a process and the parallel composition operator can be used to group processes into a new process.

Given that in production systems many activities happen at the same time, concurrent processes are very well suited to model these systems. Therefore, χ_σ has concurrent processes too. In most descriptions of the operational semantics of χ [2, 8, 148], it is not stated explicitly whether the language has a *true* concurrency semantics or an *interleaving* concurrency semantics. In the description of hybrid χ [68], an explicit choice for an interleaving semantics is made. Furthermore, since one of the goals of χ_σ is to make available process algebra techniques to production systems analysis and since an interleaving semantics is standard in process algebras, χ_σ also has an interleaving concurrency semantics.

Furthermore, parallelism inside processes can result in more concise specifications. Situations occur, in particular in real-time control, in which a process waits for several activities to occur, but the order in which they occur is irrelevant. The standard strategy to solve this problem in χ is to introduce boolean programming variables in order to keep track of which activities still have to occur and which activities have occurred already. For example, suppose a process has to send two values and receive two values, but the order of the values is irrelevant. In χ this is specified as

$$
\begin{array}{l}
b_0 := \mathit{false}\ ;\ b_1 := \mathit{false}\ ;\ b_2 := \mathit{false}\ ;\ b_3 := \mathit{false}\ ; \\
*[\ \neg b_0\ ;\ m_0\,!\,e_0 \longrightarrow b_0 := \mathit{true} \\
\ \ [\!]\ \neg b_1\ ;\ m_1\,!\,e_1 \longrightarrow b_1 := \mathit{true} \\
\ \ [\!]\ \neg b_2\ ;\ m_2\,?\,x_0 \longrightarrow b_2 := \mathit{true} \\
\ \ [\!]\ \neg b_3\ ;\ m_3\,?\,x_1 \longrightarrow b_3 := \mathit{true} \\
\ \ ].
\end{array}
$$

The boolean programming variables b_0, b_1, b_2, and b_3 keep track of which activity has occurred. If b_0 is *false*, the send statement $m_0\,!\,e_0$ has not been executed. After the send statement is executed, b_0 is set to *true*. A similar explanation can be given for the other programming variables. As far as functionality concerns, the solution is correct, however, it is not really an elegant solution. In contrast, in χ_σ we can write $m_0\,!\,e_0 \parallel m_1\,!\,e_1 \parallel m_2\,?\,x_0 \parallel m_3\,?\,x_1$.

5.2 · Communication

In both languages, communication between concurrent processes is implemented by channels. Other possibilities for inter-process communication are synchronizing actions via a (user-definable) communication function in the style of ACP, shared variables (or memory), and remote procedure calls. Since communication channels

are a characteristic feature of the χ language, χ_σ has communication channels too. Note that communication channels can be seen as a special case of synchronizing actions where the communication function is defined implicitly. For example, a send process $\sim m\,!\,e$ or a receive processes $\sim m\,?\,x$, implicitly defines the (commutative) communication function γ for send and receive actions over channel $\sim m$: $\gamma(sa(\sim m, e), ra(\sim m, x)) = ca(\sim m, x, e)$.

In addition to communication channels, χ_σ has shared programming variables. This is a result of the fact that the parallel operator of χ lost its special status in χ_σ and can be used inside state operator processes. For example, the following process has a programming variable x that is shared by two concurrent processes: $[\![\, x \mapsto 0 : \lambda_s \mid x = 0 :\rightarrow x := 1 \parallel x > 0 :\rightarrow x := 0 \,]\!]$.

Channels in χ_σ have fewer restrictions than channels in χ. First of all, χ_σ channels are bidirectional, whereas in χ, channels are unidirectional. Therefore, a process can use the same channel both to send values and to receive values. The main reason χ_σ does not have unidirectional channels is that without them the port concept need not be formalized. Moreover, it makes some specifications more concise (see Section 8.8). Since it is possible to use χ_σ channels as if they were unidirectional, no expressive power is lost. Sometimes models can be made even more intuitive, since bidirectional communication devices do exist in real-life and these can be modelled by bidirectional channels in χ_σ, whereas they would have to be modelled as two unidirectional channels in χ.

Secondly, χ_σ channels can connect more than two processes, whereas in χ a channel is connected to exactly two processes. Therefore, one channel can be used to connect an arbitrary number of processes.

Thirdly, communication channels of χ are typed, whereas in χ_σ, channels are not. Untyped channels potentially introduce more typing errors. However, in χ_σ this effect is limited because communication is still typed in the sense that a value can only be received in a programming variable of the same type. We preferred untyped channels, because they simplify the formalisation.

Finally, χ_σ channels can be used to communicate both synchronously and asynchronously. Communication takes place by first putting a value on a channel and then reading that value from the channel. In synchronous communication, this happens during one action (a communication action) whereas in asynchronous communication, this happens in two successive actions (a send action followed by a receive action). So, channels can be seen as one-place buffers. Reading is *non-*

destructive: reading a value does not remove it. Therefore, the same value can be read multiple times. On the other hand, χ only has synchronous communication. Although χ_σ can be adapted to mimic exactly the communication behaviour of χ, we think this is not necessary. The current definition of χ_σ allows the synchronous communication of χ and, in addition, it allows asynchronous communication.

In χ_σ, one can define urgent (send and receive) actions. An urgent action is an action that has to occur before time passes. Note that χ_σ's internal action and assignment actions are always urgent, whereas send, receive, and communication actions are delayable. In order to define urgent send, receive, and communication actions, the maximal progress operator of χ_σ can be used. For example, process $p \equiv \pi(\sim m\,!\,1)$ has to send the value 1 over channel $\sim m$ immediately. Here the maximal progress operator prohibits the delay transitions of the send process. The result is that p can only perform an action transition with the action $sa(\sim m, 1)$, hence, an urgent send action. As can be seen, urgent send and delay actions are a result of the interaction between different language constructs of χ_σ.

In χ, urgent send and receive statements are impossible. The best one can do is to define a very small timeout, as in $[\sim m\,!\,1 \,[\!]\, \Delta 0.0001]$. This process wants to engage in a communication over m only if it happens within 0.0001 time units. Note that it is still not an urgent action, because a very small delay is possible before the action is executed. In addition, it is not an elegant solution. In previous versions of χ this problem was noticed already and it was allowed to write $[\mathit{true}\ ;\ \sim m\,!\,1 \longrightarrow \mathsf{skip} \,[\!]\, \mathit{true}\ ;\ \Delta 0 \longrightarrow \mathsf{skip}]$ in order to prevent the send statement from delaying. However, the informal explanation of $\Delta 0$ depended on the context, which made it a difficult construct to formalize.

Still, urgent send and receive actions are useful features of a modelling language for industrial systems. For example, in a model of a conveyor belt, an urgent send action concisely models the obligation to remove a product that has reached the end of the belt. If the send action is not urgent, the product could stay at the end of the belt indefinitely and this may not always correspond to the real-life situation.

5.3 · Imperative programming

Imperative programming is a style in which there is a notion of states and transitions. Usually, states are defined by the values of programming variables. Program

execution consists of transitions between states; each transition can change the state by modifying the values of some programming variables. In a sequential programming language, the main construct to change the state is the assignment. In a concurrent language, communication constructs, like χ_σ's send and receive processes, can also change the state. Another programming style is called *functional programming* [32]. Here, a program is a (mathematical) function and program execution is just function application. The reason χ_σ has an imperative programming style, is that χ has it too. A consequence of this decision is the introduction of the state operator in χ_σ.

Related to imperative programming is scoping of variables. The scope of a variable defines that part of the program in which this variable can be used. The state operator of χ_σ defines a scope for programming variables (and channels). Since state operators can be nested, χ_σ has nested scopes. In contrast, the scope of programming variables in χ is the process in which they are declared. Since χ processes cannot be nested, χ does not have nested scopes.

The multiple assignment construct of χ enables one to assign values to several programming variables at once. This construct is not present in χ_σ. This is not really a problem, since by introducing additional programming variables, multiple assignments can be written as a series of normal assignments.

5.4 • Real numbers

As mentioned in Chapter 2, χ_σ does not have a MEL specification for the real numbers. The reason is that a MEL specification of real numbers is impossible. This can be understood easily, since MEL specifications have an initial algebra semantics (see Section C.2). Consequently, for every element of a model of a MEL specification, there is a syntactic representation and, therefore, these sets are countable. Since the set of real numbers is uncountable, it is impossible to define a MEL specification of the real numbers. We think this is a serious omission of χ_σ that should be resolved in future versions. We are aware that it is possible to define *part* of the real numbers in a first order algebraic setting [169]. The authors claim that it is strong enough for many practical purposes, in particular certain numerical applications. A similar approach can be taken for χ_σ using MEL.

5.5 · Probabilities and distributions

A language with probabilistic or stochastic constructs is very useful to model industrial systems. Therefore, χ has a distribution data type. Elements of this data type are stochastic distributions, for example, uniform distributions with given lower and upper bound and Gamma distributions with given mean and standard deviation. A distribution can be used to take samples. These samples can be used to model, for instance, inter-arrival times of products or processing times of machines.

As mentioned in Section 1.4, χ_σ does not have these probabilistic features. We admit that the lack of probabilistic language constructs is a serious omission if χ_σ is to be really useful in modelling industrial systems. Therefore, research on future versions of χ_σ should focus on the integration of probabilities. Note, however, that tools developed for χ_σ have extensions that support probabilities and distributions (see Section 7.6).

Probabilistic and stochastic formalisms are subject of active research in Formal Methods. An introduction to three different probabilistic models is presented in [78]. A widely accepted definition of weak-bisimulation on probabilistic transition systems is described in [183, 184]. The first definition of (strong) probabilistic bisimulation is given in [127]. In [98, 99] extensions of CCS with both time and probabilistic constructs are described. In [129], a probabilistic extension of CSP is given. The specification language LOTOS is extended with probabilistic constructs in [140]. In [87], stochastic process algebras are used to analyse functional and performance properties of distributed systems. In [7, 114, 15], probabilistic extensions of process algebras are described. A particular stochastic process algebra, called PEPA, is described in [104]. Finally, stochastic automata are discussed in [59].

5.6 · Current time expression

In Chapter 3, we described that χ models can refer to the current time. In a χ model, the read only programming variable τ (which should not be confused with the internal action τ of χ_σ) always has the value of the current time. In many χ specifications, τ is used to compute performance measures, like cycle time and throughput. In these situations, τ is not needed to specify the behaviour of a particular system, but it is merely used to analyze the system. On the other

hand, there are χ specifications in which τ plays a crucial role. The conveyor belt process C on page 46 is a good example. Here, each time a product is received by the conveyor, τ is used to compute the time at which the product should leave the conveyor. In situations like this, the specification can usually be rewritten into a functionally equivalent specification that does not use τ. For instance, the conveyor belt system can be rewritten into the parallel composition of n processes, where n is the capacity of the conveyor belt. Note that the original χ specification of conveyor C has infinite capacity.

The remainder of this section presents four alternatives to incorporate a current time expression in χ_σ. Firstly, it is possible to add time to action transitions, delay transitions, and terminations of χ_σ-LTSs. In literature, two variants can be found. First, it is possible to add a *time stamp* to the transitions [182, 61, 70]. Second, it is possible to add a *start time* and an *end time* to the transitions of the χ_σ-LTSs [90, 115, 16]. If the first approach is chosen for χ_σ, action transitions, delay transitions, and terminations would get the form $\langle\, p, \sigma \,\rangle \xrightarrow{a,t} \langle\, p', \sigma' \,\rangle$, $\langle\, p, \sigma \,\rangle \overset{d,t}{\longmapsto} \langle\, p', \sigma' \,\rangle$, and $\langle\, p, \sigma \,\rangle{\downarrow}^t$, respectively. Here $t \in R_{\geq 0}$ denotes the time at which the transition or termination occurs or starts. If the second approach is chosen for χ_σ, action transitions, delay transitions, and terminations would get the form $\langle\, p, \sigma, t \,\rangle \xrightarrow{a} \langle\, p', \sigma', t' \,\rangle$, and $\langle\, p, \sigma, t \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma', t' \,\rangle$, and $\langle\, p, \sigma, t \,\rangle{\downarrow}$, respectively. Here, $t, t' \in R_{\geq 0}$ are the start time and end time of the action and delay transitions. Note that termination is instantaneous and does not have an end time. In addition to changing the action transitions, delay transitions, and terminations, a new expression denoting the current time has to be introduced. The value of this expression is the value of the time stamp t in the first approach and the value of the start time t in the second approach.

Another option is to introduce a data type of real-valued *clocks*, as in *timed-automata* [64, 3, 4]. The value of a clock increases during delay transitions and it can be reset to 0 by a special language construct. In addition, a clock can be used as a read-only programming variable in expressions, thus providing a current time expression. If a data type of real-valued clocks is included in χ_σ, the state-stacks could be used to keep track of the values of the clocks. Just as programming variables and channels, clocks can be represented by identifiers to which values are associated. Consequently, processes can have multiple clocks possibly occurring in different (nested) scopes. In addition, a *clock reset* process and a *clock reset* action should be defined. The clock reset process takes a clock as argument and can perform a clock reset action on that clock. Finally, the deduction rules with

delay transitions as a conclusion need to be adapted in order to realize the increase of the clock values present in the stack. Suppose the clocks are $\gamma_0, \ldots, \gamma_{n-1}$ for some natural number n. Then in general, the conclusions will have the form $\langle\, p, \sigma \,\rangle \xrightarrow{d} \langle\, p', \sigma[\sigma(\gamma_0) + d/\gamma_0][\sigma(\gamma_1) + d/\gamma_1] \ldots [\sigma(\gamma_{n-1}) + d/\gamma_{n-1}] \,\rangle$.

For example, suppose γ is a clock identifier. The clock reset process for γ and the clock reset action for γ are denoted by $\mathsf{reset}\ \gamma$ and $cra(\gamma)$, respectively. The following example defines a machine that repeatedly receives a product in x, processes this product for 5 time units, and send that product away via $\sim out$. The total waiting time for all these products is computed in y (using the clock γ):

$$[\![\, (\gamma \mapsto 0.0) : (x \mapsto \bot) : (y \mapsto 0.0) : \lambda_s \mid P^* \ ;\ \delta \,]\!],$$

where $P = (\mathsf{reset}\ \gamma \ ;\ \sim in\,?\,x \ ;\ y := y + \gamma \ ;\ \Delta 5 \ ;\ \sim out\,!\,x)$. Suppose this process has to wait 2.57 time units before it receives a product (represented by the value 0) over channel $\sim in$. This results in the following transitions (we omit the stacks):

$$\begin{array}{l}
[\![\, (\gamma \mapsto 0.0) : (x \mapsto \bot) : (y \mapsto 0.0) : \lambda_s \mid P^* \ ;\ \delta \,]\!] \\
\xrightarrow{cra(\gamma)} \\
[\![\, (\gamma \mapsto 0.0) : (x \mapsto \bot) : (y \mapsto 0.0) : \lambda_s \\
\mid (\varepsilon \ ;\ \sim in\,?\,x \ ;\ y := y + \gamma \ ;\ \Delta 5 \ ;\ \sim out\,!\,x) \ ;\ P^* \ ;\ \delta \\
]\!] \\
\xrightarrow{2.57} \\
[\![\, (\gamma \mapsto 2.57) : (x \mapsto \bot) : (y \mapsto 0.0) : \lambda_s \\
\mid (\sim in\,?\,x \ ;\ y := y + \gamma \ ;\ \Delta 5 \ ;\ \sim out\,!\,x) \ ;\ P^* \ ;\ \delta \\
]\!] \\
\xrightarrow{ra(\sim in, x)} \\
[\![\, (\gamma \mapsto 2.57) : (x \mapsto 0) : (y \mapsto 0.0) : \lambda_s \\
\mid (\varepsilon \ ;\ y := y + \gamma \ ;\ \Delta 5 \ ;\ \sim out\,!\,x) \ ;\ P^* \ ;\ \delta \\
]\!] \\
\xrightarrow{aa(y, 2.57)} \\
[\![\, (\gamma \mapsto 2.57) : (x \mapsto 0) : (y \mapsto 2.57) : \lambda_s \mid (\varepsilon \ ;\ \Delta 5 \ ;\ \sim out\,!\,x) \ ;\ P^* \ ;\ \delta \,]\!] \\
\xrightarrow{5} \\
[\![\, (\gamma \mapsto 7.57) : (x \mapsto 0) : (y \mapsto 2.57) : \lambda_s \mid (\Delta 5 - 5 \ ;\ \sim out\,!\,x) \ ;\ P^* \ ;\ \delta \,]\!] \\
\xrightarrow{sa(\sim out, 0)} \\
[\![\, (\gamma \mapsto 7.57) : (x \mapsto 0) : (y \mapsto 2.57) : \lambda_s \mid \varepsilon \ ;\ P^* \ ;\ \delta \,]\!].
\end{array}$$

Since delaying is a global property, it seems natural to update all clocks during delay transitions, even the ones that are not visible in the stack. Note that this changes the scoping properties of the state operator. On the other hand, if only visible clocks are updated during delay transitions, hiding a clock has the effect of stopping that clock.

In addition, it is possible to specify a clock process explicitly. In [42], this approach is taken to add time to the untimed specification language PROMELA and in [178, Chapter 14] this approach is taken to add time to (untimed) CSP. In χ_σ, a clock process could be defined by $[\![c :\mapsto 0 : \lambda_s \mid (\Delta t \ ; \ c := c + t)^* \ ; \ \delta]\!]$, where $t \in R_{>0}$ is an arbitrary, but fixed, positive real number denoting the size of the minimal time step. This process models the current time by the real-valued programming variable c which is updated after delay transitions. The programming variable c represents the current time expression. The advantage of this alternative is that χ_σ is powerful enough to support it; the deduction rules need not be adapted. The disadvantage is that the clock is updated at discrete times, which effectively means that the model has a discrete time domain (with time unit t). Another potential problem with this alternative is that the update of c is an ordinary assignment action that will be interleaved arbitrarily with other actions. Therefore, if another process uses c as the current time expression, it is possible that c has not been updated yet. However, a simple solution to this problem is to split the Δt process into two sequential delay processes Δt_0 and Δt_1, such that $t_0 + t_1 = t$ and put the assignment to c in between: $[\![c :\mapsto 0.0 : \lambda_s \mid (\Delta t_0 \ ; \ c := c + t \ ; \ \Delta t_1)^* \ ; \ \delta]\!]$.

Finally, alternatives to incorporate continuous behaviour in χ_σ can be investigated. This will make χ_σ a hybrid formalism in which both discrete and continuous behaviour can be modelled. Well known hybrid formalisms are hybrid automata [102] and hybrid I/O automata [132]. Continuous functions can be specified by systems of differential equations. A clock can be modelled by a continuous function with derivative 1. An advantage of this approach is that clocks are modelled in the (hybrid) language, just as in the previous solution. Therefore, the deduction rules need not be changed and all lemmas and theorems still hold. Eventually, χ_σ should include constructs to specify hybrid systems.

Considering these alternatives, we see that all but one of the alternatives mentioned above suffer from the fact that some properties do not hold anymore. For example, Lemma 4.47 (Time factorisation) on page 113 states that $\Delta e \ ; p [\![\Delta e + e' \leftrightarrows \Delta e \ ; (p [\![\Delta e')$. Suppose the current time is denoted by the expression τ_σ. Then, the lemma does not hold anymore, since we do not have $\Delta \tau_\sigma \ ; p [\![\Delta \tau_\sigma + 2\tau_\sigma \leftrightarrows \Delta \tau_\sigma \ ; (p [\![\Delta 2\tau_\sigma)$.

The clock process solution does not have this drawback. However, the clock process solution turns χ_σ into a discrete time language.

5.7 · List-like data types

In χ several list-like data types, each with its own interface, are defined: lists, strings, files, and tuples. In contrast, χ_σ has just lists and tuples. There are at least two reasons why χ_σ has no strings and files. First of all, we think strings and files are just special types of lists: lists of characters and lists of bytes, respectively. Since χ_σ has lists, there is no need to include strings and files, too. Secondly, and more importantly, strings and files are mostly used for input and output of model parameters and analysis of data. Therefore, they are usually not used to model aspects of industrial systems, but to provide analysis functionality. We think analysis functionality should not be part of the specification, but should be provided by an experiment environment (see Section 7.8).

5.8 · Ranges

Ranges and the associated range programming variables are powerful constructs to model large systems. For example, to instantiate five parallel processes P, one can write $\| i : \text{nat} \leftarrow 0..5 : P(i)$. Recall that a range includes its lower bound, but excludes its upper bound. Semantically, the expression above is equivalent to $P(0) \| P(1) \| P(2) \| P(3) \| P(4)$. Furthermore, χ requires the range expressions to be constant expressions. Per definition, constant expressions do not depend on the value of programming variables. Therefore, ranges are just syntactic sugar and we did not include them in χ_σ.

5.9 · Terminate statement

In χ, the terminate statement is used to terminate a simulation run. That is, as soon as it is executed, it blocks all activity of the simulation. The terminate statement is introduced in χ mainly to ease simulation-based analysis. As acknowledged by the χ developers, the functionality of this statement should really be provided by an experiment environment. Therefore, the terminate statement is not included in χ_σ.

5.10 · Input and output statements

The input and output statements of χ are $?x$ and $!e$, respectively. These statements are used to provide an interactive user-interface for χ specifications. An interactive user-interface can provide functionality to set parameters of a simulation, which is very useful during analysis of the specification.

However, a specification in a modelling language should model a particular real-life system and nothing more. Only if there is a need to *model* input and output behaviour of a system, should the modelling language provide constructs to specify this behaviour. However, χ (χ_σ) already has these constructs, namely, normal send and receive statements (processes). Therefore, we did not include input and output processes in χ_σ.

We are aware of the fact that during analysis, whether it is simulation or verification, it is important to have access to the data of a concrete instantiation of a model. If input and output statements are not available, alternatives should be provided. The alternative we propose is an experiment environment with access routines or functions to manipulate concrete models. This alternative supports a clear separation between functionality of the model and functionality to analyze the model. In a verification setting this is important, since usually only the functionality of the model needs to be verified, not the functionality to analyze the model. In Section 7.8, we discuss the χ_σ experiment environment in more detail.

5.11 · Functions

The χ_σ language does not have a function definition mechanism comparable to χ. However, like all built-in functions of χ_σ, user-defined functions can be specified in MEL. Since χ has always had the requirement that user-defined functions be 'mathematical' functions (in the sense that they return the same value for the same parameters) it is possible to translate χ functions into MEL functions.

5.12 · Selection

As described in Chapter 3, χ has two selection statements: the guarded command statement and the selective waiting statement. A drawback of these statements is that they are context dependent. That is, depending on the context (non-repetitive

versus repetitive) they can or cannot deadlock. Another drawback is that they have similar functionality, which can be confusing. Furthermore, they have different delay behaviour. To be more specific, the guarded command statement does not have time factorisation, whereas the selective waiting statement does have time factorisation. The cause of these problems is that too much functionality has been put in one statement. In χ_σ, this functionality is distributed over separate process operators. This is illustrated by the translation of the guarded command statement and the selective waiting statement in the next chapter.

5.13 · Abstraction

The abstraction operator of χ_σ enables one to hide certain actions. This feature is necessary if one wants to check formally if an implementation satisfies a specification. As described in Chapter 1, such checks are typical examples of functional analysis. Since one of the main reasons to formalise χ was to enable functional analysis, the abstraction operator has been incorporated in χ_σ.

5.14 · Discussion

In this chapter, the relation between χ and χ_σ is discussed. Its purpose is twofold: describe improvements and shortcomings of χ_σ with respect to χ, and provide evidence to judge whether the formalisation has been successful.

We believe χ_σ improves upon χ in several ways. First of all, some restrictions in χ are not present in χ_σ. For example, in χ_σ, the parallel composition operator and the state operator can be mixed freely with other process operators. Also, the concept of unidirectional typed channels between exactly two processes has been generalised to bidirectional untyped channels between many processes. As a result, the number of channels needed in a χ_σ specification is usually less than in the corresponding χ specification. Also, with respect to orthogonality of language constructs, χ_σ is an improvement of χ. For example, with respect to selection, time factorisation, and guarding, χ_σ has a clear separation of concerns, whereas χ has not. Another improvement is the ability to perform specification-implementation checks in χ_σ. These checks are possible because of the abstraction operator.

Some constructs of χ have been excluded from χ_σ. The language constructs excluded were (mostly) used for analysis purposes. Functionality to analyse χ_σ

specifications should not be provided by the language, but by a mathematical framework and by an experiment environment. Examples of χ constructs excluded from χ_σ are the current time expression τ and the terminate statement.

We are aware of the fact that χ has useful features which χ_σ lacks. For instance, a data type of real numbers, a data type of distributions, and syntactic sugar to denote ranges. In future versions of χ_σ, these features should be included, because they are very useful to analyse large, real-life industrial systems.

Translation from χ to χ_σ · 6

This chapter defines a translation scheme from χ into χ_σ. The purpose of this chapter is to show that χ_σ is a formal version of χ. In particular, the 'look and feel' of the languages should the same, and the translation should be straightforward. Together with the formal semantics of χ_σ the translation provides the formal semantics of χ.

The translation scheme is defined as a 'function' from syntactic entities in χ to syntactic entities in χ_σ: $\mathcal{T} : \chi \rightarrow \chi_\sigma$. The function $\mathcal{T}$ has been designed such that it can be automated. In this respect it is interesting to note that the translation scheme is currently used in new implementations of χ tools (the *chipy* project [38]).

This chapter has the same structure as Chapter 3. That is, first, we translate type aliases and constants (Sections 6.1 and 6.2). Then, we translate processes, systems, and functions (Sections 6.3, 6.4, and 6.5). Next, we show how to translate χ experiments (Section 6.6). This chapter is concluded by a discussion (Section 6.7).

6.1 · Type aliases

Type alias definition has been defined in Section 3.1. Although type aliases usually increase readability, they are just syntactic sugar. Therefore, χ_σ does not have type aliases.

6.2 · Constants

Constant definition has been defined in Section 3.2. Constants can be defined as nullary functions in a MEL specification. In addition, the MEL specification should contain an equation that defines the value of the constant.

6.3 · Processes

In Section 3.3, the syntax of process definitions in χ has been defined. Here, we define the translation of χ process definitions into χ_σ process definitions and the translation of χ statements into χ_σ processes.

Process definitions in χ are translated into process definitions in χ_σ. There are several issues involved. First of all, local programming variables of χ processes should be mapped onto local programming variables in χ_σ processes. Therefore, we translate χ process bodies to χ_σ state operator processes. If the χ process does not have local programming variables, the χ_σ process has an empty state. Note that, according to Lemma 4.32, this does not influence the behaviour of the process. Secondly, formal parameters of χ processes are both formal parameters and local programming variables of χ_σ processes. The reason for this is that inside a χ process, formal parameters can be used as local programming variables, whereas in χ_σ processes, formal parameters are replaced by the actual parameters upon instantiation (see Section 4.18), and cannot be used as local programming variables. By defining formal parameters of χ processes in χ_σ processes both as formal parameters and as local programming variables initialized by the actual value of the formal parameters, this problem is solved. For instance, the χ process definition

$$\mathsf{proc}\ P(x : \mathrm{int}) = [\![\ y : \mathrm{nat} \mid S\ ]\!],$$

where S is an arbitrary χ statement, is translated into

$$P(x : \mathrm{int}) = [\![\ (x : \mathrm{int} \mapsto x) : (y : \mathrm{nat} \mapsto \bot) : \lambda_s \mid \mathcal{T}(S)\ ]\!].$$

The instantiation mechanism of χ_σ ensures that the local programming variable x of p is initialized with the actual value of parameter x. Notice, that the valuations are typed. In a typed valuation, identifier i is of type t. This is denoted by $i : t \mapsto c$. Typed valuations should be type-correct, meaning, the type of the identifier should be the same as the type of the value.

Channel parameters are treated differently, since in χ channel parameters cannot be used as local programming variables. Therefore, the translation of channel parameters does not involve defining local channel programming variables, but it suffices to define them as channel parameters of χ_σ process definitions. For instance, the process definition

$$\mathsf{proc}\ P(x : \mathrm{int}, c : !\mathrm{bool}) = [\![\ y : \mathrm{nat} \mid S\ ]\!],$$

where S is an arbitrary χ statement, is translated into

$$P(x : \mathrm{int}, c : \mathrm{chan}) = [\![\ (x : \mathrm{int} \mapsto x) : (y : \mathrm{nat} \mapsto \bot) : \lambda_s \mid \mathcal{T}(S)\ ]\!].$$

Note that the channel type of χ is lost, since χ_σ channels do not have types.

The translation of χ statements into χ_σ processes is defined in Tables 6.1 and 6.2. The first table translates basic statements and the second table translates compound statements. The left column in these tables shows the χ statements and the right column shows their translation into χ_σ processes. If the right column is empty, the concerning χ statement cannot be translated directly into χ_σ.

χ statement: S	χ_σ process: $\mathcal{T}(S)$
skip	skip
terminate	
setseed(d, e_{num})	
$x := e$	$x := e$
Δe	Δe
$m\,!\,e$	$m\,!\,e$
$m\,?\,x$	$m\,?\,x$
$m!$	$m\,!\,\langle\rangle$
$m?$	$[\![\ x : \mathrm{tuple}[\,] \mid m\,?\,x\ ]\!]$
$m\sim$	$m\,!\,\langle\rangle \ [\!]\ [\![\ x : \mathrm{tuple}[\,] \mid m\,?\,x\ ]\!]$
$!e$	
$?x$	

Table 6.1 · Translation of basic χ statements into χ_σ processes.

The terminate, setseed (a probabilistic construct), '$!e$', and '$?x$' statements are not translated (see Chapter 5).

The translation of assignment statements assumes that χ expressions can be translated into χ_σ expressions. A similar remark can be made for the delay statement and the send statement.

Send and receive statements are translated into send and receive processes, respectively. Since send and receive statements communicate synchronously, whereas

χ statement: S	χ_σ process: $\mathcal{T}(S)$
$[\ b_1 \longrightarrow S_1$ $[\!]\ b_2 \longrightarrow S_2$ $\vdots$ $[\!]\ b_n \longrightarrow S_n$ $]$	$(\ b_1 :\!\rightarrow \mathsf{skip}\ ;\ \mathcal{T}(S_1)$ $[\!]\ b_2 :\!\rightarrow \mathsf{skip}\ ;\ \mathcal{T}(S_2)$ $\vdots$ $[\!]\ b_n :\!\rightarrow \mathsf{skip}\ ;\ \mathcal{T}(S_n)$ $)$
$^*[\ b_1 \longrightarrow S_1$ $[\!]\ b_2 \longrightarrow S_2$ $\vdots$ $[\!]\ b_n \longrightarrow S_n$ $]$	$(\ b_1 :\!\rightarrow \mathsf{skip}\ ;\ \mathcal{T}(S_1)$ $[\!]\ b_2 :\!\rightarrow \mathsf{skip}\ ;\ \mathcal{T}(S_2)$ $\vdots$ $[\!]\ b_n :\!\rightarrow \mathsf{skip}\ ;\ \mathcal{T}(S_n)$ $)^*\ ;\ \neg(b_1 \vee b_2 \vee \ldots \vee b_n) :\!\rightarrow \varepsilon$
$[\ b_1\ ;\ s_1 \longrightarrow S_1$ $[\!]\ b_2\ ;\ s_2 \longrightarrow S_2$ $\vdots$ $[\!]\ b_n\ ;\ s_n \longrightarrow S_n$ $]$	$(\ b_1 :\!\rightarrow \mathcal{T}(s_1)\ ;\ \mathsf{skip}\ ;\ \mathcal{T}(S_1)$ $[\!]\ b_2 :\!\rightarrow \mathcal{T}(s_2)\ ;\ \mathsf{skip}\ ;\ \mathcal{T}(S_2)$ $\vdots$ $[\!]\ b_n :\!\rightarrow \mathcal{T}(s_n)\ ;\ \mathsf{skip}\ ;\ \mathcal{T}(S_n)$ $)$
$^*[\ b_1\ ;\ s_1 \longrightarrow S_1$ $[\!]\ b_2\ ;\ s_2 \longrightarrow S_2$ $\vdots$ $[\!]\ b_n\ ;\ s_n \longrightarrow S_n$ $]$	$(\ b_1 :\!\rightarrow \mathcal{T}(s_1)\ ;\ \mathsf{skip}\ ;\ \mathcal{T}(S_1)$ $[\!]\ b_2 :\!\rightarrow \mathcal{T}(s_2)\ ;\ \mathsf{skip}\ ;\ \mathcal{T}(S_2)$ $\vdots$ $[\!]\ b_n :\!\rightarrow \mathcal{T}(s_n)\ ;\ \mathsf{skip}\ ;\ \mathcal{T}(S_n)$ $)^*\ ;\ \neg(b_1 \vee b_2 \vee \ldots \vee b_n) :\!\rightarrow \varepsilon$

Table 6.2 · Translation of compound χ statements into χ_σ processes.

send and receive processes communicate both synchronously and asynchronously, the behaviour of the translated version is slightly different. In Section 6.6, we discuss how to prohibit asynchronous communication by means of the encapsulation operator.

In addition, send and receive statements in χ delay until both parties are ready to communicate: urgent communication. However, in χ_σ send and receive processes can delay even if both parties are ready to communicate: delayable communication. In Section 6.6, we discuss how to enforce urgent communication by means of the maximal progress operator.

Directed synchronisation statements are translated into send and receive processes. Since the value communicated is an empty tuple, denoted by $\langle\rangle$, there is no information transfer. Effectively, this means that the two processes synchronize. Note that the receive process is enclosed in a state operator process in order to introduce a fresh programming variable x : tuple[ ].

The undirected synchronization statement has no counterpart in χ_σ. It behaves either as a send synchronisation statement or a receive synchronisation statement. Therefore, the translation is the alternative composition of the translations of these two statements.

This completes the translation of basic χ statements. Next, we describe the translation of compound statements. As mentioned above, this translation is given in Table 6.2.

Note that Table 6.2 introduces skip processes that are not present in the respective χ statements. The reason for this is that guarded command statements and selective waiting statements in χ make a choice after the guards are evaluated, but before the statements following the '$\longrightarrow$' are executed. So, a choice is made 'at the '$\longrightarrow$' symbol'. In contrast, the alternative composition of χ_σ makes a choice by executing an action of one of its alternatives. By translating the '$\longrightarrow$' into a skip process, the selection of an alternative in χ is translated into an explicit action in χ_σ.

In the repetitive guarded command and the repetitive selective waiting, the body is repeated as long as at least one of the guards is true. If none of the guards is true, the repetition terminates successfully. To capture this behaviour, the translation is a sequential composition of a repetition process and an 'exit' process. The repetition process contains the translation of the body of the loop, which is the translation of a guarded command or a selective waiting statement. The exit process contains an empty process guarded by the negation of the disjunction of the guards of the alternatives.

6.4 · Systems

In Section 3.4, the syntax of system definitions in χ has been defined. The translation of system definitions is similar to the translation of process definitions. Therefore, the system definition

$$\mathsf{syst}\ P(x : \text{int}, c : ?\text{int}) = [\![\ y : \text{int} \mid S\ ]\!],$$

where S is a valid system body of χ, is translated into

$$P(x : \text{int}, c : \text{chan}) = [\![\ (x : \text{int} \mapsto x) : (y : \text{int} \mapsto \bot) : \lambda_s \mid S\ ]\!].$$

Notice, that the body of a χ system is a parallel composition of several process or system instantiations. Therefore, the translation of a system body is the identity function.

6.5 · Functions

In Section 3.5, the syntax of function definitions in χ has been defined. Function definitions are not translated into χ_σ. This does not mean user-definable functions are impossible in χ_σ. It just means the function definitions have to be translated manually (see Section 5.11).

6.6 · Experiments

In Section 3.6, the syntax of experiment definitions in χ has been defined. It follows from the description that an experiment is just a process or system instantiation. Therefore, a natural translation of

$$\mathsf{xper} = [\![\ P(3.5)\ ]\!]$$

would be the instantiation of the corresponding χ_σ process definition. For convenience, we call the χ_σ process definition $\mathcal{T}(P)$. This gives the following translation: $\mathcal{T}(P)(3.5)$. However, the process $\mathcal{T}(P)(3.5)$ does not have the same behaviour as $P(3.5)$. Firstly, if $P(3.5)$ has communication statements, s, these statements wait until both parties are ready to communicate. Therefore, in $P(3.5)$, send and receive statements cannot be executed in isolation. However, $\mathcal{T}(P(3.5)$ can execute isolated send and receive processes, since this is not forbidden by the SOS rules of χ_σ. A solution is to encapsulate the send and receive actions. So, if we define $A = \{sa(m, c) \mid m \in \mathit{Channel}, c \in \mathit{Value}\} \cup \{ra(m, x) \mid m \in \mathit{Channel}, x \in \mathit{Var}\}$, the translation is

$$\partial_A(\mathcal{T}(P)(3.5)).$$

Secondly, χ has maximal progress built in. Therefore, if $P(3.5)$ is at a point where at least one non-delay statement can be executed (possibly a communication), the process (or system) cannot delay. However, χ_σ does not have maximal progress built in, but provides a maximal progress operator to enforce this behaviour. Therefore, in addition to the encapsulation operator, a maximal progress operator is added to the translation of χ experiments, with A as defined above:

$$\pi(\partial_A(\mathcal{T}(P)(3.5))).$$

6.7 · Discussion

In this chapter, a translation scheme from χ into χ_σ has been defined. The fact that the scheme is straightforward and maintains the 'look and feel' of χ specifications, shows that χ_σ is a formal version of χ. This is an important observation with respect to one of our starting points: formalise an engineering language (see Section 1.3).

The presented scheme is linear in the size of the χ specification. Recently, a tool has been developed to perform the translation automatically.

Unfortunately, since χ_σ lacks several features of χ (see previous chapter), some χ statements cannot be translated.

Tool support · 7

This chapter describes the tools we used to validate the formal definitions of Chapters 2 and 4 and to perform case studies with χ_σ. These tools are collectively called the χ_σ *engine* and have been implemented in Python [24, 130] and Maude. The current version of the χ_σ engine is a prototype. It can be used to analyze (small) production systems, as we show in Chapter 8. We do not describe the implementation of the χ_σ engine but confine ourselves to its theoretical foundation and its architecture. At a global level, the χ_σ engine consists of the following components.

Front end The task of the front end of the χ_σ engine is to parse χ_σ specifications and to build an internal representation of these specifications.

SOS checker The SOS checker verifies whether a given termination or transition formula can be derived according to the SOS of χ_σ.

SOS computer The SOS computer calculates the SOS of a χ_σ process; it computes transition and termination options.

Back end The back end of the χ_σ engine provides functionality to simulate specifications and to generate state spaces of specifications.

The χ_σ engine is integrated with third party tools to minimize state spaces, check equivalences of state spaces, and visualize state spaces. In addition, we describe an experiment environment that provides a uniform interface to the functionality of the χ_σ engine and the third party tools.

This chapter is organised as follows. First of all, we discuss the goals and requirements of the χ_σ engine in Section 7.1. The tools are then discussed from front end in Section 7.2, to back end in Section 7.5. In Section 7.6, we describe several extensions of χ_σ that have been implemented. Third party tools are then discussed in Section 7.7, and in Section 7.8, we discuss an experiment environment to perform case studies. This chapter is concluded by a discussion in Section 7.9.

7.1 · Goals and requirements

The goals of the χ_σ engine are validation of the formal semantics of χ_σ, illustration of constructs and concepts, and automatic analysis of χ_σ models. These goals are discussed below.

Validation of the formal semantics of χ_σ means investigating the consequences of the formal definitions of χ_σ. This can be done in at least two ways. Firstly, by proving lemmas and theorems general results can be established that give insight in the formal semantics of χ_σ. Secondly, by executing actual χ_σ specifications in the χ_σ engine, their behaviour can be analysed. If the behaviour exposed is undesired, we can conclude that either the implementation of the χ_σ engine is not correct with respect to the formal definitions, or the formal definitions are wrong.

The χ_σ engine can be used to illustrate constructs and concepts of χ_σ. For instance, the interaction between different process operators can be visualized by a graphical representation of the state space (see Chapter 8). In addition, the χ_σ engine can illustrate concepts like nondeterminism, time factorisation, deadlock, and specification-implementation correctness.

By automatic analysis of a χ_σ specification, we obtain information about that specification. For instance, the χ_σ engine can be used to show that it is deadlock-free. In theory, this can be done by hand. However, only if the specification is small, is this approach practical. Consequently, to analyse χ_σ specifications of real-life production systems, tool support is indispensable.

Based on the three goals mentioned above, we define the following requirements.

1. The χ_σ engine should correctly implement the formal semantics of χ_σ.

   A correctness proof for the χ_σ engine is outside the scope of the research described in this thesis. In order to satisfy the first requirement, we subjected the χ_σ engine to tests. In order to increase the credibility of test results, we implemented two versions of the formal semantics: the SOS checker and the SOS computer.

2. The χ_σ engine should handle all χ_σ processes.

   As we explain in Section 7.4, we cannot implement a tool that computes the complete process graph of a χ_σ process, because it is infinite. Therefore, we present some lemmas by which we can reduce the process graph such that

is becomes finite while preserving important properties. The χ_σ engine can compute these finite process graphs.

3. The output of the χ_σ engine should be in terms of the formal semantics of χ_σ: processes, terminations, action transitions, and delay transitions.

4. The χ_σ engine should have a practical user-interface to analyse both small and large χ_σ specifications.

5. It should be relatively easy to integrate existing tools with the χ_σ engine.

   The χ_σ engine should not be a stand alone application, but it should cooperate with existing applications. As mentioned above, the χ_σ engine is integrated with different kinds of tools to analyse and visualize state spaces.

7.2 · Front end

The front end of the χ_σ engine transforms a (textual) χ_σ specification into an internal representation. During the transformation, the front end performs syntactic and semantic checks on the specification. Syntactic checking is based on the χ_σ grammar for textual input. Semantic checking is currently limited to formal-actual parameter checks of process instantiations and should be extended to type checking.

7.3 · SOS checker

The SOS checker verifies termination formulas and transition formulas (see Definition 4.3). For instance, given a termination formula $\langle\, p, \sigma \,\rangle\!\downarrow$ or a transition formula $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, the SOS checker verifies if this formula can be derived according to the deduction rules of processes and the MEL specifications of the data types. As such, the SOS checker is an automatic theorem prover for theorems of the forms $\langle\, p, \sigma \,\rangle\!\downarrow$, $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, and $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$.

The functionality of the SOS checker is useful in several ways. First of all, early in the formalisation process of χ_σ, the SOS checker greatly aided us in checking hand-made derivations of terminations and transitions. Since even for relatively small processes, generating these derivations is error-prone, a correctness check by an automated tool like the SOS checker is very valuable. Furthermore, if the result of the SOS checker did not comply with our intuition about the SOS of χ_σ, the

tool usually indicated where the mismatch occurred. Based on this information, we then had to decide whether to change the definitions or to adapt our intuition.

Another useful application of the SOS checker is to verify terminations and transitions computed by other χ_σ tools. For instance, the χ_σ simulator and model checker were both validated using the SOS checker; each termination and transition computed by these tools, can be verified by the SOS checker. During the development of the simulator and model checker, several implementation errors were discovered by the SOS checker. Furthermore, if new χ_σ tools are developed, the SOS checker can be a valuable testing device.

The main requirement of the SOS checker is that it is a correct implementation of χ_σ. In order to validate this requirement, we chose to write an implementation of the SOS checker that more or less literally resembles the formal definitions of the data types and the deduction rules.

We implemented the SOS checker for χ_σ in Maude. Recall that Maude was also used to test the data type specifications given in Chapter 2. The implementation of the χ_σ data types in Maude is straightforward, since they are defined in MEL and Maude supports MEL. The implementation of the deduction rules in Maude can be done in several ways.

First of all, deduction rules can be implemented by unconditional equations in Maude. For instance, by defining a boolean valued operator $\langle \_, \_ \rangle \xrightarrow{\ \_\ } \langle \_, \_ \rangle \to \mathrm{bool}$ in MEL (and similar operators for delay transitions and termination relations), a transition $\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle$ is a boolean term. Equations can be used to define the semantics of this operator. The main problem with this approach is that sometimes several equations apply. Consider, for instance, the action rules for the sequential composition (Rules 18 and 19, page 77):

$$\frac{\langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle}{\langle p \ ; \ q, \sigma \rangle \xrightarrow{a} \langle p' \ ; \ q, \sigma' \rangle} \quad , \qquad \frac{\langle p, \sigma \rangle\!\downarrow, \ \langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle}{\langle p \ ; \ q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle} \quad .$$

If these rules are translated into equations, as in

$$\begin{array}{lcl} \langle p \ ; \ q, \sigma \rangle \xrightarrow{a} \langle p' \ ; \ q, \sigma' \rangle & = & \langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle, \\ \langle p \ ; \ q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle & = & \langle p, \sigma \rangle\!\downarrow \wedge \langle q, \sigma \rangle \xrightarrow{a} \langle q', \sigma' \rangle, \end{array}$$

it is possible that several equations apply. If the first equation applies, then also the second equation applies. The problem with this translation is that selection of an equation is based solely on the syntactic form of the conclusion of a deduction rule.

Therefore, Maude's built-in strategy to apply equations decides which equation will actually be used. Consequently, even if the hypotheses of a deduction rule cannot be satisfied, it is still possible to select this deduction rule. This makes it useless for the implementation of the SOS checker.

Secondly, as described in [57], operational semantics can be defined using Maude's rewriting logic [137]. That is, each deduction rule is implemented by one or more *rewrite rules* in Maude. Notice that rewrite rules are not equations; Maude supports both MEL (equations) and Rewriting Logic (rewrite rules). If we apply this approach to χ_σ, each transition $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$ is translated into a rewrite rule of the form $\langle\, p, \sigma \,\rangle \Rightarrow \{a\}\langle\, p', \sigma' \,\rangle$, where $\{a\}\langle\, p', \sigma' \,\rangle$ denotes the resulting process p' and stack σ' after executing action a. Unfortunately, this approach has similar problems as the approach described above.

Based on these observations, we developed another approach. In this approach, the hypotheses of a deduction rule are regarded as conditions of the equation. In addition, the conclusion is the left-hand side of the equation, and the right-hand side of the equation is simply the boolean constant *true*. For example, the deduction rule for the action transitions of the sequential composition operator are translated into

$$\begin{array}{lcll} \langle\, p \;;\; q, \sigma \,\rangle \xrightarrow{a} \langle\, p' \;;\; q, \sigma' \,\rangle & = & \mathit{true} & \Leftarrow \quad \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle, \\ \langle\, p \;;\; q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle & = & \mathit{true} & \Leftarrow \quad \langle\, p, \sigma \,\rangle\!\downarrow \wedge \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle. \end{array}$$

Since Maude can apply an equation only if its condition is true, this approach first evaluates the hypotheses of a deduction rule and Maude can evaluate the conclusion only if the hypotheses are satisfied. Note that it is still possible that several equations apply. However, in that case it does not matter which equation is chosen, since all hypotheses are satisfied and the result will always be *true*. It is clear that the implementation of the SOS checker based on this conditional equations approach, can straightforwardly be validated. Therefore, the main requirement of the SOS checker, a correct implementation of χ_σ, is met by this approach.

7.4 · SOS computer

The SOS computer computes the semantics of a χ_σ process as defined in Chapter 4. That is, given a process, the SOS computer determines whether or not the process can terminate successfully and it determines the set of transitions the process

can execute. Formally, the SOS computer should implement the function sc of Definition 7.1.

Definition 7.1 · (SOS computer) *The SOS computer function* $sc : P \to \mathcal{P}(P \times Stack) \cup \mathcal{P}(P \times Stack \times (Action \cup R_{>0}) \times P \times Stack)$ *is defined by*

$$\begin{aligned} sc(p) = &\{(p,\sigma) \mid \langle\, p,\sigma \,\rangle\!\downarrow, \sigma \in Stack\} \\ &\cup \{(p,\sigma,a,p'\sigma') \mid \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p',\sigma' \,\rangle, \sigma,\sigma' \in Stack, p' \in P, a \in Action\} \\ &\cup \{(p,\sigma,d,p'\sigma') \mid \langle\, p,\sigma \,\rangle \overset{d}{\longmapsto} \langle\, p',\sigma' \,\rangle, \sigma,\sigma' \in Stack, p' \in P, d \in R_{>0}\}. \end{aligned}$$

The elements $(p,\sigma) \in sc(p)$ are the *terminations* of p, the elements $(p,\sigma,a,p',\sigma') \in sc(p)$ are the *action transitions* of p, and the elements $(p,\sigma,d,p'\sigma') \in sc(p)$ are the *delay transitions* of p.

In general, the result of $sc(p)$ is an infinite set. For instance, the set $sc(\varepsilon) = \{\langle\, \varepsilon,\sigma \,\rangle \mid \sigma \in Stack\}$ and since there are infinitely many stacks, this is an infinite set. Another cause of infinity of $sc(p)$ is the number of delay transitions a process can have: if a process can delay for d time units, then, for every $d' < d$, it can delay d' time units too (see Lemma 4.46). Since the time domain of χ_σ is the set of positive real numbers, there are infinitely many of such d' transitions. Consequently, there is no terminating algorithm to compute $sc(p)$ completely. However, it is possible to compute a finite subset of *representatives* of $sc(p)$ such that many interesting properties of p can be checked formally on the set of representatives.

The reduction of the set $sc(p)$ is based on two observations. Firstly, it suffices to compute only the terminations and the transitions for the empty stack. The intuition behind this is that if a termination or a transition is possible under the empty stack, it is possible under any stack. This property of the SOS is exemplified by Lemma 7.2. Notice that the visible identifiers in a non-empty stack can be added to p by combining these identifiers and p in a state operator. Moreover, the number of terminations and action transitions is finite for any stack. Lemma 7.3 shows that for a given stack, each process has at most one termination. In addition, Lemma 7.4 shows that the set of action transitions of a process is finite for a given stack. Based on this lemma, we define a function that computes the action transitions for a process and a stack (Definition 7.5). Lemma 7.6 shows the correctness of this function with respect to the SOS of χ_σ.

Secondly, in many situations, the order of actions performed by a process is more important than the exact time at which the actions occur. Lemma 7.7, shows that for a (large) subset of χ_σ processes, it suffices to compute only one delay transi-

tion. This subset includes the translation of χ processes as defined in Chapter 6. Definition 7.8 defines a function that computes a unique delay value for a process given a stack. If the result of this functions is 0, the process cannot delay under the given stack. The correctness of this function is shown in Lemma 7.9. In Definition 7.10, this unique delay value is used to compute a set of delay transitions for a given process and stack. This set contains at most one delay transition and is empty if and only if the process cannot delay (under the given stack).

Finally, the results of the observations are put together and a function sc' is defined that computes a finite number of terminations, action transitions, and delay transitions for a process (Definition 7.12). This function is implemented in the χ_σ engine.

As mentioned above, we start by showing that if a process has a termination or a transition for the empty stack, then it has terminations and transitions for any other stack.

Lemma 7.2 · *Let p and p' be processes, a be an action, and d be a positive real number, then*

$$\begin{aligned}
&\langle\, p, \lambda_\sigma \,\rangle\!\downarrow \Rightarrow \forall\sigma : \langle\, p, \sigma \,\rangle\!\downarrow,\\
&\langle\, p, \lambda_\sigma \,\rangle \xrightarrow{a} \langle\, p', \lambda_\sigma \,\rangle \Rightarrow \forall\sigma : \exists\sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p, \sigma' \,\rangle,\\
&\langle\, p, \lambda_\sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \lambda_\sigma \,\rangle \Rightarrow \forall\sigma : \exists\sigma' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p, \sigma' \,\rangle.
\end{aligned}$$

Proof · (Lemma 7.2) The proof is based on the observation that the role of stacks in deduction rules, is to provide values for programming variables occurring in χ_σ processes. Furthermore, by close inspection of the deduction rules, we see that whenever an expression is used in a rule (Table 4.1), it evaluates to a constant value: $\sigma(e) = c$. Suppose $\sigma = \lambda_\sigma$ and we have $\sigma(e) = c$ for some expression e and value c. Let σ' be another stack, then we can make the following computation: $\sigma'(e) = \sigma'(\lambda_\sigma(e)) = \sigma'(\sigma(e)) = \sigma'(c) = c$. Therefore, if an expression evaluates to a constant value under the empty stack, it evaluates to this value under any stack. Consequently, if a termination or transition concerning the empty stack can be derived, it can be derived for arbitrary stacks. □

Of course, the implication symbols of Lemma 7.2 cannot be reversed; if there is a termination or a transition for a nonempty stack, it does not mean there are terminations or transitions for the empty stack.

The notation $|S|$, where S is a set, denotes the number of elements in S. If S is an infinite set, $|S| = \infty$, otherwise, $|S| \in N$.

Lemma 7.3 · *Let $p \in P$ be a process and $\sigma \in Stack$ be a stack. Then $|\{(p,\sigma) \in sc(p,\sigma) \mid \langle\, p,\sigma \,\rangle\!\downarrow\}| \leq 1$.*

Proof · Lemma 7.3 This lemma follows immediately from the fact that termination relations are given by stacks (Section 4.2). □

Next, we show that the set of action transitions of a process is finite for a given stack. To that extent, we use $Action(p,\sigma)$ to denote the set $\{a \in Action \mid \exists p' \in P, \sigma' \in Stack : \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p',\sigma' \,\rangle\}$, for any process p and stack σ.

Lemma 7.4 · (Finite number of actions) *Let $p \in P$ be a process and $\sigma \in Stack$ be a stack, then $|Action(p,\sigma)| \in N$.*

One might expect this lemma to follow immediately from results of theory on syntactic formats for TSSs. For instance, for TSSs in the *De Simone* format [187], it is known that if certain conditions are met, the corresponding LTS is *computable*, meaning that there exists an algorithm that computes for each state of the LTS the finite set of outgoing transitions. However, the TSS defined by the deduction rules of χ_σ do not adhere to the De Simone format. This can be seen easily, since one of the restrictions of the De Simone format is that deduction rules do not have negative hypotheses.

Proof · (Lemma 7.4) Suppose p is a process and σ is a stack. We use structural induction on p to prove that $|Action(p,\sigma)| \in N$. The basis of the induction proof consists of cases for all atomic processes and the inductive step consists of cases for all compound processes. During the proofs of the inductive step, we can use induction hypothesis (IH-7.4).

> *Let $p_0 \in P$ be an argument of p and $\sigma \in Stack$ be a stack, then $|Action(p_0,\sigma)| \in N$.* (IH-7.4)

Basis We distinguish the following cases.

$p \equiv \delta$: Since there are no action rules for δ, it is clear that $Action(\delta,\sigma) = \emptyset$. Consequently, we have $|Action(p,\sigma)| = 0 \in N$.

$p \equiv \varepsilon$: The proof is similar to the previous case.

$p \equiv \mathsf{skip}$: There is only one action rule for skip, Rule 3. According to Rule 3, we have $\langle\, \mathsf{skip},\sigma' \,\rangle \xrightarrow{\tau} \langle\, \varepsilon,\sigma' \,\rangle$ for any stack σ'. Therefore, for stack σ, there is only one action transition, $\langle\, \mathsf{skip},\sigma \,\rangle \xrightarrow{\tau} \langle\, \varepsilon,\sigma \,\rangle$. So, we have $|Action(p,\sigma)| = 1 \in N$.

$p \equiv x := e$: There is only one action rule for $x := e$, Rule 4. This rule only applies if $\exists c \in \mathit{Value} : \sigma(e) = c$. So, suppose $\sigma(e) = c \in \mathit{Value}$. Then, according to Rule 4, we have $\langle\, x := e, \sigma \,\rangle \xrightarrow{aa(x,c)} \langle\, \varepsilon, \sigma[c/x] \,\rangle$, which is exactly one transition. So, we have $|\mathit{Action}(p, \sigma)| = 1 \in N$. Remains to consider the possibility that $\sigma(e) \notin \mathit{Value}$. In that case, no rule applies and $|\mathit{Action}(p, \sigma)| = 0 \in N$.

$p \equiv m \,!\, e$: The proof is similar to the previous case.

$p \equiv m \,?\, x$: The proof is similar to the previous case.

$p \equiv \Delta e$: The proof is similar to the first case.

Inductive Step We distinguish the following cases.

$p \equiv b :\rightarrow p_0$: Rule 11 is the action rule for the guard operator. This rule applies if $\sigma(b) = \mathit{true}$. Therefore, we make a case distinction based on the value of $\sigma(b)$. So, suppose $\sigma(b) = \mathit{true}$. Then, for each action a, process p_0', and σ', with $\langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle$, we can use Rule 11 to derive $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle$. Therefore, $\mathit{Action}(p_0, \sigma) = \mathit{Action}(p, \sigma)$. Using induction hypothesis (IH-7.4), we know that $|\mathit{Action}(p_0, \sigma)| \in N$. Consequently, we get $|\mathit{Action}(p, \sigma)| \in N$. Remains to consider the case $\sigma(b) \neq \mathit{true}$. Now, Rule 11 does not apply and we have $|\mathit{Action}(p, \sigma)| = 0 \in N$.

$p \equiv p_0 \,[\!]\, p_1$: Rule 14 is the action rule for the alternative composition operator. Therefore, if this rule does not apply to p and σ, we have $|\mathit{Action}(p, \sigma)| = 0 \in N$. If this rule applies, there is a process p', an action a, and a stack σ', such that $\langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$ or $\langle\, p_1, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$. Therefore, for each action transition of p_0 or p_1, we obtain an action transition of p. So, we have $|\mathit{Action}(p, \sigma)| \leq |\mathit{Action}(p_0, \sigma)| + |\mathit{Action}(p_1, \sigma)|$. Note the '$\leq$' operator, since it is possible that two action transitions of p_0 and p_1, respectively, result in the same action transition for p. Using induction hypothesis (IH-7.4), we obtain $|\mathit{Action}(p_0, \sigma)| \in N$ and $|\mathit{Action}(p_1, \sigma)| \in N$. So, we derive $|\mathit{Action}(p, \sigma)| \in N$.

$p \equiv p_0 \;;\; p_1$: The action transition rules for sequential composition are Rules 18 and 19. If none of these rules applies to p and σ, we have $|\mathit{Action}(p, \sigma)| = 0 \in N$. If at least one of these rules applies, that is, if we have $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, we can derive that p_0 or p_1 can execute action a. Consequently, we find $|\mathit{Action}(p, \sigma)| \leq |\mathit{Action}(p_0, \sigma)| + |\mathit{Action}(p_1, \sigma)|$. Using induction hypothesis (IH-7.4), we obtain $|\mathit{Action}(p_0, \sigma)| \in N$ and $|\mathit{Action}(p_1, \sigma)| \in N$. So, we derive $|\mathit{Action}(p, \sigma)| \in N$.

$p \equiv {p_0}^*$: Rule 25 is the action rule for the repetition operator. If this rule does not apply to p and σ, we have $|Action(p, \sigma)| = 0 \in N$. If this rule applies, we find an action transition for p for every action transition of p_0. So, we can derive that $|Action(p_0, \sigma)| = |Action(p, \sigma)|$. Using induction hypothesis (IH-7.4), we obtain $|Action(p_0, \sigma)| \in N$. Consequently, we have $|Action(p, \sigma)| \in N$.

$p \equiv p_0 \parallel p_1$: Rules 28 and 29 are the action transition rules for the parallel composition operator. If none of these rules applies to p and σ, we have $|Action(p, \sigma)| = 0 \in N$. If Rule 28 applies, we find an action transition of p for every action transition of p_0 and for every action transition of p_1. If Rule 29 applies, we find an action transition of p for every two matching send and receive action transitions of p_0 and p_1, respectively. Therefore, we obtain $|Action(p, \sigma)| \leq (2 \cdot |Action(p_0, \sigma)|) + (2 \cdot |Action(p_1, \sigma)|)$. Using induction hypothesis (IH-7.4), we obtain $|Action(p_0, \sigma)| \in N$ and $|Action(p_1, \sigma)| \in N$. So, we derive $|Action(p, \sigma)| \in N$.

$p \equiv [\![s \mid p_0]\!]$: Rule 33 is the action rule for the state operator. If this rule does not apply to p and σ, we have $|Action(p, \sigma)| = 0 \in N$. If this rule applies, we see that for every action transition of p_0 and stack $s :: \sigma$, we obtain an action transition of p. So, $|Action(p, \sigma)| = |Action(p_0, s :: \sigma)|$. Using induction hypothesis (IH-7.4), we obtain $|Action(p_0, s :: \sigma)| \in N$, which gives us $|Action(p, \sigma)| \in N$.

$p \equiv \partial_A(p_0)$: Rule 36 is the action rule for the encapsulation operator. If this rule does not apply to p and σ, we have $|Action(p, \sigma)| = 0 \in N$. If this rule applies, we see that for every action transition of p_0, we obtain an action transition of p. Consequently, $|Action(p, \sigma)| \leq |Action(p_0, \sigma)|$. Using induction hypothesis (IH-7.4), we obtain $|Action(p_0, \sigma)| \in N$. Therefore, we have $|Action(p, \sigma)| \in N$.

$p \equiv \pi(p_0)$: Rule 39 is the action rule for the maximal progress operator. If this rule is does not apply to p and σ, we have $|Action(p, \sigma)| = 0 \in N$. If this rule applies, we see that for every action transition of p_0, we obtain an action transition of p. Consequently, $|Action(p, \sigma)| = |Action(p_0, \sigma)|$. Using induction hypothesis (IH-7.4), we obtain $|Action(p_0, \sigma)| \in N$. Therefore, we have $|Action(p, \sigma)| \in N$.

$p \equiv \tau_A(p_0)$: Rules 42 and 43 are the action transition rules for the abstraction operator. If none of these rules applies to p and σ, we have $|Action(p, \sigma)| = 0 \in N$. If a rule applies, we see that for every action transition of p_0, we obtain an

action transition of p (possibly the action is replaced by the τ action). Therefore, $|Action(p,\sigma)| \leq |Action(p_0,\sigma)|$. Using induction hypothesis (IH-7.4), we obtain $|Action(p_0,\sigma)| \in N$ and, consequently, $|Action(p,\sigma)| \in N$. □

Lemma 7.4 guarantees that for a given process and stack, the number of action transitions is finite. However, it does not provide an algorithm to compute this set of action transitions. Definition 7.5 defines a function ac that computes this set.

Definition 7.5 · (Action computation) *The function* $ac : P \times Stack \to \mathcal{P}(P \times Stack \times Action \times P \times Stack)$ *is defined in Table 7.1.*

The correctness of the ac function is established in Lemma 7.6. This lemma says that for all processes p and stacks σ the elements of $ac(p,\sigma)$ coincide with the action transitions of p under σ.

Lemma 7.6 · (Correctness of ac) *Let* $p, p' \in P$ *be processes, let* $a \in Action$ *be an action, and let* $\sigma, \sigma' \in Stack$ *be stacks, then*

$$(p,\sigma,a,p',\sigma') \in ac(p,\sigma) \Leftrightarrow \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p',\sigma' \,\rangle.$$

Proof · (Lemma 7.6) We prove the lemma by structural induction on process p. The basis consists of the cases where p is an atomic process. The inductive step consists of the cases where p is a compound process. During the proof of the inductive step, we can use induction hypothesis (IH 7.6).

> *Let* $p_0, p' \in P$ *be processes such that* p_0 *is a process argument of* p, *let* $a \in Action$ *be an action, and let* $\sigma, \sigma' \in Stack$ *be stacks, then* $(p,\sigma,a,p',\sigma') \in ac(p,\sigma) \Leftrightarrow \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p',\sigma' \,\rangle$. (IH 7.6)

Basis We distinguish the following cases.

$p \equiv \delta$: The proof is trivial.

$p \equiv \varepsilon$: The proof is trivial.

$p \equiv \mathsf{skip}$: Rule 3 is the action rule for skip. Therefore, there is exactly one transition $\langle\, \mathsf{skip},\sigma \,\rangle \xrightarrow{\tau} \langle\, \varepsilon,\sigma \,\rangle$ for a given stack σ. According to Definition 7.5, we also have $ac(\mathsf{skip},\sigma) = \{(\mathsf{skip},\sigma,\tau,\varepsilon,\sigma)\}$.

$$
\begin{array}{rcl}
ac(\delta,\sigma) & = & \emptyset \\
ac(\varepsilon,\sigma) & = & \emptyset \\
ac(\mathsf{skip},\sigma) & = & \{(\mathsf{skip},\sigma,\tau,\varepsilon,\sigma)\} \\
ac(x := e,\sigma) & = & \{(x := e,\sigma,aa(x,c),\varepsilon,\sigma[c/x]) \wedge \sigma(e) = c \in \mathit{Value}\} \\
ac(m\,!\,e,\sigma) & = & \{(m\,!\,e,\sigma,sa(m,c),\varepsilon,\sigma[c/m]) \wedge \sigma(e) = c \in \mathit{Value}\} \\
ac(m\,?\,x,\sigma) & = & \{(m\,?\,x,\sigma,ra(m,x),\varepsilon,\sigma[c/x]) \wedge \sigma(e) = c \in \mathit{Value}\} \\
ac(\Delta e,\sigma) & = & \emptyset \\
ac(e :\rightarrow p,\sigma) & = & \{(e :\rightarrow p,\sigma,a,p',\sigma') \mid \in^a_{\sigma,\sigma'}(p,p') \wedge \sigma(e) = \mathit{true}\} \\
ac(p \,[\!]\, q,\sigma) & = & \{(p \,[\!]\, q,\sigma,a,p',\sigma') \mid \in^a_{\sigma,\sigma'}(p,p') \vee \in^a_{\sigma,\sigma'}(q,p')\} \\
ac(p\ ;\ q,\sigma) & = & \{(p\ ;\ q,\sigma,a,p'\ ;\ q,\sigma') \mid \in^a_{\sigma,\sigma'}(p,p')\} \\
 & \cup & \{(p\ ;\ q,\sigma,a,q',\sigma') \mid \in^a_{\sigma,\sigma'}(q,q') \wedge \langle\, p,\sigma\,\rangle\!\downarrow\} \\
ac(p^*,\sigma) & = & \{(p^*,\sigma,a,p'\ ;\ p^*,\sigma') \mid \in^a_{\sigma,\sigma'}(p,p')\} \\
ac(p \parallel q,\sigma) & = & \{(p \parallel q,\sigma,a,p' \parallel q,\sigma') \mid \in^a_{\sigma,\sigma'}(p,p')\} \\
 & \cup & \{(p \parallel q,\sigma,a,p \parallel q',\sigma') \mid \in^a_{\sigma,\sigma'}(q,q')\} \\
 & \cup & \{\ (p \parallel q,\sigma,a,p' \parallel q',\sigma'') \\
 & & \quad \mid a \equiv ca(m,x,c) \\
 & & \quad \wedge \quad \in^{sa(m,c)}_{\sigma,\sigma'}(p,p') \wedge \in^{ra(m,x)}_{\sigma',\sigma''}(q,q') \\
 & & \qquad \vee \in^{ra(m,x)}_{\sigma',\sigma''}(p,p') \wedge \in^{sa(m,c)}_{\sigma,\sigma'}(q,q') \\
 & & \quad \} \\
ac([\![\, s \mid p \,]\!],\sigma) & = & \{([\![\, s \mid p \,]\!],\sigma,a,[\![\, s' \mid p' \,]\!],\sigma') \mid \in^a_{s::\sigma,s'::\sigma'}(p,p')\} \\
ac(\partial_A(p),\sigma) & = & \{(\partial_A(p),\sigma,a,\partial_A(p'),\sigma') \mid \in^a_{\sigma,\sigma'}(p,p') \wedge a \notin A\} \\
ac(\pi(p),\sigma) & = & \{(\pi(p),\sigma,a,\pi(p'),\sigma') \mid \in^a_{\sigma,\sigma'}(p,p')\} \\
ac(\tau_A(p),\sigma) & = & \{(\tau_A(p),\sigma,a,\tau_A(p'),\sigma') \mid \in^a_{\sigma,\sigma'}(p,p') \wedge a \notin A\} \\
 & \cup & \{(\tau_A(p),\sigma,\tau,\tau_A(p'),\sigma') \mid \in^a_{\sigma,\sigma'}(p,p') \wedge a \in A\}
\end{array}
$$

$$\text{where } \in^a_{\sigma,\sigma'}(p,p') \equiv (p,\sigma,a,p',\sigma') \in ac(p,\sigma)$$

Table 7.1 · Definition of function ac.

$p \equiv x := e$: Rule 4 is the action rule for $x{:=}e$. Therefore, there is exactly one transition $\langle\, x := e,\sigma\,\rangle \xrightarrow{aa(x,c)} \langle\, \varepsilon,\sigma'\,\rangle$, where $\sigma(e) = c \in \mathit{Value}$, for a given stack σ. Furthermore, the rule gives us $\sigma' = \sigma[c/x]$. According to Definition 7.5, we also have $ac(x := e,\sigma) = \{(x := e,\sigma,\tau,\varepsilon,\sigma[c/x])\}$.

$p \equiv m\,!\,e$: The proof is similar to the previous case.

$p \equiv m\,?\,x$: The proof is similar to the previous case.

$p \equiv \Delta e$: The proof is trivial.

Inductive step We consider the following cases.

$p \equiv e :\rightarrow p_0$: Rule 11 is the action rule for the guard operator. So, we have

$$\begin{aligned}
& \langle\, e :\rightarrow p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Rule 11}\} \\
& \sigma(e) = \mathit{true} \wedge \langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Induction hypothesis (IH 7.6)}\} \\
& \sigma(e) = \mathit{true} \wedge (p_0, \sigma, a, p', \sigma') \in ac(p_0, \sigma) \\
\Leftrightarrow\ & \{\text{Definition 7.5 and } p \equiv e :\rightarrow p_0\} \\
& (e :\rightarrow p_0, \sigma, a, p', \sigma') \in ac(p, \sigma).
\end{aligned}$$

$p \equiv p_0 \parallel p_1$: Rule 14 is the action rule for the alternative composition operator. So, we have

$$\begin{aligned}
& \langle\, p_0 \parallel p_1, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Rule 14 (two possibilities)}\} \\
& \langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \vee \langle\, p_1, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Induction hypothesis (IH 7.6) two times}\} \\
& (p_0, \sigma, a, p', \sigma') \in ac(p_0, \sigma) \vee \\
& (p_1, \sigma, a, p', \sigma') \in ac(p_1, \sigma) \\
\Leftrightarrow\ & \{\text{Definition 7.5 and } p \equiv p_0 \parallel p_1\} \\
& (p, \sigma, a, p', \sigma') \in ac(p, \sigma).
\end{aligned}$$

$p \equiv p_0 \ ;\ p_1$: Rules 18 and 19 are the action transition rules for the sequential composition operator. So, we have

$$\begin{aligned}
& \langle\, p_0 \ ;\ p_1, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Rule 18 or Rule 19}\} \\
& \langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle \wedge p' \equiv p_0' \ ;\ q \vee \langle\, p_1, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \wedge \langle\, p, \sigma \,\rangle\!\downarrow \\
\Leftrightarrow\ & \{\text{Induction hypothesis (IH 7.6) two times}\} \\
& (p_0, \sigma, a, p_0', \sigma) \in ac(p_0, \sigma) \wedge p' \equiv p_0' \ ;\ q \vee \\
& (p_1, \sigma, a, p', \sigma') \in ac(p_1, \sigma) \wedge \langle\, p, \sigma \,\rangle\!\downarrow \\
\Leftrightarrow\ & \{\text{Definiton 7.5 and } p \equiv p_0 \ ;\ p_1\} \\
& (p, \sigma, a, p', \sigma') \in ac(p, \sigma).
\end{aligned}$$

$p \equiv {p_0}^*$: Rule 25 is the action rule for the repetition operator. So, we have

$$\begin{array}{ll} & \langle\, {p_0}^*, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\ \Leftrightarrow & \{\text{Rule 25}\} \\ & \langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle \wedge p' \equiv p_0' \ ;\ p_0 \\ \Leftrightarrow & \{\text{Induction hypothesis (IH 7.6)}\} \\ & (p_0, \sigma, a, p_0', \sigma') \in ac(p_0, \sigma) \wedge p' \equiv p_0' \ ;\ p_0 \\ \Leftrightarrow & \{\text{Definition 7.5}\} \\ & (p, \sigma, a, p', \sigma' \,\rangle). \end{array}$$

$p \equiv p_0 \parallel p_1$: Rules 28 and 29 are the action transition rules for the parallel composition operator. So, we have

$$\begin{array}{ll} & \langle\, p_0 \parallel p_1, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\ \Leftrightarrow & \{\text{Rules 28 (two possibilities) and 29 (two possibilities)}\} \\ & (\langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle \wedge p' \equiv p_0' \parallel p_1) \vee (\langle\, p_1, \sigma \,\rangle \xrightarrow{a} \langle\, p_1', \sigma' \,\rangle \wedge p' \equiv p_0 \parallel p_1') \\ \vee & (\langle\, p_0, \sigma \,\rangle \xrightarrow{sa(m,c)} \langle\, p_0', \sigma_0' \,\rangle \wedge \langle\, p_1, \sigma_0' \,\rangle \xrightarrow{ra(m,x)} \langle\, p_1', \sigma' \,\rangle \\ & \wedge p' \equiv p_0' \parallel p_1' \wedge a \equiv ca(m, x, c)) \\ \vee & (\langle\, p_0, \sigma_1' \,\rangle \xrightarrow{ra(m,x)} \langle\, p_0', \sigma' \,\rangle \wedge \langle\, p_1, \sigma \,\rangle \xrightarrow{sa(m,c)} \langle\, p_1', \sigma_1' \,\rangle \\ & \wedge p' \equiv p_0' \parallel p_1' \wedge a \equiv ca(m, x, c)) \\ \Leftrightarrow & \{\text{Induction hypothesis (IH 7.6) multiple times}\} \\ & ((p_0, \sigma, a, p_0', \sigma') \in ac(p_0, \sigma) \wedge p' \equiv p_0' \parallel p_1) \vee ((p_1, \sigma, a, p_1', \sigma') \in ac(p_1, \sigma) \\ & \wedge p' \equiv p_0 \parallel p_1') \\ \vee & ((p_0, \sigma, sa(m, c), p_0', \sigma_0') \in ac(p_0, \sigma) \wedge (p_1, \sigma_0', ra(m, x), p_1', \sigma') \in ac(p_1, \sigma_0') \\ & \wedge p' \equiv p_0' \parallel p_1' \wedge a \equiv ca(m, x, c)) \\ \vee & ((p_0, \sigma_1', ra(m, x), p_0', \sigma') \in ac(p_0, \sigma_1') \wedge (p_1, \sigma, sa(m, c), p_1', \sigma_1') \in ac(p_1, \sigma) \\ & \wedge p' \equiv p_0' \parallel p_1' \wedge a \equiv ca(m, x, c)) \\ \Leftrightarrow & \{\text{Definition 7.5 and } p \equiv p_0 \parallel p_1\} \\ & (p, \sigma, a, p', \sigma') \in ac(p, \sigma). \end{array}$$

$p \equiv [\![\, s \mid p_0 \,]\!]$: Rule 33 is the action rule for the state operator. So, we have

$$\begin{array}{ll} & \langle\, [\![\, s \mid p_0 \,]\!], \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\ \Leftrightarrow & \{\text{Rule 33}\} \\ & \langle\, p_0, s :: \sigma \,\rangle \xrightarrow{a} \langle\, p_0', s' :: \sigma_0' \,\rangle \wedge p' \equiv [\![\, s' \mid p_0' \,]\!] \wedge \sigma' \equiv s' :: \sigma_0' \\ \Leftrightarrow & \{\text{Induction hypothesis (IH 7.6)}\} \\ & (p_0, s :: \sigma, a, p_0', s' :: \sigma_0') \in ac(p_0, s :: \sigma) \wedge p' \equiv [\![\, s' \mid p_0' \,]\!] \wedge \sigma' \equiv s' :: \sigma_0' \\ \Leftrightarrow & \{\text{Definition 7.5 and } p \equiv [\![\, s \mid p_0 \,]\!]\} \\ & (p, \sigma, a, p', \sigma') \in ac(p, \sigma). \end{array}$$

$p \equiv \partial_A(p_0)$: Rule 36 is the action rule for the encapsulation operator. So, we have

$$
\begin{aligned}
& \langle\, \partial_A(p_0), \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Rule 36}\} \\
& \langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle \wedge p' \equiv \partial_A(p_0') \wedge a \notin A \\
\Leftrightarrow\ & \{\text{Induction hypothesis (IH 7.6)}\} \\
& (p_0, \sigma, a, p_0', \sigma') \in ac(p_0, \sigma) \wedge p' \equiv \partial_A(p_0') \wedge a \notin A \\
\Leftrightarrow\ & \{\text{Definition 7.5 and } p \equiv \partial_A(p_0)\} \\
& (p, \sigma, a, p', \sigma') \in ac(p, \sigma).
\end{aligned}
$$

$p \equiv \pi(p_0)$: Rule 39 is the action rule for the maximal progress operator. So, we have

$$
\begin{aligned}
& \langle\, \pi(p_0), \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Rule 39}\} \\
& \langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle \wedge p' \equiv \pi(p_0') \\
\Leftrightarrow\ & \{\text{Induction hypothesis (IH 7.6)}\} \\
& (p_0, \sigma, a, p_0', \sigma') \in ac(p_0, \sigma) \wedge p' \equiv \pi(p_0') \\
\Leftrightarrow\ & \{\text{Definition 7.5 and } p \equiv \pi(p_0)\} \\
& (p, \sigma, a, p', \sigma') \in ac(p, \sigma).
\end{aligned}
$$

$p \equiv \tau_A(p_0)$: Rules 42 and 43 are the action transition rules for the abstraction operator. So, we have

$$
\begin{aligned}
& \langle\, \tau_A(p_0), \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \\
\Leftrightarrow\ & \{\text{Rules 42 and 43}\} \\
& (\langle\, p_0, \sigma \,\rangle \xrightarrow{a} \langle\, p_0', \sigma' \,\rangle \wedge p' \equiv \tau_A(p_0') \wedge a \notin A) \\
\vee\ & (\langle\, p_0, \sigma \,\rangle \xrightarrow{a'} \langle\, p_0', \sigma' \,\rangle \wedge p' \equiv \tau_A(p_0') \wedge a \equiv \tau \wedge a' \in A) \\
\Leftrightarrow\ & \{\text{Induction hypothesis (IH 7.6) two times}\} \\
& ((p_0, \sigma, a, p_0', \sigma') \in ac(p_0, \sigma) \wedge p' \equiv \tau_A(p_0') \wedge a \notin A) \\
\vee\ & ((p_0, \sigma, a', p_0', \sigma') \in ac(p_0, \sigma) \wedge p' \equiv \tau_A(p_0') \wedge a \equiv \tau \wedge a' \in A) \\
\Leftrightarrow\ & \{\text{Definition 7.5 and } p \equiv \tau_A(p_0)\} \\
& (p, \sigma, a, p', \sigma') \in ac(p, \sigma).
\end{aligned}
$$

□

So far, we have focused on the action transitions of processes and we have shown that for a given stack, the number of action transitions is finite. Furthermore, we defined a function ac that computes this set of action transitions. Next, we focus on reducing the number of delay transitions of a process.

Lemma 7.7 can be used to reduce computation of the set of delay transitions of a maximal progress process to computation of a single delay transition, without losing the possibility to verify interesting properties.

Lemma 7.7 · *Let p be a maximal progress process: $p \equiv \pi(p')$ for some process p'. Suppose $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle r, \sigma'' \rangle$ and $\langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle q, \sigma' \rangle$ and $d' < d$. Then $\langle q, \sigma' \rangle \overset{d-d'}{\longmapsto} \langle r, \sigma'' \rangle$ and $\langle q, \sigma' \rangle \not\downarrow$ and $\langle q, \sigma' \rangle \not\rightarrow$.*

Proof · (Lemma 7.7) Suppose $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle r, \sigma'' \rangle$ and $\langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle q, \sigma' \rangle$. According to Lemma 4.53 (Time confluence), we immediately have $\langle q, \sigma' \rangle \overset{d-d'}{\longmapsto} \langle r, \sigma'' \rangle$. Similarly, we also have $\langle q, \sigma' \rangle \not\downarrow$. So, we only have to prove that $\langle q, \sigma' \rangle \not\rightarrow$. Suppose there exist a, q_a and a σ_a, such that $\langle q, \sigma' \rangle \xrightarrow{a} \langle q_a, \sigma_a \rangle$. Using Lemma 4.55 (Preservation of action transitions), this would mean that there exists a p_a and a σ_a such that $\langle p, \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_a \rangle$. However, since $p \equiv \pi(p')$ and p can perform delay transitions, this is a contradiction. Therefore, the assumption that there exist a, q_a and a σ_a, such that $\langle q, \sigma' \rangle \xrightarrow{a} \langle q_a, \sigma_a \rangle$ is invalid. Consequently, we have $\langle q, \sigma' \rangle \not\rightarrow$. □

Lemma 7.7 has reduced the problem of computing all possible delay transitions to computing only one delay transition. Note that the reduction applies to maximal progress processes only. Definition 7.8 defines a function D_{d_0} that computes a delay value for a χ_σ process and a stack. The parameter d_0 is a positive real number. It is a default delay value that is used for processes that can delay arbitrary long, like send and receive processes. Furthermore, Lemma 7.9 shows that for any process p and any stack σ, if $D_{d_0}(p, \sigma) = 0$, then $\langle p, \sigma \rangle \not\mapsto$. Also, if $D_{d_0}(p, \sigma) = d \neq 0$, then there exists a delay transition $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle$ for some p' and σ'.

Definition 7.8 · (Unique delay value) *Let $d_0 \in R_{>0}$ be an arbitrary positive real number. The function $D_{d_0} : P \times Stack \to R_{\geq 0}$ is defined in Table 7.2.*

Lemma 7.9 · (Valid unique delay value) *Let $p \in P$ be a process, $\sigma \in Stack$ be a stack, and $d_0 \in R_{>0}$ be a positive real number. Then, if $D_{d_0}(p, \sigma) = 0$ then $\langle p, \sigma \rangle \not\mapsto$, and if $D_{d_0}(p, \sigma) > 0$ then $\exists p' \in P, \sigma' \in Stack : \langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p', \sigma' \rangle$.*

Proof · (Lemma 7.9) We prove this lemma by structural induction on process p. The basis of the induction consists of the cases where p is an atomic process. The inductive step consists of the cases where p is a compound process. In the proof of the inductive step, we can use induction hypothesis (IH-7.9).

$$
\begin{array}{rcll}
D_{d_0}(\delta,\sigma) & = & 0 & \\
D_{d_0}(\varepsilon,\sigma) & = & 0 & \\
D_{d_0}(\mathsf{skip},\sigma) & = & 0 & \\
D_{d_0}(x := e,\sigma) & = & 0 & \\
D_{d_0}(m\,!\,e,\sigma) & = & d_0 & \\
D_{d_0}(m\,?\,x,\sigma) & = & d_0 & \\
D_{d_0}(\Delta e,\sigma) & = & \sigma(e) & \text{if } \sigma(e) \in \mathit{Value} \wedge \sigma(e) > 0 \\
 & = & 0 & \text{otherwise} \\
D_{d_0}(e :\rightarrow p,\sigma) & = & D_{d_0}(p,\sigma) & \text{if } \sigma(e) = \mathit{true} \\
 & = & 0 & \text{otherwise} \\
D_{d_0}(p \,[\!]\, q,\sigma) & = & D_{d_0}(p,\sigma) & \text{if } \langle\, p,\sigma \,\rangle \mapsto \wedge\, \langle\, q,\sigma \,\rangle \not\mapsto \\
 & = & D_{d_0}(q,\sigma) & \text{if } \langle\, q,\sigma \,\rangle \mapsto \wedge\, \langle\, p,\sigma \,\rangle \not\mapsto \\
 & = & \mathit{min}(D_{d_0}(p,\sigma), D_{d_0}(q,\sigma)) & \text{otherwise} \\
D_{d_0}(p \;;\; q,\sigma) & = & D_{d_0}(p,\sigma) & \text{if } \langle\, p,\sigma \,\rangle \mapsto \wedge\, (\langle\, p,\sigma \,\rangle \not\downarrow \vee \langle\, q,\sigma \,\rangle \not\mapsto) \\
 & = & D_{d_0}(q,\sigma) & \text{if } \langle\, p,\sigma \,\rangle \downarrow \wedge\, \langle\, p,\sigma \,\rangle \not\mapsto \wedge\, \langle\, q,\sigma \,\rangle \mapsto \\
 & = & \mathit{min}(D_{d_0}(p,\sigma), D_{d_0}(q,\sigma)) & \text{otherwise} \\
D_{d_0}(p^*,\sigma) & = & D_{d_0}(p,\sigma) & \\
D_{d_0}(p \,\|\, q,\sigma) & = & D_{d_0}(p,\sigma) & \text{if } \langle\, p,\sigma \,\rangle \mapsto \wedge\, \langle\, q,\sigma \,\rangle \downarrow \wedge\, \langle\, q,\sigma \,\rangle \not\mapsto \\
 & = & D_{d_0}(q,\sigma) & \text{if } \langle\, p,\sigma \,\rangle \downarrow \wedge\, \langle\, p,\sigma \,\rangle \not\mapsto \wedge\, \langle\, q,\sigma \,\rangle \mapsto \\
 & = & \mathit{min}(D_{d_0}(p,\sigma), D_{d_0}(q,\sigma)) & \text{otherwise} \\
D_{d_0}([\![\, s \mid p \,]\!],\sigma) & = & D_{d_0}(p, s :: \sigma) & \\
D_{d_0}(\partial_A(p),\sigma) & = & D_{d_0}(p,\sigma) & \\
D_{d_0}(\pi(p),\sigma) & = & D_{d_0}(p,\sigma) & \text{if } \langle\, p,\sigma \,\rangle \not\rightarrow \\
 & = & 0 & \text{otherwise} \\
D_{d_0}(\tau_A(p),\sigma) & = & D_{d_0}(p,\sigma) &
\end{array}
$$

where $\langle\, p,\sigma \,\rangle \rightarrow$ denotes $\exists a, p', \sigma' : \langle\, p,\sigma \,\rangle \xrightarrow{a} \langle\, p'\sigma' \,\rangle$, and
$\langle\, p,\sigma \,\rangle \mapsto$ denotes $\exists d, p', \sigma' : \langle\, p,\sigma \,\rangle \xmapsto{d} \langle\, p'\sigma' \,\rangle$

Table 7.2 · Definition of the function D_{d_0}.

Let $p_0 \in P$ be a process argument of p, $\sigma \in Stack$ be a stack, and $d_0 \in R_{>0}$ be a positive real number. Then, if $D_{d_0}(p_0, \sigma) = 0$ then $\langle\, p_0, \sigma \,\rangle \mapsto$, and if $D_{d_0}(p_0, \sigma) > 0$ then $\exists p_0' \in P, \sigma' \in Stack$: $\langle\, p_0, \sigma \,\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma' \,\rangle$. (IH-7.9)

Basis We distinguish the following cases.

$p \equiv \delta$: The proof is trivial, because $D_{d_0}(p, \sigma) = 0$ and $\langle\, \delta, \sigma \,\rangle \mapsto$.

$p \equiv \varepsilon$: The proof is trivial, because $D_{d_0}(p, \sigma) = 0$ and $\langle\, \varepsilon, \sigma \,\rangle \mapsto$.

$p \equiv \mathsf{skip}$: The proof is trivial, because $D_{d_0}(p, \sigma) = 0$ and $\langle\, \mathsf{skip}, \sigma \,\rangle \mapsto$.

$p \equiv x := e$: The proof is trivial, because $D_{d_0}(p, \sigma) = 0$ and $\langle\, x := e, \sigma \,\rangle \mapsto$.

$p \equiv m\,!\,e$: According to Definition 7.8 we have $D_{d_0}(p, \sigma) = d_0$. So, we have to prove $\langle\, p, \sigma \,\rangle \xmapsto{d_0} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. This immediately follows from Rule 7.

$p \equiv m\,?\,x$: The proof is similar to the proof of the previous case.

$p \equiv \Delta e$: According to Definition 7.8, we can distinguish the following cases.

- $\sigma(e) \in Value \wedge \sigma(e) > 0$: According to Definition 7.8 we have $D_{d_0}(p, \sigma) = \sigma(e)$. So, we have to prove $\langle\, p, \sigma \,\rangle \xmapsto{\sigma(e)} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. This immediately follows from Rule 9.
- $\neg(\sigma(e) \in Value \wedge \sigma(e) > 0)$: According to Definition 7.8 we have $D_{d_0}(p, \sigma) = 0$. So, we have to prove $\langle\, p, \sigma \,\rangle \mapsto$. Since $\neg(\sigma(e) \in Value \wedge \sigma(e) > 0)$, we can derive that there is no d such that $0 < d \leq \sigma(e)$. Consequently, the only delay rule for the delay process (Rule 9) does not apply.

Inductive step We distinguish the following cases.

$p \equiv e :\rightarrow p_0$: According to Definition 7.8, we distinguish the following cases.

- $\sigma(e) = true$: According to Definition 7.8 we have $D_{d_0}(p, \sigma) = D_{d_0}(p_0, \sigma)$. We distinguish two cases: $D_{d_0}(p_0, \sigma) = 0$ and $D_{d_0}(p_0, \sigma) > 0$. In the first case, we have to show that $\langle\, p, \sigma \,\rangle \mapsto$. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle\, p_0, \sigma \,\rangle \mapsto$. Consequently, the only delay rule for the guard operator (Rule 12) does not apply. In the second case, we have to

show that $\langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p', \sigma' \rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle p_0, \sigma \rangle \overset{D_{d_0}(p_0,\sigma)}{\longmapsto} \langle p'_0, \sigma'_0 \rangle$ for some p'_0 and σ'_0. So, since $\sigma(e) = \mathit{true}$, we can use Rule 12 to obtain $\langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p'_0, \sigma'_0 \rangle$.

$\neg(\sigma(e) = \mathit{true})$: According to Definition 7.8 we have $D_{d_0}(p, \sigma) = 0$. So, we have to prove $\langle p, \sigma \rangle \not\mapsto$. Since $\neg(\sigma(e) = \mathit{true})$, the only delay rule for the guard operator (Rule 12) does not apply. Consequently, $\langle p, \sigma \rangle \not\mapsto$.

$p \equiv p_0 \parallel p_1$: According to Definition 7.8, we distinguish three cases.

$\langle p_0, \sigma \rangle \mapsto \wedge \langle p_1, \sigma \rangle \not\mapsto$: Definition gives 7.8 $D_{d_0}(p, \sigma) = D_{d_0}(p_0, \sigma)$. Suppose $D_{d_0}(p, \sigma) = 0$. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle p_0, \sigma \rangle \not\mapsto$. Since we have $\langle p_0, \sigma \rangle \mapsto$, this is a contradiction. Consequently, $D_{d_0}(p, \sigma) > 0$. This means we have to show that $\langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p', \sigma' \rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle p_0, \sigma \rangle \overset{D_{d_0}(p_0,\sigma)}{\longmapsto} \langle p'_0, \sigma'_0 \rangle$ for some p'_0 and σ'_0. So, since $\langle p_0, \sigma \rangle \overset{D_{d_0}(p_0,\sigma)}{\longmapsto} \langle p'_0, \sigma'_0 \rangle$ and $\langle p_1, \sigma \rangle \not\mapsto$, we can use Rule 15 to obtain $\langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p'_0, \sigma'_0 \rangle$.

$\langle p_0, \sigma \rangle \not\mapsto \wedge \langle p_1, \sigma \rangle \mapsto$: The proof is similar to the previous case.

otherwise: Definition 7.8 gives $D_{d_0}(p, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$. We distinguish two cases: $D_{d_0}(p, \sigma) = 0$ and $D_{d_0}(p, \sigma) > 0$. In the first case, from $D_{d_0}(p, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$, it follows that $D_{d_0}(p_0, \sigma) = 0$ and $D_{d_0}(p_1, \sigma) = 0$. By using induction hypothesis (IH-7.9) on both p_0 and p_1, we obtain $\langle p_0, \sigma \rangle \not\mapsto$ and $\langle p_1, \sigma \rangle \not\mapsto$. Therefore, none of the delay rules for the alternative composition operator (Rules 15 and 16) apply. Consequently, $\langle p, \sigma \rangle \not\mapsto$. In the second case, we have to show that $\langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p', \sigma' \rangle$ for some p' and σ'. Since we have $D_{d_0}(p, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$, it follows that $D_{d_0}(p_0, \sigma) > 0$ and $D_{d_0}(p_1, \sigma) > 0$. By using induction hypothesis (IH-7.9) on both p_0 and p_1, we obtain $\langle p_0, \sigma \rangle \overset{D_{d_0}(p_0,\sigma)}{\longmapsto} \langle p'_0, \sigma'_0 \rangle$ and $\langle p_1, \sigma \rangle \overset{D_{d_0}(p_1,\sigma)}{\longmapsto} \langle p'_1, \sigma'_1 \rangle$ for some p'_0, p'_1, σ'_0, and σ'_1. According to Lemma 4.49, we have $\sigma = \sigma'_0 = \sigma'_1$. We distinguish the following cases: $D_{d_0}(p_0, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$ and $D_{d_0}(p_0, \sigma) \neq \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$. In the first case, Lemmas 4.46 and 4.49 give $\langle p_1, \sigma \rangle \overset{D_{d_0}(p_0,\sigma)}{\longmapsto} \langle p''_1, \sigma \rangle$ for some p''_1. From the facts that $\langle p_0, \sigma \rangle \overset{D_{d_0}(p_0,\sigma)}{\longmapsto} \langle p'_0, \sigma \rangle$ and $\langle p_1, \sigma \rangle \overset{D_{d_0}(p_0,\sigma)}{\longmapsto} \langle p''_1, \sigma \rangle$, we know that Rule 16 applies and therefore we can derive $\langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p'_0 \parallel p''_1, \sigma \rangle$. The second case is similar.

$p \equiv p_0 \ ; \ p_1$: According to Definition 7.8, we distinguish three cases.

- $\langle\, p_0, \sigma \,\rangle\!\mapsto \wedge\, (\langle\, p_0, \sigma \,\rangle\!\not\downarrow \vee \langle\, p_1, \sigma \,\rangle\!\not\mapsto)$: According to Definition 7.8 we know that $D_{d_0}(p, \sigma) = D_{d_0}(p_0, \sigma)$. Suppose $D_{d_0}(p, \sigma) = 0$. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle\, p_0, \sigma \,\rangle\!\not\mapsto$. Since we have $\langle\, p_0, \sigma \,\rangle\!\mapsto$, this is a contradiction. Consequently, $D_{d_0}(p, \sigma) > 0$. This means we have to show that $\langle\, p, \sigma \,\rangle \xrightarrow{D_{d_0}(p,\sigma)} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle\, p_0, \sigma \,\rangle \xrightarrow{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$ for some p_0' and σ_0'. So, since $\langle\, p_0, \sigma \,\rangle \xrightarrow{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$ and $\langle\, p_0, \sigma \,\rangle\!\not\downarrow \vee \langle\, p_1, \sigma \,\rangle\!\not\mapsto$, we can either use Rule 20 or 21 to obtain $\langle\, p, \sigma \,\rangle \xrightarrow{D_{d_0}(p,\sigma)} \langle\, p_0' \ ; \ p_1, \sigma_0' \,\rangle$.
- $\langle\, p_0, \sigma \,\rangle\!\downarrow \wedge \langle\, p_0, \sigma \,\rangle\!\not\mapsto \wedge \langle\, p_1, \sigma \,\rangle\!\mapsto$: According to Definition 7.8 we know that $D_{d_0}(p, \sigma) = D_{d_0}(p_1, \sigma)$. Suppose $D_{d_0}(p, \sigma) = 0$. By using induction hypothesis (IH-7.9) on p_1, we obtain $\langle\, p_1, \sigma \,\rangle\!\not\mapsto$. Since we have $\langle\, p_1, \sigma \,\rangle\!\mapsto$, this is a contradiction. Consequently, $D_{d_0}(p, \sigma) > 0$. This means we have to show that $\langle\, p, \sigma \,\rangle \xrightarrow{D_{d_0}(p,\sigma)} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_1, we obtain $\langle\, p_1, \sigma \,\rangle \xrightarrow{D_{d_0}(p_1,\sigma)} \langle\, p_1', \sigma_1' \,\rangle$ for some p_1' and σ_1'. So, since $\langle\, p_1, \sigma \,\rangle \xrightarrow{D_{d_0}(p_1,\sigma)} \langle\, p_1', \sigma_1' \,\rangle$ and $\langle\, p_0, \sigma \,\rangle\!\downarrow \wedge \langle\, p_0, \sigma \,\rangle\!\not\mapsto$, we can use Rule 22 to obtain $\langle\, p, \sigma \,\rangle \xrightarrow{D_{d_0}(p,\sigma)} \langle\, p_1', \sigma_1' \,\rangle$.
- *otherwise*: The negations of the conditions in the previous cases give us: if $\langle\, p_0, \sigma \,\rangle\!\mapsto$ then $\langle\, p_0, \sigma \,\rangle\!\downarrow$. According to Definition 7.8 we have $D_{d_0}(p, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$. We distinguish two cases: $D_{d_0}(p, \sigma) = 0$ and $D_{d_0}(p, \sigma) > 0$. In the first case, it follows from the fact $D_{d_0}(p, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$, that we have $D_{d_0}(p_0, \sigma) = 0$ and $D_{d_0}(p_1, \sigma) = 0$. By using induction hypothesis (IH-7.9) on both p_0 and p_1, we obtain $\langle\, p_0, \sigma \,\rangle\!\not\mapsto$ and $\langle\, p_1, \sigma \,\rangle\!\not\mapsto$. Therefore, none of the delay rules for the sequential composition operator (Rules 20 through 23) apply. Consequently, $\langle\, p, \sigma \,\rangle\!\not\mapsto$. In the second case, we have to show that $\langle\, p, \sigma \,\rangle \xrightarrow{D_{d_0}(p,\sigma)} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. From $D_{d_0}(p, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$, it follows that $D_{d_0}(p_0, \sigma) > 0$ and $D_{d_0}(p_1, \sigma) > 0$. By using induction hypothesis (IH-7.9) on both p_0 and p_1, we obtain $\langle\, p_0, \sigma \,\rangle \xrightarrow{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$ (so, we have $\langle\, p_0, \sigma \,\rangle\!\downarrow$) and $\langle\, p_1, \sigma \,\rangle \xrightarrow{D_{d_0}(p_1,\sigma)} \langle\, p_1', \sigma_1' \,\rangle$ for some p_0', p_1', σ_0', and σ_1'. According to Lemma 4.49, we have $\sigma = \sigma_0' = \sigma_1'$. We distinguish the following cases: $D_{d_0}(p_0, \sigma) = \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$ and $D_{d_0}(p_0, \sigma) \neq \mathit{min}(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$. In the first case, according to Lemmas 4.46 and 4.49 we have $\langle\, p_1, \sigma \,\rangle \xrightarrow{D_{d_0}(p_0,\sigma)} \langle\, p_1'', \sigma \,\rangle$ for some p_1''. So, since $\langle\, p_0, \sigma \,\rangle \xrightarrow{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma \,\rangle$, $\langle\, p_1, \sigma \,\rangle \xrightarrow{D_{d_0}(p_0,\sigma)} \langle\, p_1'', \sigma \,\rangle$, and $\langle\, p_0, \sigma \,\rangle\!\downarrow$,

we can use Rule 23 to obtain $\langle\, p, \sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p_0' \;;\; p_1 \,\|\, p_1'', \sigma \,\rangle$. In the second case, according to Lemmas 4.46 and 4.49, we have $\langle\, p_0, \sigma \,\rangle \xmapsto{D_{d_0}(p_1,\sigma)} \langle\, p_0'', \sigma \,\rangle$ for some p_0''. So, since $\langle\, p_0, \sigma \,\rangle \xmapsto{D_{d_0}(p_1,\sigma)} \langle\, p_0'', \sigma \,\rangle$, $\langle\, p_1, \sigma \,\rangle \xmapsto{D_{d_0}(p_1,\sigma)} \langle\, p_1', \sigma \,\rangle$, and $\langle\, p_0, \sigma \,\rangle\!\downarrow$, we can use Rule 23 to obtain $\langle\, p, \sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p_0'' \;;\; p_1 \,\|\, p_1', \sigma \,\rangle$.

$p \equiv {p_0}^*$: According to Definition 7.8, we have $D_{d_0}(p, \sigma) = D_{d_0}(p_0, \sigma)$. We distinguish two cases: $D_{d_0}(p_0, \sigma) = 0$ and $D_{d_0}(p_0, \sigma) > 0$. In the first case, we have to show that $\langle\, p, \sigma \,\rangle\!\not\mapsto$. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle\, p_0, \sigma \,\rangle\!\not\mapsto$. Consequently, the only delay rule for the repetition operator (Rule 26) does not apply. In the second case, we have to show that $\langle\, p, \sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle\, p_0, \sigma \,\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$ for some p_0' and σ_0'. So, we can use Rule 26 to obtain $\langle\, p, \sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p_0' \;;\; p^*, \sigma_0' \,\rangle$.

$p \equiv p_0 \,\|\, p_1$: According to Definition 7.8, we distinguish three cases.

$\langle\, p_0, \sigma \,\rangle\!\mapsto \wedge \langle\, p_1, \sigma \,\rangle\!\downarrow \wedge \langle\, p_1, \sigma \,\rangle\!\not\mapsto$: Definition 7.8 gives $D_{d_0}(p, \sigma) = D_{d_0}(p_0, \sigma)$. Suppose $D_{d_0}(p, \sigma) = 0$. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle\, p_0, \sigma \,\rangle\!\not\mapsto$. Since we have $\langle\, p_0, \sigma \,\rangle\!\mapsto$, this is a contradiction. Consequently, $D_{d_0}(p, \sigma) > 0$. This means we have to show that $\langle\, p, \sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle\, p_0, \sigma \,\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$ for some p_0' and σ_0'. Since we know that $\langle\, p_0, \sigma \,\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$, $\langle\, p_1, \sigma \,\rangle\!\downarrow$, and $a\langle\, p_1, \sigma \,\rangle\!\not\mapsto$ we can use Rule 30 to obtain $\langle\, p, \sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$.

$\langle\, p_0, \sigma \,\rangle\!\downarrow \wedge \langle\, p_0, \sigma \,\rangle\!\not\mapsto \wedge \langle\, p_1, \sigma \,\rangle\!\mapsto$: The proof is similar to the previous case.

otherwise: Definition 7.8 gives $D_{d_0}(p, \sigma) = min(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$. We distinguish two cases: $D_{d_0}(p, \sigma) = 0$ and $D_{d_0}(p, \sigma) > 0$. In the first case, from $D_{d_0}(p, \sigma) = min(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$, it follows that $D_{d_0}(p_0, \sigma) = 0$ and $D_{d_0}(p_1, \sigma) = 0$. By using induction hypothesis (IH-7.9) on both p_0 and p_1, we obtain $\langle\, p_0, \sigma \,\rangle\!\not\mapsto$ and $\langle\, p_1, \sigma \,\rangle\!\not\mapsto$. Therefore, none of the delay rules for the parallel composition operator (Rules 30 and 31) apply. Consequently, $\langle\, p, \sigma \,\rangle\!\not\mapsto$. In the second case, we have to show that $\langle\, p, \sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p', \sigma' \,\rangle$ for some p' and σ'. From $D_{d_0}(p, \sigma) = min(D_{d_0}(p_0, \sigma), D_{d_0}(p_1, \sigma))$, it follows that $D_{d_0}(p_0, \sigma) > 0$ and $D_{d_0}(p_1, \sigma) > 0$. By using induction hypothesis (IH-7.9) on both p_0 and p_1, we obtain $\langle\, p_0, \sigma \,\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle\, p_0', \sigma_0' \,\rangle$ and $\langle\, p_1, \sigma \,\rangle \xmapsto{D_{d_0}(p_1,\sigma)} \langle\, p_1', \sigma_1' \,\rangle$ for some p_0', p_1', σ_0', and σ_1'. According to Lemma 4.49, we have $\sigma = \sigma_0' = \sigma_1'$. We distinguish the fol-

lowing cases: $D_{d_0}(p_0,\sigma) = min(D_{d_0}(p_0,\sigma), D_{d_0}(p_1,\sigma))$ and $D_{d_0}(p_0,\sigma) \neq min(D_{d_0}(p_0,\sigma), D_{d_0}(p_1,\sigma))$. In the first case, according to Lemmas 4.46 and 4.49, we have $\langle p_1,\sigma\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle p_1'',\sigma\rangle$ for some p_1''. So, since we have $\langle p_0,\sigma\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle p_0',\sigma\rangle$ and $\langle p_1,\sigma\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle p_1'',\sigma\rangle$, we can use Rule 31 to obtain $\langle p,\sigma\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle p_0' \,[\![\, p_1'',\sigma\rangle$. The second case is similar.

$p \equiv [\![\, s \mid p_0 \,]\!]$: According to Definition 7.8, we have $D_{d_0}(p,\sigma) = D_{d_0}(p_0, s :: \sigma)$. We distinguish two cases: $D_{d_0}(p_0, s :: \sigma) = 0$ and $D_{d_0}(p_0, s :: \sigma) > 0$. In the first case, we have to show that $\langle p,\sigma\rangle \mapsto$. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle p_0, s :: \sigma\rangle \mapsto$. Consequently, the only delay rule for the state operator (Rule 34) does not apply. In the second case, we have to show that $\langle p,\sigma\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle p',\sigma'\rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle p_0, s :: \sigma\rangle \xmapsto{D_{d_0}(p_0,s::\sigma)} \langle p_0',\sigma_0'\rangle$ for some p_0', and σ_0'. According to Lemma 4.49, we have $s :: \sigma = \sigma_0'$. Therefore, we have $\langle p_0, s :: \sigma\rangle \xmapsto{D_{d_0}(p_0,s::\sigma)} \langle p_0', s :: \sigma\rangle$. So, we can use Rule 34 to obtain $\langle p,\sigma\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle [\![\, s \mid p_0' \,]\!], \sigma\rangle$.

$p \equiv \partial_A(p_0)$: The proof is similar to the proof of the case $p \equiv {p_0}^*$.

$p \equiv \pi(p_0)$: According to Definition 7.8, we distinguish the following cases.

- $\langle p_0,\sigma\rangle \not\rightarrow$: According to Definition 7.8 we have $D_{d_0}(p,\sigma) = D_{d_0}(p_0,\sigma)$. We distinguish two cases: $D_{d_0}(p_0,\sigma) = 0$ and $D_{d_0}(p_0,\sigma) > 0$. In the first case, we have to show that $\langle p,\sigma\rangle \mapsto$. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle p_0,\sigma\rangle \mapsto$. Consequently, the only delay rule for the maximal progress operator (Rule 40) does not apply. In the second case, we have to show that $\langle p,\sigma\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle p',\sigma'\rangle$ for some p' and σ'. By using induction hypothesis (IH-7.9) on p_0, we obtain $\langle p_0,\sigma\rangle \xmapsto{D_{d_0}(p_0,\sigma)} \langle p_0',\sigma_0'\rangle$ for some p_0' and σ_0'. So, since $\langle p_0,\sigma\rangle \not\rightarrow$, we can use Rule 40 to obtain $\langle p,\sigma\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle \pi(p_0'),\sigma_0'\rangle$.
- $\neg\langle p_0,\sigma\rangle \not\rightarrow$: According to Definition 7.8 we have $D_{d_0}(p,\sigma) = 0$. So, we have to prove $\langle p,\sigma\rangle \mapsto$. Since $\neg\langle p_0,\sigma\rangle \not\rightarrow$, we can derive that $\langle p_0,\sigma\rangle \xrightarrow{a} \langle p_0',\sigma_0'\rangle$ for some a, p_0', and σ_0'. Therefore, the only delay rule for the maximal progress operator (Rule 40) does not apply. Consequently, $\langle p,\sigma\rangle \mapsto$.

$p \equiv \tau_A(p_0)$: The proof is similar to the proof of the case $p \equiv {p_0}^*$. □

The next step is to define a function dc_{d_0} that takes a process and a stack and computes a delay transition using D_{d_0}. The dc_{d_0} function is defined in Definition 7.10.

Definition 7.10 · (Delay computation) *Let $d_0 \in R_{>0}$ be a positive real number. The function $dc_{d_0} : P \times Stack \to (P \times Stack \times R_{>0} \times P \times Stack)$ is defined by*

$$\begin{aligned} dc_{d_0}(p,\sigma) = \{\ & (p,\sigma,D_{d_0}(p,\sigma),p',\sigma') \\ & \mid D_{d_0}(p,\sigma) > 0 \wedge (p,\sigma,D_{d_0}(p,\sigma),p',\sigma') \in sc(p,\sigma) \\ & \}. \end{aligned}$$

Note that, due to time determinism (Lemma 4.48), the set $dc_{d_0}(p,\sigma)$ for a process p and stack σ contains at most one element. This is formalised in the next lemma.

Lemma 7.11 · *Let $p \in P$ be a process and $\sigma \in Stack$ be a stack, then $|dc_{d_0}(p,\sigma)| \leq 1$.*

Proof · (Lemma 7.11) Suppose that $|dc_{d_0}(p,\sigma)| > 1$. Then according to Definition 7.10 there are p', p'', σ', and σ'', such that $(p,\sigma,D_{d_0}(p,\sigma),p',\sigma')$ and $(p,\sigma,D_{d_0}(p,\sigma),p'',\sigma'')$ and $p' \neq p''$ or $\sigma' \neq \sigma''$. Since $(p,\sigma,D_{d_0}(p,\sigma),p',\sigma') \in sc(p,\sigma)$ we can use Definition 7.1, to obtain $\langle\, p,\sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p',\sigma' \,\rangle$. Similarly, we obtain $\langle\, p,\sigma \,\rangle \xmapsto{D_{d_0}(p,\sigma)} \langle\, p'',\sigma'' \,\rangle$. According to Lemma 4.48, we have $p' = p''$ and $\sigma' = \sigma''$. This means we have a contradiction. □

We are now able to define a *finite SOS computer* function sc' that determines a finite number of terminations and transitions of a process. The terminations and transitions form a subset of the terminations and transitions computed by the function sc of Definition 7.1: only terminations and transition under the empty stack are considered, and at most one delay transition is computed. Since we use function D_{d_0} from Definition 7.8, the finite SOS computer function is parameterized by a positive real number d_0 and denoted by sc'_{d_0}. This function is given in Definition 7.12.

Definition 7.12 · (Finite SOS computer) *Let d_0 be a positive real number. The finite SOS computer function $sc'_{d_0} : P \to \mathcal{P}(P \times Stack) \cup \mathcal{P}(P \times Stack \times (Action \cup R_{>0}) \times P \times Stack)$ is defined by*

$$sc'_{d_0}(p) = \{(p,\lambda_\sigma) \mid (p,\lambda_\sigma) \in sc(p)\} \cup ac(p,\lambda_\sigma) \cup dc_{d_0}(p,\lambda_\sigma).$$

The fact that $sc'_{d_0}(p)$ is finite, results from the following properties of $sc'_{d_0}(p)$:

- it contains at most one termination (Lemma 7.3),

- it contains finitely many action transitions (Lemma 7.4),
- it contains at most one delay transition (Lemma 7.11).

Recall that the function sc defines the terminations, the action transitions, and the delay transitions of processes. That is, for a given process, it defines a χ_σ-LTS. In general, this LTS is infinite. The function sc'_{d_0} also defines a χ_σ-LTS for a given process. This LTS is finite. Therefore, we can define a reduction for any process p that reduces $sc(p)$ into $sc'_{d_0}(p)$. This reduction has two important applications. Firstly, since the state space of a χ_σ specification is finite (after reduction), effective implementations to compute this state space are possible. The χ_σ engine is an example of such an implementation. Secondly, existing tools that manipulate finite LTSs can be used to manipulate χ_σ processes. For instance, using tools from the Formal Methods, bisimulation checks on χ_σ processes can be performed. These tools implement various kinds of bisimulations on LTSs, like (untimed) strong bisimulation and (untimed) branching bisimulation. Consequently, although we did not formalise timed branching-bisimulation on χ_σ processes, these tools can be used to perform specification-implementation checks (see Chapter 8).

In order to the verify a property using the reduced χ_σ-LTS, it is important to know whether the property is preserved under the reduction. Based on Lemma 7.7, we can conclude that the reduction does preserve many properties of maximal progress processes (proofs are omitted), see Table 7.3. Notice that all translations of χ processes are maximal progress processes (Chapter 6).

property	preserved
$\pi(p)$ has deadlock	yes
$\pi(p)$ has live lock	yes
a always occurs before b	yes
a occurs at most d time before b	yes
$\pi(p)$ can/cannot delay	yes
$\pi(p)$ can/cannot delay d time	no
$\pi(p)$ can do different delays	no

Table 7.3 · Influence of reduction on example properties.

This concludes the discussion about the SOS computer. We have shown that even though in general the set of terminations and transitions of a process is infinite, it

is possible to define a relevant, finite subset that can be computed effectively. The SOS computer determines this finite subset. In the next section, we will describe the components of the χ_σ engine that employ the SOS computer.

7.5 · Back end

The back end of the χ_σ engine provides functionality to simulate χ_σ specifications and functionality to compute the state space of χ_σ specifications. A state space is in fact a χ_σ-LTS (see Definition 4.2).

The simulator functionality consists of:

- instantiating a simulator with a χ_σ process,
- computing the set of terminations and transitions according to the SOS computer function sc' (see Definition 7.12),
- selecting a transition for execution,
- executing a transition, and
- executing arbitrary many, randomly chosen, successive transitions.

Selection of a transition can be controlled by the user. This enables user-directed simulations in which the effect of certain sequences of transitions can be studied. By executing (randomly chosen) successive transitions, the χ_σ engine provides simulation functionality comparable to that of previous χ engines [2].

The state space computation functionality consists of:

- instantiating a state space generator with a χ_σ process,
- computing the state space for the current process,
- checking for deadlock states,
- writing the state space in a format suitable for model checking, and
- writing the state space in a format suitable for visualisation.

The algorithm *ComputeStateSpace* for state space computation is given in code listing below. It has one parameter p: the χ_σ process for which the state space

should be computed. Each state of the state space is a χ_σ process. The algorithm explores the state space in a breadth-first-search approach. Variable Q is a queue of unexplored states; it is initialized to the singleton list containing process p. Variable S is the state space that is computed; it is initialized to the empty state space (no states, no terminations, and no transitions). As long as queue Q has unexplored states ($Q \neq [\,]$) the first state of Q is selected for exploration and removed from Q. This state, the current state, is stored in the variable *cp*. It is explored by computing its (reduced) semantics $sc'_{d_0}(cp)$. This semantics is stored in the variable *sos*. Note that it contains terminations, action transitions, and delay transitions (see Definition 7.1). The semantics stored in *sos* is added to the state space S. Further, the fresh states in *sos* are appended to the queue Q. Fresh states are those states in *sos* that are neither in S nor in Q. These states are computed by the function *freshStates*. If there are no more unexplored states, the algorithm ends and returns the state space S.

$ComputeStateSpace(p)$:
  $Q := [p]$
  $S := emptyStateSpace()$
  while $Q \neq [\,]$:
    $cp := hd(Q)$
    $Q := tl(Q)$
    $sos := sc'_{d_0}(cp)$
    $S := S \cup sos$
    $Q := Q +\!\!+ freshStates(S, Q, sos)$
  return S

Once the state space is computed, deadlock states can easily be detected by scanning through all states and filtering out those states that cannot terminate and that do not have (outgoing) transitions. In addition, the state space can be saved both as an *fc2* file and as a *dot* file. The *fc2* file can be used for model checking by the FCTOOLS. The *dot* file can be used for visualisation by the Graphviz tools. These tools are discussed in Section 7.7.

7.6 · Tool related extensions

In this section we discuss some extensions of χ_σ that are implemented in the χ_σ engine, but are not defined formally. The extensions include a data type for

real numbers, some syntactic sugar for programming variable declarations and initializations, and probabilistic language constructs.

The real number extension is based on the real number data type of Python (the implementation language of the χ_σ engine). This extension adds a type for real numbers, called real, and common operations on those numbers.

Recall that states have the syntactic form $v_1 : v_2 : \ldots : v_n : \lambda_s$, where for $0 < i \leq n$, we have that v_i is a valuation. There are two types of valuations: channel valuations and programming variable valuations. A channel valuation has the form $m \mapsto c$, for m a channel and c a value. A programming variable valuation has the form $x : t \mapsto c$, for x a programming variable, t a type, and c a value. A concrete example of a state is $(x : \text{nat} \mapsto \bot) : (\sim m \mapsto 1) : (y : \text{real} \mapsto -1.2) : \lambda_s$. Recall that '$\bot$' denotes an unspecified value. This state contains two programming variable valuations (programming variables x : nat and y : real) and one channel valuation (channel $\sim m$). Instead of the '$\mapsto$', we can use the equality symbol '='. Also, the colons separating the valuations in a state may be replaced by commas. In that case, the parentheses and the 'λ_s' at the end can be dropped as well. Finally, channel valuations of the form $\sim m \mapsto \bot$ can be abbreviated to $\sim m$, and programming variable valuations of the form $x : t \mapsto \bot$ can be abbreviated to $x : t$. So, our example can be abbreviated to $x : \text{nat}, \sim m = 1, y : \text{real} = -1.2$.

In Section 5.5, we mentioned that χ does have probabilistic language constructs and χ_σ does not. We also mentioned that this is a serious omission of χ_σ, since without these language constructs, performance properties like average cycle time and average throughput cannot be determined. Therefore, the χ_σ engine is extended with a data type for real-valued distributions and a sample expression to draw samples from a distribution. The real-valued distribution type is denoted by dist[*real*]. A distribution can be sampled by means of the *sample* construct. For instance, if d is a distribution, sampling d is denoted by $\mathit{sample}(d)$. Values of distribution types are created by a built-in function. Table 7.4 shows the distributions implemented in the χ_σ engine.

Let us reconsider machine M presented on page 45. This machine processes lots with a processing time that is Gamma distributed. Lots are represented by natural numbers and the setseed statement is not translated (see Section 6.3). The definition of machine M in χ_σ reads

$$M(a, b\colon \text{chan},\ m, v\colon \text{real}) =$$
$$[\![\, d\colon \text{dist[real]} = \mathit{gamma}(\tfrac{m^2}{v}, \tfrac{v}{m}), x\colon \text{nat} \mid (a\,?\,x\ ;\ \Delta\mathit{sample}(d)\ ;\ b\,!\,x)^*\ ;\ \delta\,]\!].$$

Distribution	Syntax	Parameters	Value range
Exponential	$exp(m)$	$m \in R$ mean value	$R_{\geq 0}$
Normal	$nor(m, d)$	$m \in R$ mean value $d \in R$ standard deviation	R
Gamma	$gamma(p, q)$	$p, q \in R$ $p \times q$ mean value $p \times q^2$ variance	$R_{\geq 0}$
Uniform	$uni(l, u)$	$l \in R$ lower bound $u \in R$ upper bound	$\{r \in R \mid l \leq r < u\}$

Table 7.4 · Distributions of the χ_σ engine.

7.7 · Third party tools

The χ_σ engine is designed in such a way that it can be integrated with third party tools. The main reason for this is that we wanted to reuse existing applications and libraries as much as possible. The main consequence of this decision is that the programming interface of the χ_σ engine is defined in Python, since this language is particularly useful if it comes to integration of tools and libraries. Currently, the χ_σ engine is integrated with two different tools: fcTools, and Graphviz.

The fcTools are developed jointly by INRIA and Ecole des Mines/CMA as part of the MEIJE research team [168]. The tools share a common file exchange format, called *fc2*, for networks of communicating systems, and provide functionality to construct, reduce, and analyze concurrent systems. The *fc2* file format is designed to specify finite LTSs. Multiple LTSs can be structured using *networks* of LTSs. Furthermore, a network can contain sub-networks, thereby enabling a hierarchical architecture of systems.

The fcTools provide functionality to flatten hierarchical networks of LTSs into one big LTS. After that, it is possible to minimize the LTSs under different equivalence relations on the states of the LTS. The equivalence relations the fcTools understand are strong bisimulation, weak bisimulation, and branching bisimulation. In addition, the fcTools can check if two LTSs are equal under one of these equivalence relations. This functionality enables specification-implementation checks of models of industrial systems. To this end, the user can define both a specification and an implementation of the concurrent system under consideration. Usually,

the implementation contains internal actions. Under branching equivalence, the FCTOOLS can abstract from these internal actions and check if the abstracted LTS of the implementation is branching bisimilar to the specification LTS.

The χ_σ engine can write state spaces of χ_σ processes as LTSs in the *fc2* file format. Since the FCTOOLS do not make a distinction between delay transitions and action transitions, the transformation from state spaces of χ_σ processes into *fc2* LTSs disregards this difference. As mentioned above, the χ_σ-LTSs computed by the χ_σ engine are finite and can therefore be analysed by the FCTOOLS. Consequently, after generation of the (reduced) state space of a process, it is possible to use the FCTOOLS to analyze this state space. For example, we can minimize the state space under (untimed) strong bisimulation or (untimed) branching bisimulation. If two processes p and q are branching bisimular under this (untimed) branching bisimulation, we write $p \leftrightarrows_b q$. In addition, the χ_σ engine contains a script that transforms *fc2* files into *dot* files, the input format for the visualisation tools discussed in the next section. Using this script, it is possible to visualize state spaces minimized by the FCTOOLS.

Graphviz [74, 124] is an open toolkit for graph visualisation. It is developed at AT&T Labs-Research. The Graphviz tools use a common language to specify attributed graphs. This language is called *Libgraph*, but is probably better known as the *dot* format, after its best-known application. Graphviz provides tools for graph filtering and graph rendering. The filtering tools can be batch-oriented as well as interactive.

For our application, visualisation of state spaces of χ_σ processes, we only need a small part of the functionality offered by Graphviz. For instance, there are only four different types of nodes (initial, termination, deadlock, and normal nodes) and two different types of edges (action and delay transitions) in our graphs. Moreover, we do not need Graphviz facilities to structure graph specifications hierarchically.

7.8 • Experiment environment

If different instantiations of χ_σ process definitions should be analysed, manipulation of output becomes a considerable task. An experiment environment provides functionality to perform this task efficiently. The experiment environment of χ_σ is a front end to the functionality of the χ_σ engine. In addition, it provides scripting

features and interfaces to the FCTOOLS and the Graphviz toolkit. In Appendix B, a sample session with the experiment environment is presented.

We implemented the experiment environment in Python, the same language as the implementation language of the χ_σ engine. Python is an object-oriented scripting language. Consequently, χ_σ experiments can be defined as Python scripts. Since the functionality of the χ_σ engine is available to these scripts, they can read, simulate, and model check χ_σ specifications. This results in internal representations of specifications, traces (sequences of transitions), and process graphs, respectively. These results can be manipulated like ordinary Python objects. For example, after simulating the specification of a production system, the throughput and average cycle time can be computed by analysing the trace obtained. So, analysis need not be coded in the specification. Moreover, scripts have complete control over simulation steps. For instance, the number of simulation steps and the selection of steps can be programmed in scripts. Therefore, functionality as offered by the terminate statement of χ is also provided.

7.9 · Discussion

In this chapter, we showed that χ_σ can be supported by tools. The tools we developed for χ_σ provide functionality to check if a process can terminate, and if it can perform an action or a delay transition. We validated the correctness of the implementation with respect to the formal semantics by testing. In addition, two versions of the formal semantics have been implemented and checked for mutual consistency. Furthermore, we established theoretical results enabling a reduction of infinite process graphs to finite process graphs. This reduction enables effective computation of (reduced) process graphs and integration with existing tools to manipulate finite process graphs. The reduction applies to all (χ_σ translations of) χ processes.

The tools developed are integrated with third party tools to analyse and visualise process graphs. In addition, an experiment environment for χ_σ has been developed. It provides a uniform interface to the functionality of both the χ_σ tools and the integrated third party tools. Even though the current version of the χ_σ engine is a prototype, it already proved to be useful during analysis of industrial systems (see next chapter).

Examples and cases · 8

In this chapter, we explain the behaviour of χ_σ operators and show how they interact with each other. Examples containing small χ_σ models are discussed and their behaviour is illustrated by process graphs. Furthermore, we describe results of case studies. Each study considers a (small) production system that we specify in χ_σ and subsequently analyse with respect to its performance behaviour or functional behaviour.

In this chapter, we applied the reduction technique for infinite LTSs as implemented by the χ_σ engine (see Section 7.4). Some examples and case studies that are discussed in this chapter, were analysed using the χ_σ engine and use real numbers and distributions.

This chapter is organised as follows. Section 8.1 describes graphical conventions used in this chapter and Section 8.2 describes process specifications. Next, Sections 8.3 through 8.5 illustrate fundamental concepts of χ_σ. Section 8.6 illustrates specification-implementation checks by discussing a toy example. Sections 8.7 through 8.9 illustrate how industrial systems can be analysed in χ_σ. This chapter is concluded by a discussion in Section 8.10.

8.1 · Process graphs

Section 4.2 introduced conventions to depict process graphs:

- action transitions are solid edges labelled by an action,
- delay transitions are dashed edges labelled by a positive real number,
- terminations are represented by grey states,
- the initial state has a double circle,
- states without (outgoing) transitions and terminations are black.

It is often the case that there exists more than one graph which correctly illustrates a process' behaviour. For instance, consider the process $[\![(x : \text{nat} \mapsto \bot) : \lambda_s \mid (x := 4)^*]\!]$. Both graphs in Figure 8.1 are correct process graphs for this process.

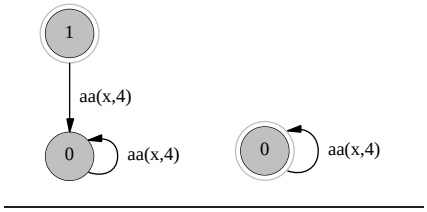


Figure 8.1 · Process graphs.

8.2 · Process specifications

In Section 4.18, we described how process specifications can be written in χ_σ. Here, we illustrate the general case by an example. Consider the specification of machine M:

$$M(a, b : \text{chan}, pt : \text{int}) = [\![(x : \text{int} \mapsto 0) : \lambda_s \mid (a\,?\,x \;;\; \Delta pt \;;\; b\,!\,x)^* \;;\; \delta]\!].$$

By instantiating a process specification, a concrete process results. For instance, $M(\sim m, \sim n, 3)$, that is, instantiating M with channels $\sim m$ and $\sim n$, and with process time 3, results in the process

$$[\![(x : \text{int} \mapsto 0) : \lambda_s \mid (\sim m\,?\,x \;;\; \Delta 3 \;;\; \sim n\,!\,x)^* \;;\; \delta]\!].$$

Note that the formal parameters of M are replaced by the actual parameters; instantiation is just syntactic replacement. Therefore, the formal parameter pt cannot be used as a programming variable in the process term that defines M. Therefore, the definition

$$M(a, b : \text{chan}, pt : \text{int}) =$$
$$[\![(x : \text{int} \mapsto 0) : \lambda_s \mid pt := pt + 2 \;;\; (a\,?\,x \;;\; \Delta pt \;;\; b\,!\,x)^* \;;\; \delta]\!]$$

is an invalid process definition. The problem is that after instantiation, the left-hand side of the assignment is not a programming variable anymore, but a value:

$$[\![(x : \text{int} \mapsto 0) : \lambda_s \mid 3 := 3 + 2 \;;\; (\sim m\,?\,x \;;\; \Delta 3 \;;\; \sim n\,!\,x)^* \;;\; \delta]\!].$$

The syntactic replacement applies only to the free programming variables in a process. A programming variable is free in a process, if it is not defined in a state operator enclosing that process. However, free programming variables can occur in expressions that define initial values for programming variables in the state of a state process. Consequently, it is possible to use one identifier both as a parameter and as a local programming variable:

$$M(a, b : \text{chan}, pt : \text{int}) =$$
$$[\![(x : \text{int} \mapsto 0) : (pt : \text{int} \mapsto pt) : \lambda_s \mid pt := pt + 2 \ ; \ (a \,?\, x \ ; \ \Delta pt \ ; \ b \,!\, x)^* \ ; \ \delta]\!].$$

If this process specification is instantiated as above, $M(\sim m, \sim n, 3)$, we obtain

$$[\![(x : \text{int} \mapsto 0) : (pt : \text{int} \mapsto 3) : \lambda_s \mid pt := pt + 2 \ ; \ (\sim m \,?\, x \ ; \ \Delta pt \ ; \ \sim n \,!\, x)^* \ ; \ \delta]\!].$$

As can be seen, only the occurrence of pt in the right-hand side of a valuation of the local state is substituted by the actual parameter.

8.3 · Time factorisation and maximal progress

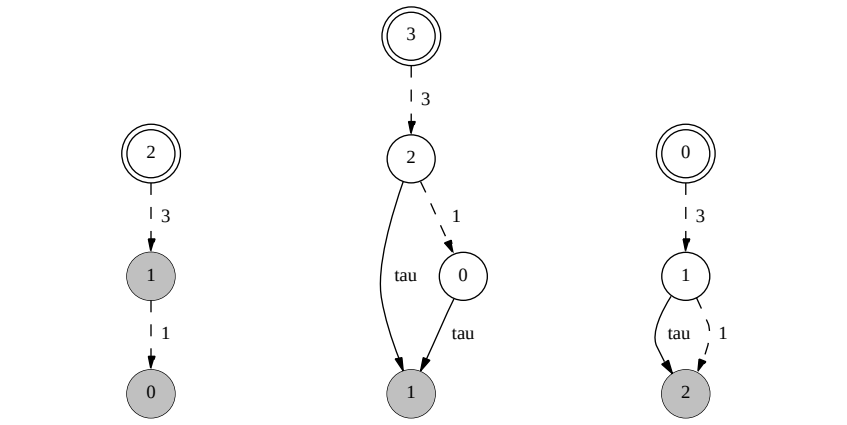
In Section 4.3, we discussed the notion of time factorisation and maximal progress. We explained that we decided for an interpretation of time factorisation such that opportunities for action performance and termination cannot be ignored (Figure 4.3(c)). The examples below illustrate that in χ_σ this is indeed the case. Furthermore, we also illustrate the distribution of the sequential composition operator over the alternative composition operator as formalised in Lemma 4.23 and the time factorisation property as formalised in Lemma 4.47. Consider the following processes:

$$TF_1 = \Delta 3 \,[\!]\, \Delta 4,$$
$$TF_2 = \Delta 3 \ ; \ \mathsf{skip} \,[\!]\, \Delta 4 \ ; \ \mathsf{skip},$$
$$TF_3 = (\Delta 3 \,[\!]\, \Delta 4) \ ; \ \mathsf{skip},$$
$$TF_4 = \Delta 3 \ ; \ \mathsf{skip} \,[\!]\, \Delta 4,$$
$$TF_5 = \Delta 3 \ ; \ (\mathsf{skip} \,[\!]\, \Delta 1).$$

The graph of process TF_1 is depicted in Figure 8.2(a). It shows that both alternatives delay together and that TF_1 cannot delay more than 3 time units at once.

Processes TF_2 and TF_3 are equal according to Lemma 4.23. Consequently, their process graphs are identical (see Figure 8.2(b)). Also here, we see that both

alternatives delay together and that they cannot delay more than 3 time units at once. The same holds for the processes TF_4 and TF_5. They are equal according to Lemma 4.47 and their process graph is depicted in Figure 8.2(c).



(a) Graph of TF_1. (b) Graph of $TF_{2,3}$. (c) Graph of $TF_{4,5}$.

Figure 8.2 · Time Factorisation.

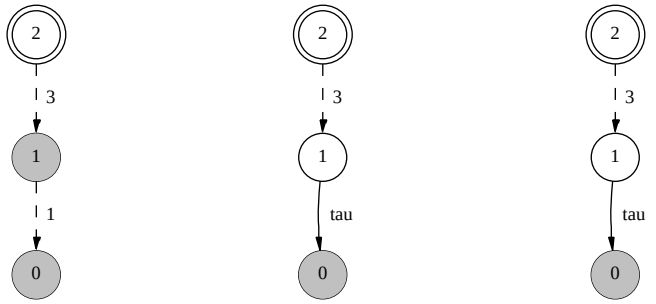
Let us next reconsider the processes TF_1, TF_3, and TF_5 under maximal progress:

$$MP_1 = \pi(\Delta 3 \between \Delta 4),$$
$$MP_2 = \pi((\Delta 3 \between \Delta 4) \mathbin{;} \mathsf{skip}),$$
$$MP_3 = \pi(\Delta 3 \mathbin{;} (\mathsf{skip} \between \Delta 1)).$$

The graph of process MP_1, as depicted in Figure 8.3(a), is identical to the graph of process TF_1. Here the maximal progress operator has no effect because process TF_1 does not perform any actions. However, the maximal progress operator does have effect when applied to the processes TF_3 and TF_5. Their process graphs are depicted in Figure 8.3(b) and 8.3(c). As can be seen, the alternative to delay for 1 time unit is lost because of the opportunity to perform the internal action τ. As a consequence, under maximal progress the processes TF_3 and TF_5 are equal.

Besides the alternative composition operator, also the sequential composition operator accounts for time factorisation. This is illustrated by the following process:

$$TF_6 = (\Delta 3 \between \varepsilon) \mathbin{;} \Delta 4.$$



(a) Graph of MP_1. (b) Graph of MP_2. (c) Graph of MP_3.

Figure 8.3 · Maximal progress.

Since the first argument of the sequential composition can both delay and terminate, and the second argument can delay too, we have in fact two delay alternatives, which should delay together.

The graph of process TF_6 is depicted in Figure 8.4. At first sight, it may not be obvious that this process graph corresponds to the behaviour specified in process TF_6. Therefore, consider the following computation:

$$\begin{aligned}
TF_6 &\leftrightarrows (\Delta 3 \between \varepsilon) \mathbin{;} \Delta 4 \\
&\leftrightarrows \{\text{Lemma 4.23}\} \\
&\quad \Delta 3 \mathbin{;} \Delta 4 \between \varepsilon \mathbin{;} \Delta 4 \\
&\leftrightarrows \{\text{Lemma 4.20}\} \\
&\quad \Delta 3 \mathbin{;} \Delta 4 \between \Delta 3 + 1 \\
&\leftrightarrows \{\text{Lemma 4.47}\} \\
&\quad \Delta 3 \mathbin{;} (\Delta 4 \between \Delta 1) \\
&\leftrightarrows \{\text{Lemma 4.16}\} \\
&\quad \Delta 3 \mathbin{;} (\Delta 1 \between \Delta 4) \\
&\leftrightarrows \{\text{Lemma 4.19}\} \\
&\quad \Delta 3 \mathbin{;} (\Delta 1 \mathbin{;} \varepsilon \between \Delta 1 + 3) \\
&\leftrightarrows \{\text{Lemma 4.47}\} \\
&\quad \Delta 3 \mathbin{;} \Delta 1 \mathbin{;} (\varepsilon \between \Delta 3).
\end{aligned}$$

So, we see that $TF_6 \leftrightarrows \Delta 3 \mathbin{;} \Delta 1 \mathbin{;} (\varepsilon \between \Delta 3)$, which corresponds directly to the graph of Figure 8.4.

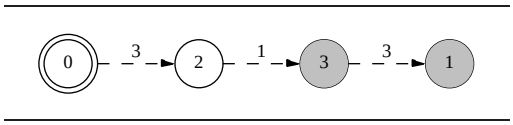


Figure 8.4 · Graph of TF_6.

8.4 · Programming variables and scoping

This section explains how we can use programming variables by introducing the state operator. We first present a rather simple example, followed by a more complex one containing nested states.

Consider the following process:

$$S_1 = [\![(x : \mathrm{nat} \mapsto 1) : \lambda_s \mid x := x + 1]\!].$$

We derive the action this process can perform and we derive the process that results from performing that action. If there is a transition possible, Rule 33 should apply:

$$\langle [\![(x : \mathrm{nat} \mapsto 1) : \lambda_s \mid x := x + 1]\!], \sigma \rangle \xrightarrow{a} \langle [\![s \mid p]\!], \sigma' \rangle,$$

with a an action, s a state, and p a process. According to that same rule, we also have

$$\langle x := x + 1, (x : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \rangle \xrightarrow{a} \langle p, s :: \sigma' \rangle.$$

This is indeed possible due to Rule 4, since $((x : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma)(x + 1) = 2$ according to Definition A.11. Consequently, we obtain

$$\langle x := x + 1, (x : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \rangle \xrightarrow{aa(x,2)} \langle \varepsilon, ((x : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma)[2/x] \rangle.$$

According to Definition A.10, this is equal to

$$\langle x := x + 1, (x : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \rangle \xrightarrow{aa(x,2)} \langle \varepsilon, (x : \mathrm{nat} \mapsto 2) : \lambda_s :: \sigma \rangle.$$

This now gives us the transition we were looking for, because we can apply Rule 33 and obtain

$$\langle [\![(x : \mathrm{nat} \mapsto 1) : \lambda_s \mid x := x + 1]\!], \sigma \rangle \xrightarrow{aa(x,2)} \langle [\![(x : \mathrm{nat} \mapsto 2) : \lambda_s \mid \varepsilon]\!], \sigma \rangle.$$

The next example we consider is slightly more complicated. Consider a process S_2 with two nested states. The outermost state has two programming variables, x and y, whereas the innermost state has only one programming variable, y:

$$
\begin{array}{l}
S_2 = \\
[\![\, (x : \mathrm{nat} \mapsto 3) : (y : \mathrm{nat} \mapsto 1) : \lambda_s \\
\mid [\![\, (y : \mathrm{nat} \mapsto 2) : \lambda_s \mid x := x + y \;;\; y := x \times y \,]\!] \;;\; y := x - y \\
]\!].
\end{array}
$$

The behaviour of this process is depicted in Figure 8.5. As can be seen, first the assignment action $aa(x, 5)$ is performed, thereby updating the value of x in the outermost state to 5. Note that the value of y in the expression $x + y$ is obtained from the innermost state. After that, it performs action $aa(y, 10)$ and updates the value of y in the innermost state to 10. The final action S_2 performs is $aa(y, 4)$, thereby updating the value of y in the outermost state to 4. Note that the value of y in the innermost state does not influence this final assignment to y. The first transition of process S_2 is derived as follows. Using Rule 4, we have (for arbitrary σ)

$$
\begin{array}{l}
\langle\, x := x + y \\
,\, (y : \mathrm{nat} \mapsto 2) : \lambda_s :: (x : \mathrm{nat} \mapsto 3) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle \\
\quad \xrightarrow{aa(x,5)} \\
\langle\, \varepsilon \\
,\, (y : \mathrm{nat} \mapsto 2) : \lambda_s :: (x : \mathrm{nat} \mapsto 5) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle.
\end{array}
$$

According to Rule 18 we have

$$
\begin{array}{l}
\langle\, x := x + y \;;\; y := x \times y \\
,\, (y : \mathrm{nat} \mapsto 2) : \lambda_s :: (x : \mathrm{nat} \mapsto 3) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle \\
\quad \xrightarrow{aa(x,5)} \\
\langle\, \varepsilon \;;\; y := x \times y \\
,\, (y : \mathrm{nat} \mapsto 2) : \lambda_s :: (x : \mathrm{nat} \mapsto 5) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle.
\end{array}
$$

Applying Rule 33 then gives

$$
\begin{array}{l}
\langle\, [\![\, (y : \mathrm{nat} \mapsto 2) : \lambda_s \mid x := x + y \;;\; y := x \times y \,]\!] \\
,\, (x : \mathrm{nat} \mapsto 3) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle \\
\quad \xrightarrow{aa(x,5)} \\
\langle\, [\![\, (y : \mathrm{nat} \mapsto 2) : \lambda_s \mid \varepsilon \;;\; y := x \times y \,]\!] \\
,\, (x : \mathrm{nat} \mapsto 5) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle.
\end{array}
$$

Next, by applying Rule 18 we obtain

$$
\begin{array}{l}
\langle\, [\![\, (y : \mathrm{nat} \mapsto 2) : \lambda_s \mid x := x + y \;;\; y := x \times y \,]\!] \;;\; y := x - y \\
,\, (x : \mathrm{nat} \mapsto 3) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle \\
\quad \xrightarrow{aa(x,5)} \\
\langle\, [\![\, (y : \mathrm{nat} \mapsto 2) : \lambda_s \mid \varepsilon \;;\; y := x \times y \,]\!] \;;\; y := x - y \\
,\, (x : \mathrm{nat} \mapsto 5) : (y : \mathrm{nat} \mapsto 1) : \lambda_s :: \sigma \\
\,\rangle.
\end{array}
$$

Finally, we apply Rule 33 again and find the transition we were looking for:

$$
\begin{array}{l}
\langle\, [\![\, (x : \mathrm{nat} \mapsto 3) : (y : \mathrm{nat} \mapsto 1) : \lambda_s \\
\quad \mid [\![\, (y : \mathrm{nat} \mapsto 2) : \lambda_s \mid x := x + y \;;\; y := x \times y \,]\!] \;;\; y := x - y \\
\quad]\!] \\
,\, \sigma \\
\,\rangle \\
\quad \xrightarrow{aa(x,5)} \\
\langle\, [\![\, (x : \mathrm{nat} \mapsto 5) : (y : \mathrm{nat} \mapsto 1) : \lambda_s \\
\quad \mid [\![\, (y : \mathrm{nat} \mapsto 2) : \lambda_s \mid \varepsilon \;;\; y := x \times y \,]\!] \;;\; y := x - y \\
\quad]\!] \\
,\, \sigma \\
\,\rangle.
\end{array}
$$

Similar derivations can be made for the remaining transitions of Figure 8.5.

8.5 · Concurrency and communication

In this section we present some examples in order to illustrate the idea of concurrency and the concept of communication. The main operator under consideration

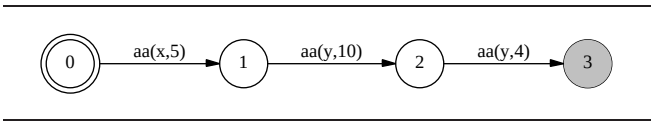


Figure 8.5 · Graph of S_2.

is the '$\|$' operator. With respect to this operator, we look at action interleaving, communication, and delay behaviour.

Consider the following process:

$$C_1 = \mathsf{skip}\ ;\ \Delta 3 \,\|\, [\![\, (b : \mathrm{bool} \mapsto \bot) : \lambda_s \mid b := \mathit{true}\ ;\ \Delta 4 \,]\!].$$

By looking at its process graph, which is depicted in Figure 8.6, we can see that if two processes are put in parallel, their actions are interleaved. As far as their delay behaviour is concerned, we see that if two processes are able to perform a certain delay, then they perform that delay together. In Figure 8.6 that would be the delay of 3 time units. After that delay, the left-hand side argument of the '$\|$' operator of C_1 can do nothing but terminate, which enables the right-hand side argument to finish by performing a delay of 1 time unit.

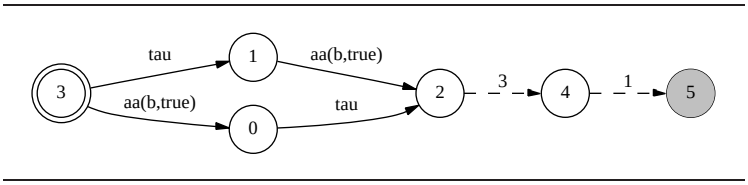


Figure 8.6 · Graph of C_1.

Besides interleaving actions, the merge operator is also able to let send and receive processes communicate. This is illustrated by the following process:

$$C_2 = [\![\, (\sim m \mapsto \bot) : \lambda_s \mid \sim m\,!\,\mathit{true} \,\|\, [\![\, (b : \mathrm{bool} \mapsto \bot) : \lambda_s \mid \sim m\,?\,b \,]\!] \,]\!].$$

In Figure 8.7(a) the process graph of C_2 is depicted. As can be seen, process C_2 can perform a send action and send the value *true* via channel $\sim m$. After that, it can perform a receive action and receive the value *true* in programming variable b

via $\sim m$. Besides asynchronous communication, C_2 can also perform a communication action and thereby communicate the value *true* synchronously via $\sim m$. The process graph of C_2 also shows that both the send and receive process in C_2, $\sim m!true$ and $\sim m?b$, can delay for an arbitrary number of time units (this is needed because we want send and receive processes to be able to wait for communication).

Often, we are only interested in synchronous communication. In that case we can use the encapsulation operator to encapsulate send and receive actions. Suppose we want to encapsulate all send and receive actions in process C_2. In that case, we define a set $A = \{sa(m,c) \mid m \in \mathit{Channel} \wedge c \in \mathit{Value}\} \cup \{ra(m,x) \mid m \in \mathit{Channel} \wedge x \in \mathit{Var}\}$ and a process

$$C_3 = \partial_A[\![(\sim m \mapsto \bot) : \lambda_s \mid \sim m \,!\, \mathit{true} \parallel [\![(b : \mathrm{bool} \mapsto \bot) : \lambda_s \mid \sim m \,?\, b]\!]]\!].$$

The process graph of C_3 is depicted in Figure 8.7(b). The ability to delay an arbitrary number of times still exist. The latter is mostly also unwanted. We do not want processes to delay if they can also perform an action. This can be established by using the maximal progress operator. This operator enforces a process to perform actions if it can and allows it to delay if it cannot perform any activity. If we apply the maximal progress operator to process C_3, we obtain

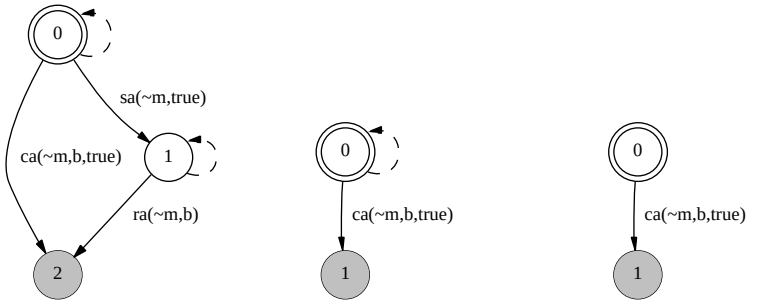
$$C_4 = \pi\partial_A[\![(\sim m \mapsto \bot) : \lambda_s \mid \sim m \,!\, \mathit{true} \parallel [\![(b : \mathrm{bool} \mapsto \bot) : \lambda_s \mid \sim m \,?\, b]\!]]\!].$$

The process graph of C_4 is depicted in Figure 8.7(c). As we can see, the ability to delay is lost.

8.6 · Specification-implementation equivalence

Branching bisimulation enables us to analyse specification-implementation equivalence. For example, if the desired external behaviour of a system is defined separately in the specification, then we can check whether the implementation satisfies this behaviour by abstracting from internal behaviour. Recall that we use untimed branching bisimulation on process graphs as described in Section 7.7.

For example, consider the processes Sum_1 and Sum_2:



(a) Graph of C_2. (b) Graph of C_3. (c) Graph of C_4.

Figure 8.7 · Communication.

$Sum_1(a, b : \text{chan}) =$
$[\![(x : \text{nat} \mapsto \bot) : (y : \text{nat} \mapsto 0) : \lambda_s$
$\mid a\,?\,x \;;\; (x > 0 :\to y := y + x \;;\; x := x - 1)^* \;;\; (x = 0 :\to b\,!\,y)$
$]\!]$,

$Sum_2(a, b : \text{chan}) =$
$[\![(x : \text{nat} \mapsto \bot) : \lambda_s \mid a\,?\,x \;;\; b\,!\,\frac{1}{2} \times x \times (x + 1)]\!]$.

Process Sum_1 computes the sum of the series $\sum_{i=0}^{x} i$ for a natural number x that it receives via a. The computed result is sent via b. However, the desired result can be obtained in a more efficient way because

$$\sum_{i=0}^{x} i = \frac{1}{2} \times x \times (x + 1),$$

for every natural number x (proof omitted). This way to compute the sum of series $\sum_{i=0}^{x} i$ is specified in process Sum_2.

Let us now define the two processes I and S which represent implementation and specification, respectively:

$I(m : \text{nat}) = [\![(\sim in \mapsto m) : (\sim out \mapsto \bot) : \lambda_s \mid \pi Sum_1(\sim in, \sim out)]\!]$,

$S(m : \text{nat}) = [\![(\sim in \mapsto m) : (\sim out \mapsto \bot) : \lambda_s \mid \pi Sum_2(\sim in, \sim out)]\!]$,

and show that for $A = \{aa(x,c) \mid x \in \mathit{Var} \wedge c \in \mathit{Value}\}$ we have $\tau_A I(n) \leftrightarrows_b S(n)$ for every natural number n.

We start with the definition of process I':

$$
\begin{array}{l}
I'(m : \mathrm{nat}, i : \mathrm{nat}) = \\
\pi[\![\ (x : \mathrm{nat} \mapsto m) : (y : \mathrm{nat} \mapsto i) : \lambda_s \\
\quad \mid (x > 0 :\to y := y + x\ ;\ x := x - 1)^*\ ;\ (x = 0 :\to {\sim}\mathit{out}\,!\,y) \\
\quad]\!].
\end{array}
$$

Its process graph is defined as follows, where n and j are natural numbers:

$$
\begin{array}{ll}
I'(0,j) \xrightarrow{sa({\sim}out,j)} \pi[\![\ (x : \mathrm{nat} \mapsto n) : (y : \mathrm{nat} \mapsto j) : \lambda_s \mid \varepsilon\ ]\!] & \text{if } n = 0, \\
I'(n,j) \xrightarrow{aa(y,j+n)} I'(n, j+n) \xrightarrow{aa(x,n-1)} I'(n-1, j+n) & \text{if } n > 0.
\end{array}
$$

For the graph of process $\tau_A I(n)$ we find

$$
\begin{array}{l}
\tau_A[\![\ ({\sim}\mathit{in} \mapsto n) : ({\sim}\mathit{out} \mapsto \bot) : \lambda_s \\
\quad \mid \pi[\![\ (x : \mathrm{nat} \mapsto \bot) : (y : \mathrm{nat} \mapsto 0) : \lambda_s \\
\qquad \mid {\sim}\mathit{in}\,?\,x\ ;\ (x > 0 :\to y := y + x\ ;\ x := x - 1)^*\ ;\ (x = 0 :\to {\sim}\mathit{out}\,!\,y) \\
\qquad]\!] \\
\quad]\!] \\
\\
\xrightarrow{ra({\sim}in,x)} \\
\\
\tau_A[\![\ ({\sim}\mathit{in} \mapsto n) : ({\sim}\mathit{out} \mapsto \bot) : \lambda_s \\
\quad \mid \pi[\![\ (x : \mathrm{nat} \mapsto n) : (y : \mathrm{nat} \mapsto 0) : \lambda_s \\
\qquad \mid \varepsilon\ ;\ (x > 0 :\to y := y + x\ ;\ x := x - 1)^*\ ;\ (x = 0 :\to {\sim}\mathit{out}\,!\,y) \\
\qquad]\!] \\
\quad]\!],
\end{array}
$$

which is in fact

$$\tau_A I(n) \xrightarrow{ra({\sim}in,x)} \tau_A[\![\ ({\sim}\mathit{in} \mapsto n) : ({\sim}\mathit{out} \mapsto \bot) : \lambda_s \mid I'(n,0)\ ]\!].$$

So, we find that the process graph of $\tau_A I(n)$ starts with receive action $ra({\sim}in, n)$, followed by n pairs of internal actions τ (the hidden possible assignment actions of $I'(n,0)$), and finishes with send action ${\sim}\mathit{out}\,!\,n'$ where n' is the computed sum of the series $0 + \sum_{i=0}^{n} n - i$, which is equal to $\sum_{i=0}^{n} i$ (proof omitted).

For the graph of process $S(n)$ we find

$$\tau_A\llbracket \sim in \mapsto n : \sim out \mapsto \bot : \lambda_s \\ \quad \mid \pi\llbracket (x : \mathrm{nat} \mapsto \bot) : (y : \mathrm{nat} \mapsto 0) : \lambda_s \mid \sim in \,?\, x \;;\; \sim out \,!\, \tfrac{1}{2} \times x \times (x+1)) \rrbracket \\ \quad \rrbracket$$

$$\xrightarrow{ra(\sim in, x)}$$

$$\tau_A\llbracket (\sim in \mapsto n) : (\sim out \mapsto \bot) : \lambda_s \\ \quad \mid \pi\llbracket (x : \mathrm{nat} \mapsto n) : (y : \mathrm{nat} \mapsto 0) : \lambda_s \mid \varepsilon \;;\; \sim out \,!\, \tfrac{1}{2} \times x \times (x+1)) \rrbracket \\ \quad \rrbracket$$

$$\xrightarrow{sa(\sim out, n')}$$

$$\tau_A\llbracket (\sim in \mapsto n) : (\sim out \mapsto \bot) : \lambda_s \\ \quad \mid \pi\llbracket (x : \mathrm{nat} \mapsto n) : (y : \mathrm{nat} \mapsto 0) : \lambda_s \mid \varepsilon \rrbracket \\ \quad \rrbracket,$$

where n' is the computed outcome of $\frac{1}{2} \times n \times (n+1)$.

Now we know the structure of the process graphs of both $\tau_A I(n)$ and $S(n)$, we can construct a branching bisimulation relation between the two. Figure 8.8 depicts such a branching bisimulation relation. The dotted arrow in the process graph of $\tau_A I(n)$ represents the n pairs of τ steps.

8.7 · A simple flow line

In this section, we discuss a small case study on performance analysis. The performance properties we study here are throughput and average cycle time. The production system we consider is a flow line that consists of four buffer-machine units. The environment is modelled by a generator, which generates the lots to be processed by the machines, and an exit process that consumes all produced lots.

The throughput of the flow line and average cycle time of the lots depends on the number of lots present in the system during production. This number is referred to as WIP level; the level of work in process. Our objective is to determine how throughput and average cycle time relate to the selected WIP level. To that end, we introduce a WIP controller. This controller is integrated in the generator and keeps the WIP level at a constant level. Such controllers are also referred to as CONWIP controllers [110].

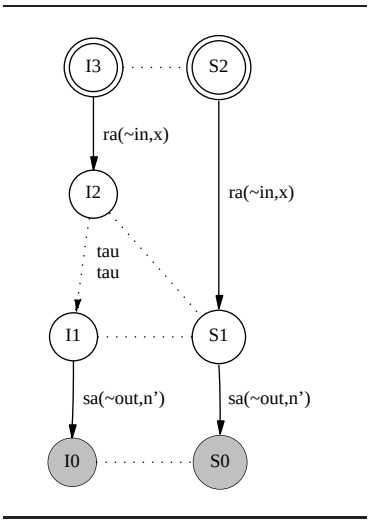


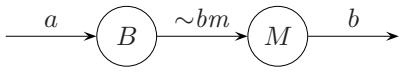
Figure 8.8 · Branching bisimulation between $\tau_A I(n)$ and $S(n)$.

The generator, the buffers, the machines, and the exit are each modelled by a process. The material and information flow in the system is modelled by communication between the different processes. That is, communication between two processes models transport of a lot from one process to another, or the exchange of information between two processes.

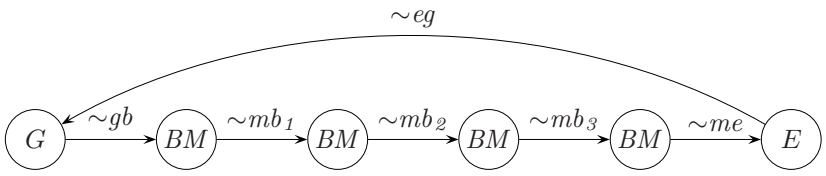
The generator, the buffers, the machines, and the exit are represented by the processes G, B, M, and E, respectively. Processes B and M are combined into a buffer machine unit BM. The flow line itself, called FL, is constructed from these units. The processes mentioned above are connected by the channels $\sim gb$, $\sim bm$, $\sim mb_1$, $\sim mb_2$, $\sim mb_3$, $\sim me$, and $\sim eg$. All channels but channel $\sim eg$ model the transportation of lots. Channel $\sim eg$ is part of the WIP controller. The architecture of BM and FL is depicted in Figure 8.9.

We start with the definition of the processes G and E:

$$
\begin{aligned}
&G(a, b : \text{chan},\ \mathit{wip} : \mathit{nat}) = \\
&[\![\, (w : \text{nat} \mapsto 0) : (x : \text{bool} \mapsto \bot) : \lambda_s \\
&|\ (w < \mathit{wip} :\rightarrow a\,!\,0\ ;\ w := w + 1 \,[\!]\, b\,?\,x\ ;\ w := w - 1)^*\ ;\ \delta \\
&]\!],
\end{aligned}
$$



(a) System BM with parameters a and b.



(b) System FL.

Figure 8.9 · Architecture of production system FL.

$$E(a, b : \text{chan}) = [\![\, (x : \text{nat} \mapsto \bot) : \lambda_s \mid (a\,?\,x \;;\; b\,!\,\mathit{true})^* \;;\; \delta \,]\!].$$

We specify infinite behaviour by defining a sequential composition containing a repetition followed by a deadlock. The deadlock process prevents the repetition from terminating which results in an infinite loop. The WIP level of FL is controlled by parameter wip of process G. The actual number of lots in the system is registered by programming variable w. If $w < wip$, a lot is sent to the first buffer and w is increased. Note that lots are represented by natural numbers. If we can communicate with the exit via b, this means that a lot has left the system so w is decreased.

Next, we define the processes B and M. Process B is defined by

$$\begin{aligned}
&B(a, b : \text{chan}) = \\
&[\![\, (x : \text{nat} \mapsto \bot) : (y : \text{nat} \mapsto 0) : \lambda_s \\
&\mid (a\,?\,x \;;\; y := y + 1 \,[\![\, y > 0 :\!\rightarrow b\,!\,0 \;;\; y := y - 1)^* \;;\; \delta \\
&]\!].
\end{aligned}$$

Process B is a buffer with unlimited capacity. This allows us to specify every WIP level we want. Namely, should the buffer capacity be limited, then the maximal WIP level is also limited. Depending on the value of guard $y > 0$ and opportunities for communication over the parameters a and b, process B either receives a lot via a, or sends a lot via b. The number of lots present in the buffer is registered in y.

Process M is defined by

$$
\begin{array}{l}
M(a, b : \text{chan},\ m, v : \text{real}) = \\
[\![\ (d : \text{dist}[\text{real}] \mapsto \text{uni}(0,1)) : (s : \text{real} \mapsto \bot) : (x : \text{nat} \mapsto \bot) : \lambda_s \\
|\ (\ a\,?\,x \\
\quad ;\ s := \mathit{sample}(d)\ ;\ (s < 0.5 :\!\rightarrow \Delta m - v\ [\!]\ s \geq 0.5 :\!\rightarrow \Delta m + v) \\
\quad ;\ b\,!\,x \\
\quad)^*\ ;\ \delta \\
]\!].
\end{array}
$$

This process repeatedly receives a lot, processes it, and sends it away. Variation in processing times is modelled by specifying a mean processing time m plus or minus a variation v. We define a continuous uniform distribution d with lower bound 0 and upper bound 1 (see Table 7.4), and determine the processing time of each lot by drawing a sample s from distribution d. If $s < 0.5$, the processing time equals $m - v$. If $s \geq 0.5$, the processing time equals $m + v$.

Define $A = \{sa(\sim bm, c) \mid c \in \mathit{Value}\} \cup \{ra(\sim bm, x) \mid x \in \mathit{Var}\}$, then process BM is now defined as

$$
BM(a, b : \text{chan}) = [\![\ (\sim bm \mapsto \bot) : \lambda_s \mid \partial_A B(a, \sim bm) \parallel M(\sim bm, b, 0.2, 0.1)\ ]\!].
$$

For reasons of simplicity, we assume that all machines have identical processing times. Their mean processing time is set to 0.2 hours with a variation of 0.1 hours.

Finally, we define $A' = \{sa(m, c) \mid m \in \mathit{Channel} \wedge c \in \mathit{Value}\} \cup \{ra(m, x) \mid m \in \mathit{Channel} \wedge x \in \mathit{Var}\}$ and process FL as

$$
\begin{array}{l}
FL(\mathit{wip} : \text{nat}) = \\
\pi\partial_{A'}[\![\ (\sim mb_1 \mapsto \bot) : (\sim mb_2 \mapsto \bot) : (\sim mb_3 \mapsto \bot) \\
\qquad : (\sim gb \mapsto \bot) : (\sim me \mapsto \bot) : (\sim eg \mapsto \bot) : \lambda_s \\
\qquad \mid G(\sim gb, \sim eg, \mathit{wip}) \\
\qquad \parallel BM(\sim gb, \sim mb_1) \parallel BM(\sim mb_1, \sim mb_2) \\
\qquad \parallel BM(\sim mb_2, \sim mb_3) \parallel BM(\sim mb_3, \sim me) \\
\qquad \parallel E(\sim me, \sim eg) \\
\qquad]\!].
\end{array}
$$

Process FL can be simulated and its dynamic behaviour can be studied. We investigate how throughput and average cycle time depend on the specified WIP level. The result is depicted in Figure 8.10. It shows the throughput and average cycle

time versus WIP level. As can be seen, there is a trade off between throughput and average cycle time. Depending on certain factors one can choose to have an almost maximal throughput if one accepts the average cycle time to be quite large. If a large average cycle time is not acceptable, then one will have to compromise on the throughput.

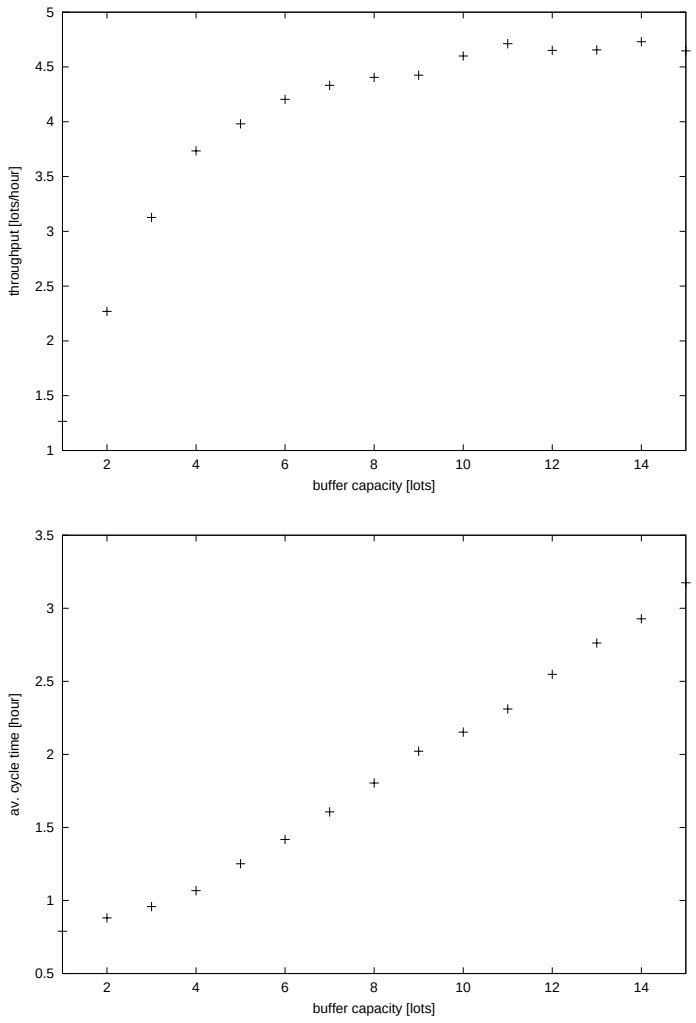


Figure 8.10 · Throughput and average cycle time versus WIP.

8.8 · A coating system

In this section, we design a controller for a coating system and show using specification-implementation equivalence that it works properly. In addition, we analyse performance properties of the coating system. A preliminary version of this case study is published in [41]. Below, we describe the case study in more detail. The coating system example, is taken from the semiconductor industry. It consists of a buffer, a coating machine, and a controller. The coating machine covers products with a special layer, the coating, in order to give it particular chemical and physical properties. Once the machine starts coating products, it should operate continuously. If the machine has to wait for a new product, then the coating process hampers and the machine must be stopped and cleaned. Furthermore, after a certain amount of products have been processed the machine has to be cleaned anyway. Our goal is to design a correct controller for the buffer-coating-machine combination.

Figure 8.11 shows the components of the coating system, which are, buffer B, coating machine M, and controller C. Buffer B receives products from the environment via channel EB and stores them. Machine M processes these products, which it receives via channel $\sim bm$, and returns them to the environment via channel me. We consider all products equal and model them by the value 0. Via channels $\sim cb_{get}$, $\sim cb_{put}$, $\sim cm_{coat}$, and $\sim cm_{clean}$ the controller communicates control signals to buffer B and machine M. All control signals are modelled by the value *true*.

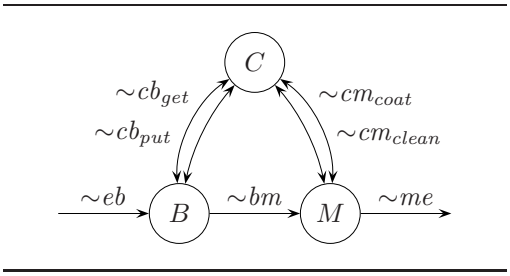


Figure 8.11 · Architecture of the coating system.

We assume the arrival times of the products at buffer B are nondeterministic. This is a legal assumption, since if we can establish correctness of the controller in situations with nondeterministic arrival times, then the system will also operate

correctly in more realistic situations where we have stochastic or deterministic arrival times.

We start with the definition of the processes B and M. Process B is defined by

$$
\begin{array}{l}
B(in, out, get, put : \mathrm{chan}) = \\
[\![\ (b : \mathrm{bool} \mapsto \bot) : (x : \mathrm{nat} \mapsto \bot) : (m : \mathrm{nat} \mapsto 0) : \lambda_s \\
\mid\ (get\,?\,b\ ;\ in\,?\,x\ ;\ m := m + 1\ ;\ get\,!\,true)^*\ ;\ \delta \\
\|\ (put\,?\,b\ ;\ (m > 0 :\rightarrow out\,!\,0)\ ;\ m := m - 1\ ;\ put\,!\,true)^*\ ;\ \delta \\
]\!].
\end{array}
$$

Buffer B has four parameters. Via the parameters *in* and *out* products are transported to and from the buffer, respectively. Via the parameters *get* and *put*, B is instructed to receive a product from the environment and to send a product to the machine, respectively. Within process B three programming variables are used. Programming variable b is used to receive control signals, programming variable x is used to receive products, and programming variable m denotes the number of products in the buffer.

As can be seen, process B has two concurrently executing infinite repetitions. In the first repetition, B waits for a control signal over *get*. This signal indicates that the buffer should try to receive a product from the environment via *in*. If a product is received, m is increased and B sends a control signal back over *get* (note that we communicate over channels in two directions). Parallel to this cycle, the buffer executes another cycle that waits for a control signal over *put*. If this signal arrives, process B tries to send a product to machine M via *out*. However, this is only possible if there is at least one product available. Therefore, we guarded process $out\,!\,0$ with the boolean expression $m > 0$. Consequently, if the buffer receives a control signal over *put* and it contains no products, it will deadlock. The implementation of the controller should prevent this. After B has sent a product, it sends a control signal back to the controller via *put*.

Process M is defined by

$$
\begin{array}{l}
M(in, out, coat, clean : \mathrm{chan}, t_p, t_c : \mathrm{real}) = \\
[\![\ (b : \mathrm{bool} \mapsto \bot) : (x : \mathrm{nat} \mapsto \bot) : \lambda_s \\
\mid\ (\ coat\,?\,b\ ;\ \pi(in\,?\,x\ ;\ \Delta t_p\ ;\ out\,!\,0\ ;\ coat\,!\,true) \\
\quad ;\ (\pi(coat\,?\,b\ ;\ in\,?\,x\ ;\ \Delta t_p\ ;\ out\,!\,0\ ;\ coat\,!\,true))^* \\
\quad ;\ clean\,?\,b\ ;\ \Delta t_c\ ;\ clean\,!\,true\ )^*\ ;\ \delta \\
]\!].
\end{array}
$$

Process M has six parameters. Via the parameters *in* and *out* products are received and sent, respectively. Via the parameters *coat* and *clean* control signals are received that instruct the machine to start coating and cleaning, respectively. Parameter t_p models the required processing time for coating one product and parameter t_c models the time it takes to clean machine M.

Machine M operates as follows. It waits for a control signal over *coat*. If this signal has been received, then it must be able to receive a product via *in*, process this product, send this product to the environment via *out*, and notify the controller that it did so over *coat*. Furthermore, due to the maximal progress operator, M is not able to wait for communication. If possible without delay, this can be repeated, else it waits for a cleaning signal over *clean*. When this signal has been received, the machine is cleaned, which is then signalled back to the controller via *clean*. So, the controller of the machine has to make sure that whenever it instructs the machine to start coating a product, the machine is able to receive a product without delay and, after processing, is able to send the product to the exit without delay. If either one of these actions is impossible, the machine deadlocks.

Before we specify the controller, we first define the desired external behaviour of the coating system. Depending on this specification, the controller has to satisfy some formal requirements.

We assume that the buffer has a finite storage capacity *cap*. As long as there are less than *cap* products in the buffer, new products can be accepted. If the buffer is full, we simply do not accept more products until at least one product is removed from the buffer. Further, in order to not risk cleaning obligations after only a few products, we define a minimal batch size *min*. The coating machine should not be started unless there are at least *min* products available in the buffer. While the coating machine is processing products, new products can arrive. These products are added to the current batch. However, the number of products that can be put in one batch is limited, since after a certain amount of products the coating machine has to be cleaned anyway. Therefore, we define another parameter *max*, representing the maximal batch size. As long as the machine has not processed *max* products in one batch, new products should be added to the current batch. Besides the parameters *cap*, *min*, and *max*, for the specification we also need the already introduced parameters *in*, *out*, t_p and t_c.

Observe that during operation, there are two parallel activities going on. On the one hand, new products are stored in the buffer, while on the other hand products are processed by the coating machine. Moreover, these activities influence each

other. That is, a full buffer can only accept a new product after it has sent a product to the coating machine, and the coating of products can start only if there are at least *min* products available in the buffer. Based on this observation, we define the specification, process S, by

$$
\begin{array}{l}
S(in, out : \text{chan}, min, max, cap : \text{nat}, t_p, t_c : \text{real}) = \\
[\![(\sim sync \mapsto \bot) : (m : \text{nat} \mapsto 0) : (n : \text{nat} \mapsto 0) \\
: (bs : \text{nat} \mapsto 0) : (x : \text{nat} \mapsto \bot) : (b : \text{bool} \mapsto \bot) : \lambda_s \\
| \ (\ m < cap \ :\rightarrow in\,?\,x \ ; \ m := m + 1 \\
\qquad\qquad ; \ (bs < max :\rightarrow bs := bs + 1 \ [\!] \ bs \geq max :\rightarrow \varepsilon) \\
\quad [\!] \ bs \geq min :\rightarrow \sim sync\,!\,true \\
\quad [\!] \sim sync\,?\,b \ ; \ (m \leq max :\rightarrow bs := m \ [\!] \ m > max :\rightarrow bs := max) \\
\quad)^* \ ; \ \delta \\
\| \ (\ \sim sync\,?\,b \\
\quad ; (bs > n :\rightarrow m := m - 1 \ ; \ \Delta t_p \ ; \ \pi(out\,!\,0) \ ; \ n := n + 1)^* \\
\quad ; (bs = n :\rightarrow n := 0 \ ; \ \Delta t_c) \\
\quad ; \sim sync\,!\,true \\
\quad)^* \ ; \ \delta \\
]\!].
\end{array}
$$

Process S is a state process with the following channels and programming variables in its state:

- $\sim sync$: an internal channel to synchronize between the two parallel activities,
- m: the number of products in the buffer,
- n: the number of products processed in the current batch,
- bs: the size of the current batch,
- x: a programming variable to receive products,
- b: a programming variable to receive control signals.

The process part of the state process consists of a parallel process. The first argument of the parallel process models the buffering activity. The second argument of the parallel process models the coating activity. Both activities are modelled by infinite repetitions. Regarding the buffering activity, we see that as long as $m < cap$ products can be received. If the size of the current batch is smaller

than the maximum batch size, $bs < max$, then the received product is added to the current batch. If this is not the case, $bs > max$, nothing happens. When the buffering activity detects that the number of buffered products is at least the minimal batch size, $bs \geq min$, then it sends a control signal via $\sim sync$ to the second argument of the parallel process. This argument then starts processing all products currently in the batch. As soon as a complete batch is processed, $bs = n$, the coating activity starts to delay for t_c time units. This models the cleaning of the coating machine. After that, the coating activity notifies the buffering activity via $\sim sync$ that it is ready to process a new batch. Since the buffering activity was running while the machine was being cleaned, new products can have been stored in the buffer. Programming variable m denotes the total number of these products. The products should be processed in the next batch. However, if $m \geq max$, we cannot include all m products in the next batch. Therefore, bs is set to the minimum of m and max ($m \leq max :\rightarrow bs := m \,[\!]\, m > max :\rightarrow bs := max$).

Based on specification S, we design an implementation of the controller. We derive the controller from the specification and consequently they resemble each other quite strongly. Controller C is defined by

$$
\begin{array}{l}
C(get, put, coat, clean : \mathrm{chan}, min, max, cap : \mathrm{nat}) = \\
[\![\ (\sim sync \mapsto \bot) : (m : \mathrm{nat} \mapsto 0) : (n : \mathrm{nat} \mapsto 0) : \\
\quad (bs : \mathrm{nat} \mapsto 0) : (b : \mathrm{bool} \mapsto \bot) : \lambda_s \\
\quad |\ (\ m < cap :\rightarrow (\ get\,!\,true \\
\qquad\qquad\qquad [\!]\ get\,?\,b; m := m + 1 \\
\qquad\qquad\qquad\qquad ; (bs < max :\rightarrow bs := bs + 1 \,[\!]\, bs \geq max :\rightarrow \varepsilon) \\
\qquad\qquad\qquad)\ [\!]\ bs \geq min :\rightarrow \sim sync\,!\,true \\
\qquad [\!]\ \sim sync\,?\,b\ ;\ (m \leq max :\rightarrow bs := m \,[\!]\, m > max :\rightarrow bs := max)\ )^*\ ;\ \delta \\
\quad \|\ (\ \sim sync\,?\,b \\
\qquad ; (bs > n :\rightarrow (put\,!\,true \parallel coat\,!\,true) \\
\qquad\qquad\qquad ; m := m - 1\ ;\ put\,?\,b\ ;\ coat\,?\,b\ ;\ n := n + 1 \\
\qquad)^* \\
\qquad ; (bs = n :\rightarrow n := 0\ ;\ clean\,!\,true\ ;\ clean\,?\,b) \\
\qquad ; \sim sync\,!\,true \\
\qquad)^*\ ;\ \delta \\
]\!].
\end{array}
$$

As can be seen, the only differences with specification S are the places where S communicates with the environment via *in* and *out*, and the parameters t_p and t_c, respectively. At those places the controller communicates with the buffer via *get* and *put*, and with the machine via *coat* and *clean*.

Let $A = \{sa(m,c) \mid m \in \mathit{Channel} \wedge c \in \mathit{Value}\} \cup \{ra(m,x) \mid m \in \mathit{Channel} \wedge x \in \mathit{Var}\}$ be the set of all send actions and receive actions. The implementation I of the coating system as depicted in Figure 8.11 is defined by

$$
\begin{aligned}
&I(\mathit{in}, \mathit{out} : \mathrm{chan}, \mathit{min}, \mathit{max}, \mathit{cap} : \mathrm{nat}, t_p, t_c : \mathrm{real}) = \\
&\pi\partial_A(\ B(\mathit{in}, {\sim}bm, {\sim}cb_{get}, {\sim}cb_{put}) \\
&\quad \parallel M({\sim}bm, \mathit{out}, {\sim}cm_{coat}, {\sim}cm_{clean}, t_p, t_c) \\
&\quad \parallel C({\sim}cb_{get}, {\sim}cb_{put}, {\sim}cm_{coat}, {\sim}cm_{clean}, \mathit{min}, \mathit{max}, \mathit{cap}) \\
&\quad).
\end{aligned}
$$

As mentioned above, our goal is to establish correctness of the coating system with respect to its specification. To be more precise, we analyse for a number of relevant parameter settings if the implementation is equivalent to the specification as far as input, output, and delay behaviour is concerned. To that extent, we also define the environment by the processes E_g and E_e representing a generator and exit, respectively:

$$E_g(\mathit{out} : \mathrm{chan}) = (\mathsf{skip}\ ;\ \mathit{out}\,!\,0 \,[\!]\, \mathsf{skip}\ ;\ \Delta 1)^*\ ;\ \delta,$$

$$E_e(\mathit{in} : \mathrm{chan}) = [\![\ (x : \mathrm{nat} \mapsto \bot) : \lambda_s \mid (\mathit{in}\,?\,x)^*\ ;\ \delta\ ]\!].$$

Process E_g is a simple infinite repetition. During each execution of the repetition, it decides to send a product or to delay for one time unit. The choice between these two options is nondeterministic. Note that by choosing the delay option several times, different delays can occur. Process E_e is an even simpler infinite repetition. During each execution of the loop, it waits until it can receive a product. So, the environment is always able to receive a product from machine M.

Using process E_g and E_e, we can define

$$
\begin{aligned}
&I'(\mathit{min}, \mathit{max}, \mathit{cap} : \mathrm{nat}, t_p, t_c : \mathrm{real}) = \\
&\pi\partial_A(E_g({\sim}eb) \parallel I({\sim}eb, {\sim}me, \mathit{min}, \mathit{max}, \mathit{cap}, t_p, t_c) \parallel E_e({\sim}me)),
\end{aligned}
$$

$$
\begin{aligned}
&S'(\mathit{min}, \mathit{max}, \mathit{cap} : \mathrm{nat}, t_p, t_c : \mathrm{real}) = \\
&\pi\partial_A(E_g({\sim}es) \parallel S({\sim}es, {\sim}se, \mathit{min}, \mathit{max}, \mathit{cap}, t_p, t_c) \parallel E_e({\sim}se)),
\end{aligned}
$$

with $\sim eb$, $\sim me$, $\sim es$, and $\sim se$ channels to connect I' and S to the environment, respectively.

Since both I' and S' contain more detail than just input, output, and delay behaviour, we abstract from behaviour irrelevant for this verification. We define the set

$$\begin{aligned} A' = \{ & aa(x,c) \mid x \in \mathit{Var} \wedge c \in \mathit{Value} \} \\ \cup \{ & \mathit{ca}(m,x,c) \mid m \in \{\sim bm, \sim cb_{get}, \sim cb_{put}, \sim cm_{coat}, \sim cm_{clean}, \} \\ & \wedge x \in \mathit{Var} \\ & \wedge c \in \mathit{Value} \\ \} & \end{aligned}$$

and check for various parameter settings whether

$$\tau_{A'} I'(\mathit{min}, \mathit{max}, \mathit{cap}, t_p, t_c) \leftrightarrows_b \tau_{A'} S'(\mathit{min}, \mathit{max}, \mathit{cap}, t_p, t_c).$$

Recall that $p \leftrightarrows_b q$ denotes that there exists an untimed branching bisimulation between p and q, see discussion at the end of Section 7.7.

Table 8.1 shows the number of states for different settings of the capacity of the buffer with a fixed minimal and maximal batch size and a fixed processing time and cleaning time. For these configurations, we found that indeed

$$\tau_{A'} I'(\mathit{min}, \mathit{max}, \mathit{cap}, t_p, t_c) \leftrightarrows_b \tau_{A'} S'(\mathit{min}, \mathit{max}, \mathit{cap}, t_p, t_c).$$

The minimal batch size, *min*, is set to 4 and the maximal batch size, *max*, is set to 8. The values 4 and 8 are not arbitrary, but real-life settings. The processing times and cleaning times relate to each other as 1:6. So, since the time unit to use is arbitrary, we set t_p to 1 and t_c to 6. The capacity, *cap*, ranges between 3 and 9. Note that for $\mathit{cap} = 3$, the minimal batch size will never be reached. Therefore, no products will be processed. This explains the very small number of states for the $\mathit{cap} = 3$ configuration.

In addition to the functional analysis described above, performance properties of the coating system have been analysed. In particular, we determined the average cycle time and throughput. These properties were determined experimentally by performing simulations.

Verification showed that the implementation is branching bisimilar to the specification. Consequently, simulation of the implementation will produce the same results as simulation of the specification. Since the specification has fewer states than the implementation, it is more efficient to simulate the specification than the

cap	$\tau_{A'}S'(4, 8, cap, 1, 6)$	$\tau_{A'}I'(4, 8, cap, 1, 6)$	minimized
3	70	133	12
4	3164	20737	456
5	3699	26821	577
6	4206	32723	704
7	4685	38443	836
8	5126	43961	902
9	5567	49508	968
10	6008	55055	1034

Table 8.1 · Number of states of $\tau_{A'}S'$ and $\tau_{A'}I'$ for varying buffer capacities.

implementation. Here, we see an interesting application of combining simulation with verification.

In order to simulate realistic situations, we had to adapt the nondeterministic generator process E_g. Instead of nondeterministic arrival times, the new generator process has stochastic arrival times. The new generator process E'_g is defined by

$$
\begin{aligned}
&E'_g(out : \mathrm{chan}) = \\
&[\![(d : \mathrm{dist}[\mathrm{real}] \mapsto exp(1.0)) : (s : \mathrm{real} \mapsto \bot) : \lambda_s \\
&| \ (s := sample(d) \ ; \ \Delta s \ ; \ out \,!\, 0)^* \ ; \ \delta \\
&]\!].
\end{aligned}
$$

As can be seen, E'_g is a state operator process with two programming variables in its state. Programming variable d is a distribution programming variable which is initialised with the value $exp(1.0)$, that is, the exponential distribution with mean value 1.0. Programming variable s is a real programming variable which is used to store samples of distribution d. The main loop of the new generator starts with sampling d. After that, the process delays for the time just sampled. Finally, it sends a product over channel out, after which the main loop is repeated.

We are aware of the fact that by changing the generator, it is unclear whether the verification results can be applied to systems containing the new generator. That is, we verified that the controller satisfies the specification in an environment with a nondeterministic generator, but by changing the environment, it could be possible that the controller operates differently and probably unsatisfactorily. We cannot

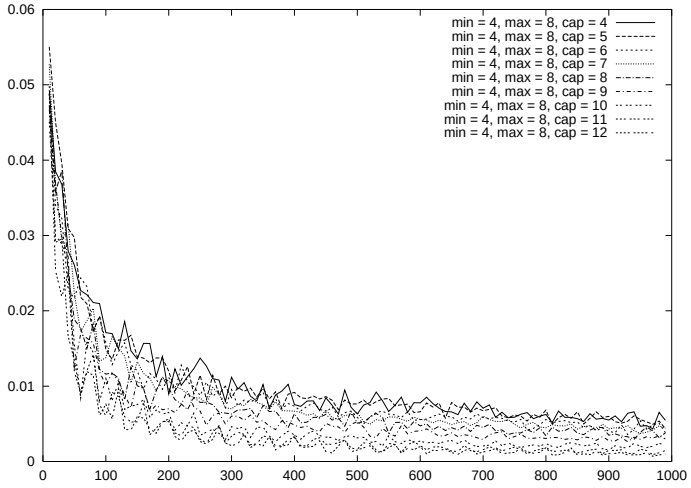
provide formal arguments showing that this will not occur, because stochastic behaviour is not formalised in χ_σ. However, we can make it plausible as follows. As can be seen from the definition of E'_g, only the delay behaviour of the generator is changed. Furthermore, close inspection of the definition of controller C shows that it does not make assumptions about the arrival times of products. Therefore, if it is correct for nondeterministic arrival times, it is correct for stochastic arrival times, too.

The simulation experiments to determine the average throughput of the coating machine and average cycle time of the products are set up as follows. First, a large number of simulations with different parameter settings was performed in order to estimate the minimal number of products that should be produced in order to get reliable simulation results. Next, the performance properties were determined for simulations in which this minimal number of products was produced.

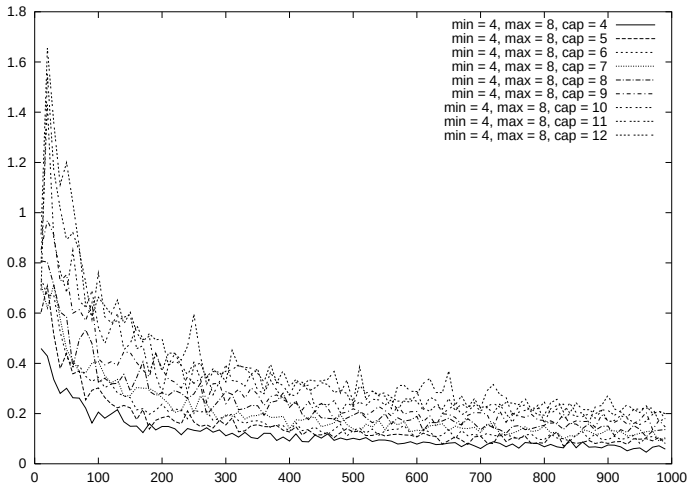
In order to determine the reliability of the simulations, we proceeded as follows. We defined an experiment that simulated the (stochastic version of) the coating machine in which only the buffer capacity was variable. For each buffer capacity, we performed simulations to generate 10, 20, ..., 1000 products. Furthermore, for each number of products, the process was simulated 50 times. For each simulation, the average throughput and cycle time are computed. This resulted in 10.000 numbers, which we call $tp_{i,j}$ and $ct_{i,j}$, where i denotes the number of products, $i \in \{10, 20, \ldots, 1000\}$, and j denotes the sequence number in each series of 50 simulations, $0 \leq j < 50$. After that, for each $i \in \{10, 20, \ldots, 1000\}$ the average cycle time and throughput and the standard deviation of these averages was computed. This resulted in 400 numbers: ct_i (average cycle time), tp_i (average throughput), $\sigma(ct_i)$ (standard deviation average cycle time), and $\sigma(tp_i)$ (standard deviation throughput), for $i \in \{10, 20, \ldots, 1000\}$. Figures 8.12(a) and 8.12(b) show the graphs of $\sigma(ct_i)$ and $\sigma(tp_i)$, respectively.

Since $\sigma(ct_i)$ is the standard deviation of the average of 50 average cycle times, it is a measure for the reliability of the simulation: the smaller the standard deviation, the higher the reliability of the simulation. Similarly, $\sigma(tp_i)$ is a measure for the reliability of the simulation, too. As can be seen in the last two graphs, the reliability of the simulation increases if the number of products increases. However, in the beginning it increases much faster than at the end. That is, the increase in reliability from 10 to 500 products is much higher than the increase in reliability from 500 to 1000 products. We conclude that the reliability does not increase

significantly for simulation runs of more than 500 products. So, simulating 500 products suffices to obtain reliable results.



(a) Standard deviation of the throughput.



(b) Standard deviation of the average cycle time.

Figure 8.12 · Standard deviation of performance properties.

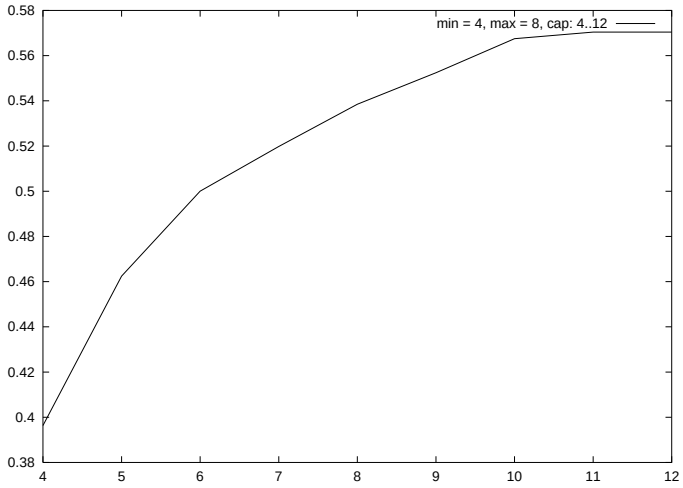
After determining the minimal number of products to produce in order to get reliable simulation results, we computed the average cycle time and throughput for several buffer capacities. The results are depicted in Figures 8.13(a) and 8.13(b). The graphs do not suggest an optimal buffer capacity. If small average cycle times are desired, one should compromise on the throughput. Similarly, if a maximal throughput is desired, one should compromise on the average cycle times. However, since average cycle times range from 4–14 and throughput ranges from 0.4–0.57, a change of buffer capacity has much more effect on the average cycle time than on the throughput. Therefore, it is probably best to minimize the average cycle time by taking a buffer capacity of 4.

We can conclude that we managed to prove that our implementation satisfies the specification for some relevant parameter settings and that both are deadlock free. Furthermore, we can conclude that once the specification has been properly formulated, a controller can be derived from it. Unfortunately, coming up with a proper formulation of the desired specification is still not easy. Finally, by performing several simulation experiments, we analysed the performance of the coating system.

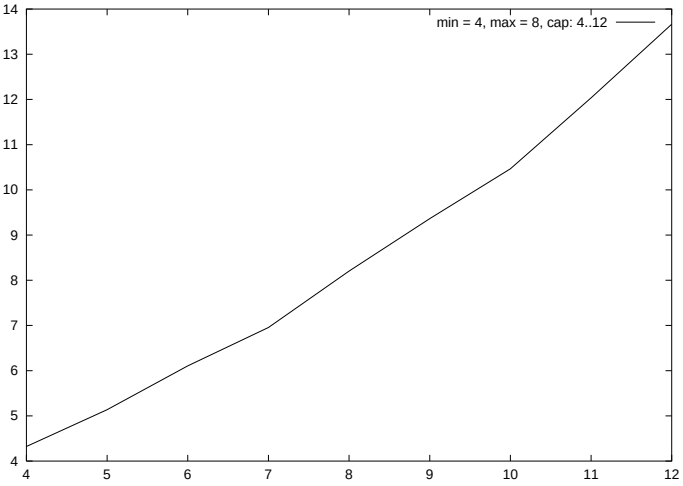
8.9 · A turntable system

The subject of this case study is a turntable with a drill and a testing device. Products are transported by the turntable so that they can be drilled and tested. Testing is necessary because it is possible that drilling went wrong. For example, the drill could break during intensive usage.

The turntable is used for research in the area of (real-time) machine control. A χ model of the turntable exists, that has been analysed with the χ simulator. In this particular case, the emphasis is on functional analysis. As mentioned in Chapter 1, simulation-based analysis is insufficient in this respect. A first attempt to improve the simulation-based analysis can be found in [40]. There we showed that the existing χ model can be translated into Promela and checked by the model checker Spin [109] (alternative 1 of Section 1.3). Even though successful, this method has some drawbacks. First, a χ modeller now has to deal with two formalisms instead of one and has to perform a translation. Second, analysis has to be performed on the translated version, hence on a specification in another language. Furthermore, should you consider the translation scheme from χ to Promela as another definition of χ's semantics (a semantics in terms of Promela constructs), then it



(a) Throughput for different buffer capacities.



(b) Average cycle time for different buffer capacities.

Figure 8.13 · Performance properties of the coating system.

should be possible to compare this alternative semantics with the semantics provided in this thesis and establish some kind of equivalence. Unfortunately, this is

not possible because there is no definite document available describing PROMELA's semantics. Papers like [193, 31, 154] undertake attempts to provide a semantics but have no official status.

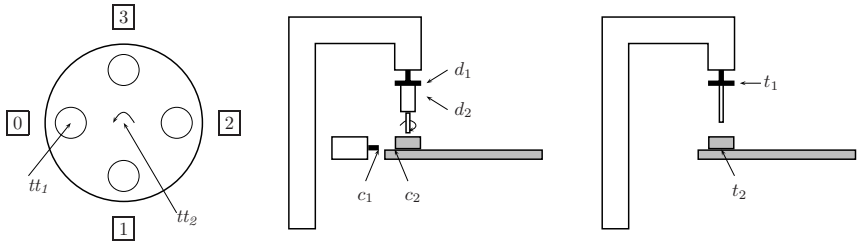
The existing χ model, and hence its PROMELA translation, were not suited for functional analysis because they abstracted from error situations. For example, in reality, the drill can be moved down (to drill a product) without being switched on. Also, the turntable can receive a new product while turning. Obviously, such behaviour is undesired and it is up to the controller to prevent this. However, these situations cannot occur in the χ model. Therefore, using this model, it is impossible to check whether the controller functions properly. In the study presented here, we discuss a χ_σ model that is suited for functional analysis because things *can* go wrong.

The turntable system consists of a round turntable, a clamp, a drill, and a testing device as depicted in Figure 8.14. Figure 8.14(a) depicts the turntable itself. It transports products to the drill and the testing device. It has four slots that can hold a product. Each slot can hold at most one product. A slot can be in one of the following positions: input position (0), drill position (1), test position (2), and output position (3). There are two sensors attached to the turntable: tt_1 and tt_2. If a new product is added at position 0, tt_1 sends a signal. If the turntable has completed a 90° counter-clockwise rotation, tt_2 sends a signal.

Figure 8.14(b) is a schematic view of the clamp and drill located at position 1 of the turntable. Sensors d_1 and d_2 detect whether the drill is in its up or down position, respectively. Note that both sensors are located above the turntable. Therefore, if d_2 signals, this merely means the drill is in its down position and not that the drill succeeded in drilling through a product. The sensors c_1 and c_2 detect whether the clamp is unlocked or locked, respectively.

Figure 8.14(c) schematically depicts the tester. The tester is located at position 2 of the turntable and has two sensors t_1 and t_2. Sensor t_1 detects whether the tester is in its up position. Sensor t_2 detects whether the tester has reached its down position. Unlike sensor d_2, sensor t_2 is located at the surface of the turntable. Therefore, only if the drill drilled completely through the product (or if there is no product at position 2) will t_2 send a signal if the tester has reached its down position.

Table 8.2 defines the controller interface for the turntable system. This is a high-level interface that abstracts from the low-level hardware interface. We chose the



(a) Turntable. (b) Drill and clamp. (c) Tester.

Figure 8.14 · Components of the turntable system.

high-level interface for reasons of simplicity; the same approach to verification can be taken for a low-level interface. The interface consists of commands and sensors for each of the physical components.

The turntable is controlled via the command *turnOn*. It instructs the turntable to rotate 90° counter-clockwise. So, by one rotation, a product can be transported from the input position to the drill position, or from the drill position to the test position, etc. Note that if a product is not removed from position 3, it will be transported to position 1 (the input position).

The clamp, drill, and tester are controlled by *switching* commands. We call them switching commands, since they have two possible effects, which alternate with each invocation. The command *clampOnOff* instructs the clamp to lock if it is unlocked and instructs it to unlock if it is locked. The command *drillOnOff* turns the drill motor on or off and the command *drillUpDown* moves the drill up or down. The tester is also controlled by a switching command. This command, *testUpDown*, either moves the tester up or down.

Before we specify the components of the turntable controller, we first mention some assumptions. For example, it is assumed that the master controller can send requests to the environment to add a new product at position 0. These requests are honoured immediately and confirmed by a signal from sensor tt_1. So, eventually, the turntable enters a cycle in which there are always products at position 1 and 2. Also, we know that if position 0 is empty, a new product is added without delay. Similarly, the master controller can send a request to the environment to remove a product from position 3. Since there is no sensor to detect removal from this position, it is assumed that the environment always honours

Command		
Turntable	*turnOn*	start 90° rotation
Clamp	*clampOnOff*	lock or unlock product
Drill	*drillOnOff*	start or stop drill motor
	drillUpDown	start moving up or down
Tester	*testUpDown*	start moving up or down
Sensor		
Turntable	tt_1	product received at position 0
	tt_2	90° rotation completed
Clamp	c_1	clamp is unlocked
	c_2	clamp is locked
Drill	d_1	drill is up
	d_2	drill is down
Tester	t_1	tester is up
	t_2	tester is down

Table 8.2 · Control interface of the turntable system.

such a request immediately. Consequently, if there is a product at position 3, it is removed without delay. Given these assumptions, we define the initial state of the turntable as follows, position 0: empty, position 1 and 2: not empty, and position 0: empty. Furthermore, we assume that the operations that can be performed at each position still have to be started.

Concerning the tester, we assume that if a good product is tested, the signal 'tester is down' from sensor t_2 will be received in at most 2 time units. Therefore, at most 2 time units after issuing the command *testUpDown*, a signal should have been received. Otherwise, drilling did not succeed. The controller then knows whether the drill succeeded in drilling through the product.

Given these assumptions, we want to verify the following properties:

1. no deadlock,
2. no obsolete remove operations,
3. every product is locked during drilling,
4. the turntable does not rotate when operations are being performed.

The architecture of the resulting system is depicted in Figure 8.15. The physical components, turntable, clamp, drill, and tester, are represented by the processes TT, C, D, and T, respectively. Each physical component has its own controller, TTC, CC, DC, and TC, respectively. The master controller MC is responsible for proper cooperation between these controllers. The environment is modelled by the processes A and R, which add and remove products from the turntable, respectively, and process E which consumes all error products.

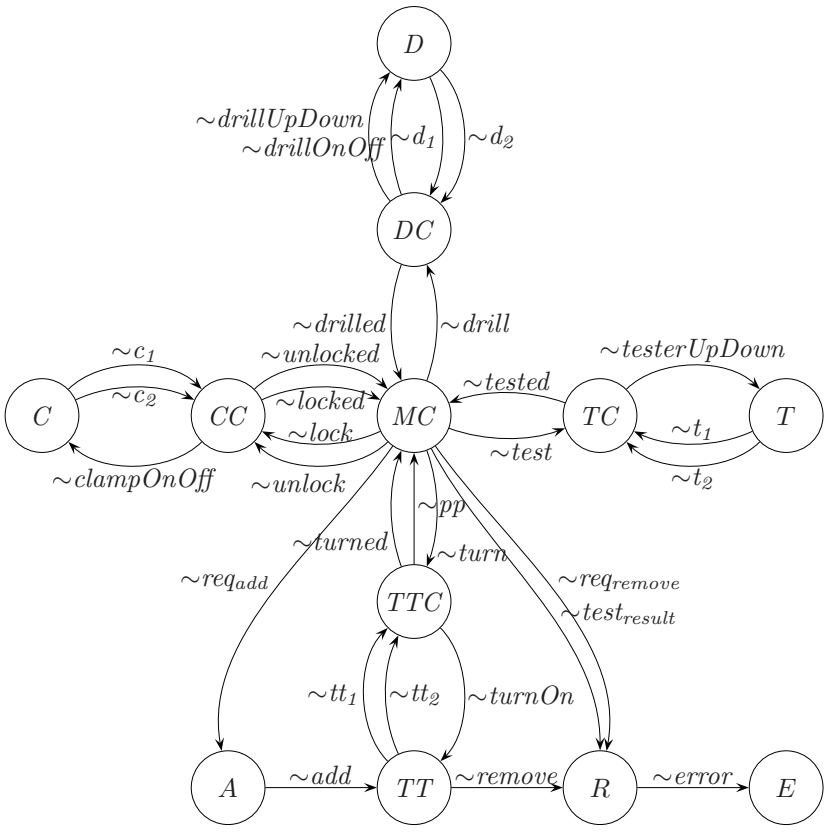


Figure 8.15 • Components of the turntable model.

The turntable and its controller are specified in process TT and TTC, respectively. Process TT is defined by

$$
\begin{array}{l}
TT(tt_1, tt_2, turnOn, add, remove : \mathrm{chan}) = \\
[\![\ (p_0 : \mathrm{bool} \mapsto false) : (p_1 : \mathrm{bool} \mapsto false) : (p_2 : \mathrm{bool} \mapsto false) \\
: (p_3 : \mathrm{bool} \mapsto false) : (x : \mathrm{bool} \mapsto \bot) : (y : \mathrm{bool} \mapsto \bot) : \lambda_s \\
|\ (\ turnOn\,?\,x \\
\quad ;\ \Delta 4 \\
\quad ;\ y := p_3\ ;\ p_3 := p_2\ ;\ p_2 := p_1\ ;\ p_1 := p_0\ ;\ p_0 := y \\
\quad ;\ tt_2\,!\,true \\
\quad)^*\ ;\ \delta \\
\|\ (add\,?\,x\ ;\ (\neg p_0 :\rightarrow p_0 := true)\ ;\ tt_1\,!\,true)^*\ ;\ \delta \\
\|\ (remove\,!\,p_3\ ;\ p_3 := false)^*\ ;\ \delta \\
]\!],
\end{array}
$$

and process TTC is defined by

$$
\begin{array}{l}
TTC(tt_1, tt_2, turnOn, pp, turn, turned : \mathrm{chan}) = \\
[\![\ (x : \mathrm{bool} \mapsto \bot) : \lambda_s \\
|\ (tt_1\,?\,x\ ;\ pp\,!\,true\ ;\ turn\,?\,x\ ;\ turnOn\,!\,true\ ;\ tt_2\,?\,x\ ;\ turned\,!\,true)^*\ ;\ \delta \\
]\!].
\end{array}
$$

In process TT the programming variables p_0, p_1, p_2, and p_3 represent the presence of products at the four positions (*false*: no product, *true*: product present). The turntable executes three infinite repetitions in parallel. First, it is able to rotate 90° if instructed to do so via *turnOn*. Second, a new product can be added. If position 0 is not empty ($\neg p_0$), the turntable deadlocks. Third, a product can be removed. Notice that if position 3 is empty, the value *false* is sent. Process TT needs correct control such that no products are added or removed while the turntable is turning. Also, to prevent deadlock, no products should be added if position 0 is not empty. The turntable controller first waits for a signal from tt_1 and then notifies the master controller via *pp* that position 0 contains a product. Then it waits for a signal (*turn*) to start a rotation via *turnOn*. If a signal via tt_2 is received, the turntable controller confirms that the 90° turn was successful via *turned*.

The clamp and its controller are specified in process C and CC, respectively. Process C is defined by

$$
\begin{array}{l}
C(c_1, c_2, clampOnOff : \mathrm{chan}) = \\
[\![\ (x : \mathrm{bool} \mapsto \bot) : \lambda_s \\
|\ (clampOnOff\,?\,x\ ;\ \Delta 2\ ;\ c_2\,!\,true\ ;\ clampOnOff\,?\,x\ ;\ \Delta 2\ ;\ c_1\,!\,true)^*\ ;\ \delta \\
]\!],
\end{array}
$$

and process CC is defined by

$$
\begin{array}{l}
CC(c_1, c_2, \mathit{clampOnOff}, \mathit{lock}, \mathit{locked}, \mathit{unlock}, \mathit{unlocked} : \mathrm{chan}) = \\
[\![\, (x : \mathrm{bool} \mapsto \bot) : \lambda_s \\
|\ (\ \mathit{lock}\,?\,x\ ;\ \mathit{clampOnOff}\,!\,\mathit{true}\ ;\ c_2\,?\,x\ ;\ \mathit{locked}\,!\,\mathit{true} \\
\quad ;\ \mathit{unlock}\,?\,x\ ;\ \mathit{clampOnOff}\,!\,\mathit{true}\ ;\ c_1\,?\,x\ ;\ \mathit{unlocked}\,!\,\mathit{true} \\
\quad)^*\ ;\ \delta \\
]\!].
\end{array}
$$

Initially, the clamp is unlocked. Signal *clampOnOff* instructs the clamp to lock. This takes 2 time units. If the clamp is locked, it sends a signal to its controller via c_2. Then it waits to be instructed to unlock via signal *clampOnOff*. Unlocking also takes 2 time units and via c_1, the controller is signalled that unlocking succeeded. The clamp controller is instructed by the master controller to lock via *lock*. Then it locks the clamp via *clampOnOff*, waits for confirmation via c_2, and confirms locking to the master controller via *locked*. Next, it waits to be instructed to unlock via *unlock*, unlocks the clamp via *clampOnOff*, waits for confirmation via c_1, and confirms unlocking to the master controller via *unlocked*.

The drill and its controller are specified in process D and DC, respectively. Process D is defined by

$$
\begin{array}{l}
D(d_1, d_2, \mathit{drillOnOff}, \mathit{drillUpDown} : \mathrm{chan}) = \\
[\![\, (x : \mathrm{bool} \mapsto \bot) : \lambda_s \\
|\ (\ (\mathit{drillOnOff}\,?\,x)^*\ ;\ \delta \\
\quad \|\, (\mathit{drillUpDown}\,?\,x\ ;\ \Delta 3\ ;\ d_2\,!\,\mathit{true}\ ;\ \mathit{drillUpDown}\,?\,x\ ;\ \Delta 2\ ;\ d_1\,!\,\mathit{true})^*\ ;\ \delta \\
\quad) \\
]\!],
\end{array}
$$

and process DC is defined by

$$
\begin{array}{l}
DC(d_1, d_2, \mathit{drillOnOff}, \mathit{drillUpDown}, \mathit{drill}, \mathit{drilled} : \mathrm{chan}) = \\
[\![\, (x : \mathrm{bool} \mapsto \bot) : \lambda_s \\
|\ (\ \mathit{drill}\,?\,x\ ;\ \mathit{drillOnOff}\,!\,\mathit{true}\ ;\ \mathit{drillUpDown}\,!\,\mathit{true}\ ;\ d_2\,?\,x \\
\quad ;\ \mathit{drillUpDown}\,!\,\mathit{true}\ ;\ d_1\,?\,x\ ;\ \mathit{drillOnOff}\,!\,\mathit{true}\ ;\ \mathit{drilled}\,!\,\mathit{true} \\
\quad)^*\ ;\ \delta \\
]\!].
\end{array}
$$

The drill can be turned on and off via *drillOnOff* and can be moved up and down via *drillUpDown*. Initially, the drill is turned off and in its up position.

If the drill receives a signal via *drillUpDown* from the drill controller, it moves downwards which takes 3 time units. When it reaches its down position, it sends a signal to its controller via d_2. Then it waits to be instructed to move back up, again via *drillUpDown*. Moving up takes 2 time units and when the up position is reached the controller is signalled via d_1. Like the turntable, also the drilling device needs to be controlled correctly in order to behave as desired. For example, the drill should be turned on before being moved down into a product. The drill controller is instructed to start drilling via *drill*. It then starts a drill session by turning the drill on via *drillOnOff*. The drill is then instructed to move downwards via *drillUpDown*. The controller waits for confirmation via d_2 that the drill has reached its down position. Then the drill is instructed to move up again via *drillUpDown* and confirmation is received via d_1. Finally, the master controller is informed via *drilled* that drilling finished.

The tester and its controller are specified in process T and TC, respectively. Process T is defined by

$$
\begin{array}{l}
T(t_1, t_2, \mathit{testerUpDown} : \mathrm{chan}) = \\
[\![\, (x : \mathrm{bool} \mapsto \bot) : \lambda_s \\
\mid (\ \mathit{testerUpDown}\,?\,x\ ;\ \Delta 2\ ;\ (t_2\,!\,\mathit{true} \,[\!]\, \mathsf{skip}) \\
\quad ;\ \mathit{testerUpDown}\,?\,x\ ;\ \Delta 2\ ;\ t_1\,!\,\mathit{true} \\
\quad)^*\ ;\ \delta \\
]\!],
\end{array}
$$

and process TC is defined by

$$
\begin{array}{l}
TC(t_1, t_2, \mathit{testerUpDown}, \mathit{test}, \mathit{tested} : \mathrm{chan}) = \\
[\![\, (x : \mathrm{bool} \mapsto \bot) : (y : \mathrm{bool} \mapsto \bot) : \lambda_s \\
\mid (\ \mathit{test}\,?\,x \\
\quad ;\ \mathit{testerUpDown}\,!\,\mathit{true}\ ;\ (t_2\,?\,x\ ;\ y := \mathit{true} \,[\!]\, \Delta 2\ ;\ y := \mathit{false}) \\
\quad ;\ \mathit{testerUpDown}\,!\,\mathit{true}\ ;\ t_1\,?\,x \\
\quad ;\ \mathit{tested}\,!\,y \\
\quad)^*\ ;\ \delta \\
]\!].
\end{array}
$$

The tester operates similar to the clamp and the tester controller operates similar to the clamp controller. The only difference is the fact that if a bad product is tested, no confirmation via t_2 is sent. The possible test results are implemented by nondeterministic choice $(t_2\,!\,\mathit{true} \,[\!]\, \mathsf{skip})$. If no confirmation is received, the tester

controller times out after 2 time units and sets programming variable y to *false*. If a good product is tested, a signal is received via t_2 and y is set to *true*. This test result is sent to the master controller via *tested*.

The master controller is specified in process MC. This process is defined by

$$
\begin{aligned}
&MC(turn, turned, pp, lock, locked, unlock, unlocked,\\
&\qquad drill, drilled, test, tested, req_{add}, req_{remove}, test_{result} : \mathrm{chan}) =\\
&[\![(p_0 : \mathrm{bool} \mapsto false) : (p_1 : \mathrm{bool} \mapsto false) : (p_2 : \mathrm{bool} \mapsto false)\\
&: (p_3 : \mathrm{bool} \mapsto false)(x : \mathrm{bool} \mapsto \bot) : (y : \mathrm{bool} \mapsto \bot) : (z : \mathrm{bool} \mapsto \bot) : \lambda_s\\
&| (\ (\ (p_0 :\!\!\rightarrow \varepsilon \,[\![\, \neg p_0 :\!\!\rightarrow req_{add} \,!\, true \ ;\ pp \,?\, x \ ;\ p_0 := true)\\
&\quad \| (\ p_1 :\!\!\rightarrow lock \,!\, true \ ;\ locked \,?\, x\\
&\qquad\qquad ; drill \,!\, true \ ;\ drilled \,?\, x\\
&\qquad\qquad ; unlock \,!\, true \ ;\ unlocked \,?\, x\\
&\qquad [\![\, \neg p_1 :\!\!\rightarrow \varepsilon\\
&\qquad)\\
&\quad \| (p_2 :\!\!\rightarrow test \,!\, true \ ;\ tested \,?\, y \,[\![\, \neg p_2 :\!\!\rightarrow \varepsilon)\\
&\quad \| (p_3 :\!\!\rightarrow req_{remove} \,!\, true \ ;\ p_3 := false \ ;\ test_{result} \,!\, z \,[\![\, \neg p_3 :\!\!\rightarrow \varepsilon)\\
&\quad)\\
&\ ; turn \,!\, true\\
&\ ; turned \,?\, x\\
&\ ; x := p_3 \ ;\ p_3 := p_2 \ ;\ p_2 := p_1 \ ;\ p_1 := p_0 \ ;\ p_0 := x\\
&\ ; (a_3 :\!\!\rightarrow z := y \,[\![\, \neg a_3 :\!\!\rightarrow \varepsilon)\\
&\ )^* \ ;\ \delta\\
&]\!].
\end{aligned}
$$

In process MC, the programming variables p_0, p_1, p_2, and p_3 represent the four product positions at the turntable. Initially, all slots are empty (p_0 through p_3 are set to *false*). If the turntable does not rotate, the following tasks are executed in parallel by the master controller. If there is no product at position 0 (p_0 equals *false*), a request to add a product is sent via req_{add} and the controller waits for confirmation via *pp*. If a product is present at position 1 (p_1 equals *true*), a drill session is started. First, the clamp is locked via *lock* and *locked*, then a product is drilled via *drill* and *drilled*, and finally the clamp is unlocked via *unlock* and *unlocked*. If a product is present at position 2 (p_2 equals *true*), a test session is started via *test* and *untested*. The test result for that product is stored in programming variable y. If a product is present at position 3 (p_3 equals *true*), a request to remove that product is sent via req_{remove} and the test result for that product is sent to the process that removes it via $test_{result}$. Once these tasks have

been performed, the controller instructs the turntable to turn via *turn* and waits for confirmation that a 90° turn has been completed via *turned*. Finally, it updates the values for the programming variables p_0 through p_3 and, if necessary, z which contains the test result for the product now at position 3.

This ends our discussion on the specifications for the turntable components and their controllers. We omit the specifications of the processes A, R, and E. Model checking the specification of the turntable system results in a state space containing 1346 states. We can use abstraction to generate a picture of the state space that only shows the external behaviour of the turntable system (the actions $ca(\sim add, x, true)$, $ca(\sim remove, x, true)$, and $ca(\sim error, x, true)$). This picture is displayed in Figure 8.16.

Looking at Figure 8.16, we see that properties 1 (no deadlock) and 2 (no obsolete remove) are satisfied, because there is no deadlock state and there is no action $ca(\sim remove, x, false)$. Before we consider the other verification properties, we first describe Figure 8.16. Recall that in the initial state, positions 0 and 3 are empty, and positions 1 and 2 contain a product. Furthermore, none of the operations has started yet. The initial state of the displayed graph is state 3. First, a new product is added. Note that no product can be removed. Next, drilling and testing starts. After testing (state 1), the graph splits into a part representing the case that the tested product is properly drilled (left), and a part representing the case that the tested product is not properly drilled (right). After completion of all operations (drilling, testing, and rotating), the system is either in state 6 or in state 16. In these states, new products can be added and processed products can be removed. Finally, note that in case of a negative test result (determined in state 1), an error message is sent.

Where the graph of Figure 8.16 displays the external behaviour of the turntable system, other graphs can be generated that display other behaviour. For example, if we want to check property 3, the following actions are important and can therefore not be abstracted from: $ca(\sim locked, x, true)$, $ca(\sim drill, x, true)$, $ca(\sim drilled, x, true)$, and $ca(\sim unlocked, x, true)$. Figure 8.17 depicts the graph of the system after abstracting from all but these actions. As can be seen, the pair of actions $ca(\sim drill, x, true)$ and $ca(\sim drilled, x, true)$ is enclosed by the pair $ca(\sim locked, x, true)$ and $ca(\sim unlock, x, true)$. Therefore, a product is always locked while being drilled.

Property 4 can be verified similarly. We do not provide details but suffice by mentioning the property is satisfied.

We conclude that in χ_σ one can specify and analyse the turntable system. It improves the χ specification in a sense that now the model can be verified directly. In case of the χ model, only validation by means of simulation was possible, or translation to another formalism was required.

8.10 · Discussion

The smaller examples showed specific features of χ_σ. These features, like time factorisation and communication, are results of combinations of process operators. In addition, we illustrated strong and branching bisimulation equivalence by means of examples. In order to show the practical value of χ_σ, we performed several case studies. Both functional and performance properties were analysed. Furthermore, the case studies show that the features illustrated earlier by some small examples do occur in specifications of real-life systems. Therefore, it is beneficial to understand them properly.

This chapter shows that performance analysis as possible with χ can also be done with χ_σ. Moreover, it also shows that χ_σ enables functional analysis. The approach we have chosen for functional analysis is called model checking. Of all formal techniques, model checking is among the most successful and has been applied to many industrial case studies.

We realize that the number and the size of the case studies performed in this chapter are insufficient to conclude that χ_σ is a *practical* formal method. However, they confirm our believe that χ_σ has the potential to become one.

Besides more and larger case studies, some more deficiencies need to be addressed. For instance, the case study of the coating system shows that a formal treatment of distributions is desired. Also, branching bisimulation should be incorporated in our mathematical framework.

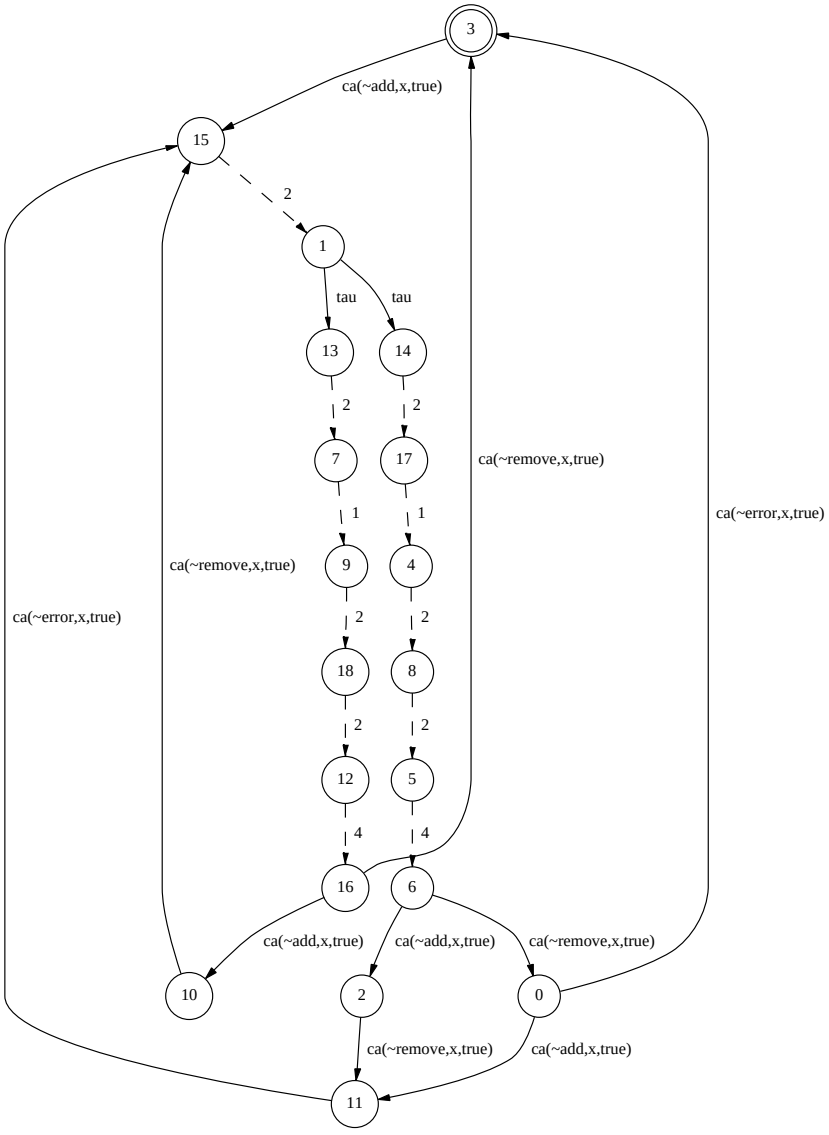


Figure 8.16 · External behaviour of the turntable system.

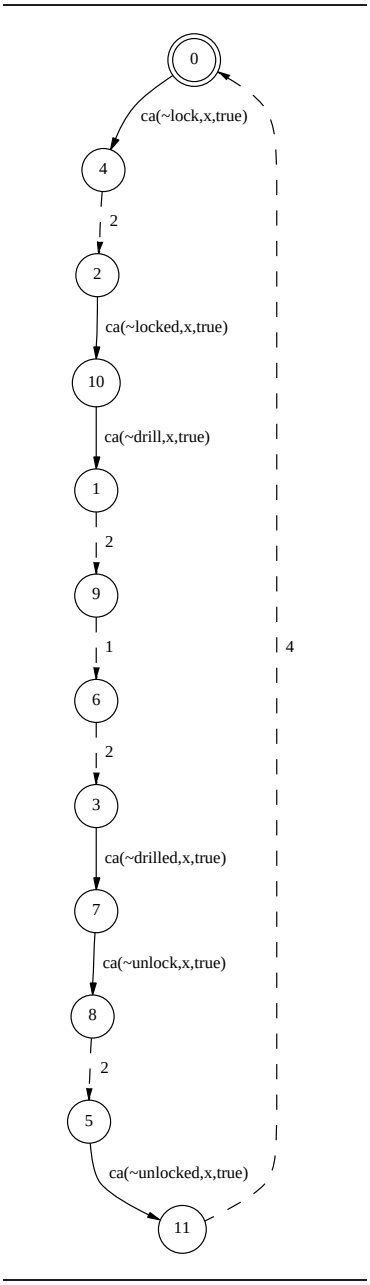


Figure 8.17 · Every product is locked during drilling.

Conclusions · 9

In this chapter, we draw conclusions regarding the work presented in this thesis. First, we draw general conclusions regarding formal specification and analysis of industrial systems. These are followed by conclusions regarding the engineering language χ and the formal language χ_σ. Finally, we discuss opportunities for further research.

Existing analysis techniques are powerful with respect to performance analysis, but lack proper support for functional analysis. We suggested that formal methods are good candidates for improvements in that direction. This lead to Research topic 1: *Is it possible to improve specification and analysis techniques for industrial systems by means of formal methods?* Regarding this topic, we conclude that existing analysis techniques can be improved with respect to functional analysis using formal methods.

Integration of simulation techniques with formal methods techniques will enable both performance and functional analysis. This lead to Research topic 2: *Is it possible to integrate formal methods with existing simulation techniques?* Regarding this topic we conclude that integration is possible indeed. Moreover, performance analysis can be done more efficiently by using results obtained from functional analysis. This is illustrated by the case study presented in Section 8.8.

As discussed in Section 1.3, our choice to integrate simulation techniques with formal methods is to formalise an engineering language. We decided to formalise χ, because it is a good representative. This lead to Research topic 3: *Is it possible to convert χ into a formal method?* With respect to this topic, we are interested mainly in formal theories, rather than the languages based on these theories. Research in the field of Formal Methods has produced various theories to define formal languages. Well-known examples are Algebraic Specification (AS) and Structural Operational Semantics (SOS). Looking at these theories, we notice that each one has its own specific purpose. For example, AS is well suited to define

(countable) data types, and SOS is well suited to define dynamic behaviour. Formalising a whole engineering language requires a combination of such theories. Unfortunately, we find that little is known on how to combine AS and SOS (see for example Section 4.16). Furthermore, for some common aspects of engineering languages (like real number arithmetic), to our knowledge it is unknown how to achieve rigorous formalisation at all (see Section 5.4).

Recall that we restricted ourselves to the basis of χ, called discrete χ. First of all, we can conclude that we managed to completely formalise this language except for the real numbers and the probabilistic constructs. This resulted in the formal method χ_σ consisting of a formal syntax and semantics, a mathematical framework, and tool support. Developing a formal method requires an integral approach towards these three aspects since they influence each other. For example, the tools have proved themselves valuable during the development of the semantics of χ_σ; only after several experiments did we understand how certain operators, like the ' ; ' and the '$\|$' operator, should be defined.

We believe χ_σ has the potential to be a *practical* formal method. Firstly, χ_σ resembles discrete χ closely (see Chapters 5 and 6). To perform formal analysis, engineers using discrete χ need not learn another language. Secondly, in Chapter 7, we established results describing a property preserving reduction from infinite to finite transition systems. This reduction applies to all (χ_σ translations of) discrete χ specifications. With respect to tool support, this is of great relevance. Thirdly, we were able to implement prototype tools to simulate and model check χ_σ specifications. Of all formal techniques, model checking is among the most successful. Furthermore, much research on model checking aims to improve practical applicability by including, for instance, probabilistic and continuous features. Finally, the case studies conducted in Chapter 8 show that in χ_σ, formal analysis can be combined with existing analysis techniques.

We believe χ_σ improves upon discrete χ in several ways. Firstly, χ_σ enables mathematical reasoning. Secondly, χ_σ has notions of equivalence and of abstraction. These notions enable equivalence checks and specification-implementation checks. Thirdly, χ_σ has a maximal progress operator. Using this operator, it is possible to distinguish delayable and urgent actions (see Section 5.2). This distinction can be useful to model industrial systems (see Section 5.2). Finally, the constructs of the language are more orthogonal with respect to each other. For example, the parallel composition operator and the state operator can be mixed freely with other process operators, and guarded command statements and selective waiting

statements are no longer separate constructs, but can be constructed from simpler constructs.

Besides improvements, χ_σ lacks several features of discrete χ that are useful to model industrial systems. The most important missing features are probabilistic constructs (like distributions) and a formal treatment of the real numbers. In Chapter 5, we discussed why these features are not included in χ_σ.

We conclude this chapter by indicating directions for further research. As mentioned above, we believe that χ_σ has the potential to be a practical formal method. To become a practical formal method, future research on χ_σ should concern theory development, tool design, and case studies. For instance, theory development should result in the definition of timed branching bisimulation on χ_σ processes, and tool design should result in robust and efficient implementations rather than prototype implementations. In addition, more and larger case studies should be conducted.

As mentioned in Section 1.4, eventually χ_σ should replace discrete χ. This is not yet possible and therefore further research in this direction is needed. In particular, solutions have to be found to incorporate real number arithmetic and probabilistic constructs. So, discrete χ cannot yet be replaced. In this context, it is interesting to remark that currently new discrete χ tools have been developed that use χ_σ internally. In fact, these tools implement an automatic translation from discrete χ into χ_σ.

Further research should also investigate opportunities to extend χ_σ with continuous behaviour. In particular, we hint at the inclusion of differential algebraic equations. If this succeeds, χ_σ can replace hybrid χ (discrete χ with extensions to describe continuous behaviour).

Bibliography

[1] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In Bergstra et al. [28], chapter 3, pages 197–292.

[2] W.T.M. Alberts and G. Naumoski. *A Discrete-Event Simulator for Systems Engineering*. PhD thesis, Technische Universiteit Eindhoven, 1998.

[3] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Annual on Logic in Computer Science*, pages 414–425, Philadelphia, 1990. IEEE Computer Society Press.

[4] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[6] M. Andersson. *Object-oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, 1994.

[7] S. Andova. *Probabilistic Process Algebra*. PhD thesis, Technische Universiteit Eindhoven, 2002. To appear.

[8] N.W.A. Arends. *A Systems Engineering Specification Formalism*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[9] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification*. Springer, 1999.

[10] J. Backus. The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. In *Proceedings ICIP*, pages 125–131. Unesco, 1960.

[11] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.

[12] J.C.M. Baeten and J.A. Bergstra. Recursive process definitions with the state operator. *Theoretical Computer Science*, 82(2):285–302, 1991.

[13] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.

[14] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, IX(2):127–168, 1986.

[15] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. Axiomatizing probabilistic processes: ACP with generative probabilities. *Information and Computation*, 121(2):234–255, September 1995.

[16] J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: Real time and discrete time. In Bergstra et al. [28], chapter 10, pages 627–684.

[17] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *Proceedings CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 477–492, Hildesheim, 1993. Springer Verlag.

[18] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, Semantic Modelling, pages 149–268. Oxford University Press, 1995.

[19] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[20] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, July/August 1982.

[21] P.I. Barton. *The Modelling and Simulation of Combined Discrete/Continuous Processes*. PhD thesis, University of London, 1998.

[22] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

[23] T. Basten and D. Bošnački. Enhancing partial-order reduction via process clustering. In *16th IEEE Conference on Automated Software Engineering (ASE 2001)*. IEEE Computer Society Press, 2001.

[24] David M. Beazley. *Python Essential Reference*. New Riders, 2000.

[25] D.A. van Beek, A. van den Ham, and J.E. Rooda. Modelling and control of process industry batch production systems. In *Proceedings of the 2002 IFAC World Congress*, Barcelona, 2002. To appear.

[26] D.A. van Beek and J.E. Rooda. Languages and applications in hybrid modelling and simulation: Positioning of χ. *Control Engineering Practice*, 8(1):81–91, 2000.

[27] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*, ACM Press Frontier Series. ACM Press, 1989.

[28] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science B.V., 2001.

[29] G. Berry. The foundations of Esterel. In G.D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[30] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[31] W.R. Bevier. Towards an operational semantics of Promela in ACL2. In *Proceedings of the 3rd* Spin *Workshop*, The Netherlands, April 1997.

[32] R. Bird and P. Wadler. *Introduction to Functional Programming using Haskell*. Prentice Hall, London, 1998. 2nd ed.

[33] G. Birtwistle. *Demos – Discrete Event Modeling on Simula*. MacMillan, New York, 1979.

[34] G. Birtwistle and C.M.N. Tofts. An operational semantics of process-oriented simulation languages: Part I μDemos. *Transactions of the Society for Computer Simulation*, 10(4):299–333, 1993.

[35] G. Birtwistle and C.M.N. Tofts. An operational semantics of process-oriented simulation languages: Part II μDemos. *Transactions of the Society for Computer Simulation*, 11(4):303–336, 1994.

[36] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 229–239, San Diego, California, January 1988.

[37] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, Asilomar, 1996.

[38] V. Bos and A.T. Hofkamp. The *chipy* project. Online document: `http://se.wtb.tue.nl/`, 2002. To appear.

[39] V. Bos and J.J.T. Kleijn. Formalisation of a production systems modelling language: the operational semantics of χ Core. *Fundamenta Informaticae*, 41(4):367–392, 2000.

[40] V. Bos and J.J.T. Kleijn. Automatic verification of a production system. *Robotics and Computer-Integrated Manufacturing*, 17(3):185–198, 2001.

[41] V. Bos and J.J.T. Kleijn. Formal specification and analysis of production systems. In *Proceedings of the 16th International Conference on Production Research*, Prague, Czech Republic, 2001. ICPR-16.

[42] D. Bošnački and D. Dams. Discrete-Time Promela and Spin. In A.P. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 307–310. Springer-Verlag, 1998.

[43] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer-Verlag, 1997.

[44] D. Bošnački. *Enhancing State Space Reduction Techniques for Model Checking*. PhD thesis, Technische Universiteit Eindhoven, 2001.

[45] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new ASF+SDF meta-environment. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing*. Springer-Verlag, 1997.

[46] E. Brinksma, H. Hermanns, and J-P. Katoen, editors. *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*. Springer, July 2000.

[47] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[48] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), August 1986.

[49] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

[50] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 2:142–170, 1998.

[51] E.J.J. van Campen. *Design of a Multi-Process Multi-Product Wafer Fab*. PhD thesis, Technische Universiteit Eindhoven, 2000.

[52] F.E. Cellier. *Continuous System Modeling*. Springer-Verlag, 1991. ISBN 0-387-97502-0.

[53] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[54] E.M. Clarke, O. Grumberg, H. Hiraishim, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[55] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999. Second printing, 2000.

[56] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude as a metalanguage. In *2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998.

[57] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Towards Maude 2.0. In *3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[58] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98*, Sumazu, Japan, April 1998. CafeOBJ project.

[59] P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, Department of Computer Science, University of Twente, November 1999.

[60] J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall Series in Computer Science. Prentice Hall, 1996.

[61] J. Davies and S. Schneider. A brief history of timed CSP. Technical Report PRG-96, Programming Research Group, Oxford University, April 1992.

[62] G. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III, Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[63] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.

[64] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of Workshop on Automatic Verification Methods for Finite-State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, June 1989.

[65] D.L. Dill and E.M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEEE Proceedings*, Part E 133(5), 1986.

[66] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. Elsevier Science Publishers B.V., Amsterdam, 1989. Result of the ESPRIT/SEDOS Project.

[67] H. Elmqvist. Dymola – dynamic modeling language – user's manual. Technical report, Dynasim AB, Lund, Sweden, 1994.

[68] G. Fábián. *A Language and Simulator for Hybrid Systems*. PhD thesis, Technische Universiteit Eindhoven, 1999.

[69] J.J.H. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. PhD thesis, Technische Universiteit Eindhoven, 2000.

[70] C. Fidge and J. Žic. A simple, expressive real-time CCS. Technical Report 94-27, Software Verification Reseach Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, December 1994.

[71] W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer-Verlag, Berlin, 2000.

[72] W.J. Fokkink and C. Verhoef. A conservative look at operational semantics with variable binding. *Information and Computation*, 146(1):24–54, October 1998.

[73] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment. In *1st IEEE Intl. Conf. of Formal Engineering Methods*, pages 170–181, 1997.

[74] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software–Practice and Experience*, 30(11):1203–1233, September 2000.

[75] H. Garavel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, and B. Vivien. CADP'97 - Status, Applications, and Perspectives. In I. Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design*, Zagreb, Croatia, June 1997.

[76] H. Garavel and M. Sighireanu. Towards a second generation of formal description techniques — rationale for the design of E-LOTOS. In J.F. Groote, B. Luttik, and J. van Wamel, editors, *Third International Workshop on Formal Methods for Industrial Critical Systems*, pages 187–230, CWI Amsterdam, The Netherlands, May 1998.

[77] R.J. van Glabbeek. What is branching time and why to use it? In *The Concurrency Column, Bulletin of the EATCS*, volume 53, pages 190–198, 1994.

[78] R.J. van Glabbeek, S.A. Smolka, B. Steffen, and C.M.N. Tofts. Reactive, generative and stratified models of probabilistic processes. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 130–141, Philadelphia, 1990.

[79] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[80] S. Gnesi and D. Latella. Report on the first workshop and meeting of the ERCIM working group on formal methods for industrial critical systems. Technical Report 59, European Association for Theoretical Computer Science—EATCS, Oxford, UK, March 1996. Appeared as a special issue of *Formal Methods in System Design*, 12(2), March 1998.

[81] Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors. *5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems, Proceedings of FMICS'2000*, volume REP-FOKUS-2000-91, Berlin, April 2000.

[82] P. Godefroid. Using partial orders to improve automatic verification techniques. In *Proceedings of the 2nd Workshop on Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, 1990.

[83] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 5th Workshop on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer, 1993.

[84] J.A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1996.

[85] J.A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, November 1992.

[86] M.J.C. Gordon. *Introduction to HOL*. Cambridge University Press, 1993.

[87] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In L. Donateillo and R. Nelson, editors, *Performance Evaluation of Computer and Communication Systems*, volume 729 of *Lecture Notes in Computer Science*, pages 121–146. Springer-Verlag, 1993.

[88] J.A. Govaarts. Efficiency in a lean assembly line: A case study at NedCar Born. Master's thesis, Stan Ackermans Institute, Eindhoven, October 1997.

[89] J.F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 18(2):263–299, 1993.

[90] J.F. Groote. The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997.

[91] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Technical Report SEN-R0117, CWI, Amsterdam, 2001.

[92] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing, pages 231–250, The Netherlands, 1994. Springer-Verlag.

[93] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *ACP: Algebra of Communicating Processes*, Workshops in Computing, pages 26–62, Utrecht, The Netherlands, 1995. Springer-Verlag.

[94] J.F. Groote and J. Springintveld. Focus points and convergent process operators: A proof strategy for protocol verification. Logic Group Preprint Series 142, Utrecht Research Institute for Philosophy, October 1995.

[95] J.F. Groote and J. Springintveld. Focus points and convergent process operators: a proof strategy for protocol verification. *The Journal of Logic and Algebraic Programming*, 49:31–60, 2001.

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

[97] P.A.M. Haagh, A.U. Wilkens, H.J.A. Rulkens, E.J.J. van Campen, and J.E. Rooda. Application of a layout design method to the dielectric decomposition area in a 300 mm wafer fab. In *Proceedings of the Seventh International Symposium on Semiconductor Manufacturing*, pages 69–72, Tokyo, Japan, October 1998. ISSM, Ultra Clean Society.

[98] H. Hansson. *Time and probability in formal design of distributed systems*. PhD thesis, University of Uppsala, 1991. Published in Real-Time Safety Critical Systems, Vol. 1, Elsevier, 1994.

[99] H. Hansson and B. Jonsson. A calculus for communicating systems with time and probabilities. In *Proceedings IEEE Real-Time Systems Symposium*, pages 278–287. IEEE, Computer Society Press, 1990.

[100] K.M. van Hee, L.J. Somers, and M. Voorhoeve. The EXSPECT tool. In S. Prehn and W.J. Toetenel, editors, *VDM'91 - Formal Software Development*, volume 551 of *Lecture Notes in Computer Science*, pages 683–684, Noordwijkerhout, The Netherlands, October 1991. Springer.

[101] M. Hennessy. *Algebraic Theory of Processes*. MIT Press Series in the Foundations of Computing. MIT Press, 1988.

[102] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 278–292, 1996.

[103] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[104] J. Hillston. *A compositional approach to performance modelling*. Cambridge University Press, 1996.

[105] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[106] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[107] A.T. Hofkamp. *Reactive Machine Control: a Simulation Approach using χ*. PhD thesis, Technische Universiteit Eindhoven, 2001.

[108] G.J. Holzmann. *Design and validation of computer protocols*. Prentice Hall International, London, 1991.

[109] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. (See also the SPIN homepage: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`).

[110] W.J. Hopp and M.L. Spearman. *Factory Physics, Foundations of Manufacturing Management*. Irwin, 1996.

[111] ITU-TS. Message sequence chart (MSC). Recommendation Z.120, ITU-TS, Geneva, 2000.

[112] H.E Jensen, K.G. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. In Rutgers, editor, *Proceedings of the 2nd SPIN Workshop*, University, New Jersey, USA, August 1996.

[113] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall Series in Computer Science. Prentice Hall, 1991. 2nd ed.

[114] B. Jonsson, W. Yi, and K.G. Larsen. Probabilistic extensions of process algebras. In Bergstra et al. [28], chapter 11, pages 685–710.

[115] J-P. Katoen, D. Latella, R. Langerak, E. Brinksma, and T. Bolognesi. A consistent causality-based view on a timed process algebra including urgent interactions. *Journal of Formal Methods in System Design*, 12(2):189–216, 1998.

[116] D.L. Kettenis. COSMOS: A simulation language for continuous, discrete and combined models. *Simulation*, 58:32–41, 1992.

[117] D.L. Kettenis. *Issues of Parallelization in Implementation of the Combined Simulation Language COSMOS*. PhD thesis, Delft University of Technology, 1994.

[118] B. Khoshnevis. *Discrete Systems Simulation*. McGraw-Hill, 1994.

[119] J.J.T. Kleijn, M.A. Reniers, and J.E. Rooda. A process algebra based verification of a production system. In J. Staples, M.G. Hinchley, and S. Liu, editors, *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 90–99, Brisbane, Australia, December 1998. IEEE Computer Society.

[120] J.J.T. Kleijn, M.A. Reniers, and J.E. Rooda. Analysis of an industrial system. *Formal Methods in System Design*, 2002. To appear.

[121] J.J.T. Kleijn and J.E. Rooda. χ manual. Online document: `http://se.wtb.tue.nl/documentation/`, 2001.

[122] J.W. Klop. Term rewriting systems. In S. Abramsky, Dov. M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2 (Background: Computational Structures). Clarendon Press, Oxford, 1992.

[123] A.S. Klusener. *Models and axioms for a fragment of real time process algebra*. PhD thesis, Technische Universiteit Eindhoven, 1993.

[124] Eleftherios Koutsofios and Stephen C. North. Drawing graphs with *dot*. Online document: `http://www.research.att.com/sw/tools/graphviz/`, 1999.

[125] I.A. van Langevelde. A compact file format for labeled transition systems. Technical Report SEN-R0102, ISSN 1386-369X, CWI, 2001.

[126] K.G. Larsen, P. Pettersson, and W Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[127] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1992.

[128] H. Lönn and P Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.

[129] G. Lowe. Probabilistic and prioritized models of timed csp. *Theoretical Computer Science*, 135:315–352, 1995.

[130] Mark Lutz. *Programming Python*. O'Reilly & Associates, first edition, October 1996.

[131] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[132] N.A. Lynch, R. Segala, F.W. Vaandrager, and H.B. Weinberg. Hybrid i/o automata. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer-Verlag, December 1995.

[133] Zohar Manna and the STeP group. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.

[134] Zohar Manna and the STeP group. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In *8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, July 1996.

[135] S.E. Mattsson, H. Elmqvist, and J.F. Broenink. Physical system modeling with Modelica. *Control Engineering Practise*, 6:501–510, 1998.

[136] S. Mauw and G.J. Veltink. *Algebraic specification of communication protocols*. Number 36 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[137] J. Meseguer. Conditional rewriting logic as unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[138] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[139] C.A. Middelburg. Variable binding operators in transition system specifications. *Journal of Logic and Algebraic Programming*, 47(1):15–45, 2001.

[140] C. Miguel, A. Fernández, and L. Vidaller. LOTOS extended with probabilistic behaviours. *Formal Aspects of Computing*, 5(3):253–281, 1993.

[141] R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science*, volume 92. Springer-Verlag, 1980.

[142] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, July 1983.

[143] R. Milner. *Communication and Concurrency*. Prentice-Hall International, London, UK, 1989.

[144] B. Mishra and E.M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269–291, 1985.

[145] E. Mitchell and J. Gauthier. Advanced continuous simulation language. *Simulation*, 26(3):72–78, 1976.

[146] A. Mitschele-Thiel. *Systems Engneering with SDL: Developing Performance-Critical Communication Systems*. J.W. Wiley, 2001. ISBN 0-471-49875-0.

[147] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Online document: `http://www.dcs.ed.ac.uk/home/cwb/`.

[148] J.M. van de Mortel-Fronczak. Operational semantics of χ. Technical Report WPA 420062, Technische Universiteit Eindhoven, 1995.

[149] J.M. van de Mortel-Fronczak, R.J.A. Gorter, and J.E. Rooda. A case study in simulation-based system specification design. In *Proceedings of ESM'2000*, pages 232–238, Ghent, Belgium, May 2000.

[150] J.M. van de Mortel-Fronczak and J.E. Rooda. Application of concurrent programming to specification of industrial systems. In *Proceedings of INCOM'95*, pages 421–426, Beijing, China, October 1995.

[151] J.M. van de Mortel-Fronczak and J.E. Rooda. Heterarchical control systems for production cells – a case study. In *Proceedings of MIM'97*, pages 243–248, Vienna, Austria, February 1997.

[152] J.M. van de Mortel-Fronczak, J.E. Rooda, and N.J.M. van den Nieuwelaar. Specification of a flexible manufacturing system using concurrent programming. *The International Journal of Concurrent Engineering: Research & Applications*, 3(3):187–194, September 1995.

[153] J.M. van de Mortel-Fronczak, J.P.M. Schmitz, and J.E. Rooda. Experimental comparison of control architectures. In Kai Mertins, Olivier Krause, and Burkhard Schallock, editors, *Proceedings of APMS'99*, pages 530–537, Berlin, Germany, September 1999.

[154] V. Natarajan and G.J. Holzmann. Outline for an Operational-Semantics of Promela. Technical report, Bell Laboratories, 1996.

[155] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[156] M. Nivat and J.C. Reynolds, editors. *Algebraic methods in semantics*. Cambridge University Press, 1985.

[157] R. Overwater. *Processes and Interaction, An Approach to the Modelling of Industrial Systems*. PhD thesis, Technische Universiteit Eindhoven, 1987.

[158] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag.

[159] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings 5th GI (Gesellschaft für Informatik) Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Karlsruhe, Germany, 1981.

[160] C. Pegden, R. Shannon, and R. Sadowski. *Introduction to Simulation using SIMAN*. McGraw-Hill, 1995.

[161] D.A. Peled. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8(1):39–64, 1996.

[162] C.A. Petri. Kommunikation mit Automaten. Schriften des IIM nr. 2, Institut für Instrumentelle Mathematik, Bonn, 1962. English translation available as *Communication with Automata*, Tech. Report RADC-TR-65-377, Vol. 1, Suppl. 1, Applied Data Research, Princeton, NJ 1966.

[163] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Århus University, Computer Science Department, 1981.

[164] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, November 1977. IEEE.

[165] N. Rajan, S. Shankar and M.K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Proceedings of the 1995 Workshop on Computyer-Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, June 1995.

[166] W. Reisig. *Petri Nets - An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[167] M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. PhD thesis, Technische Universiteit Eindhoven, 1998.

[168] Annie Ressouche, Robert de Simone, Amar Bouali, and Valérie Roy. *The* FCTOOLS *User Manual*. INRIA Sophia Antipolis/EN-SMP-CMA, Sophia Antipolis. Online document: `http://www-sop.inria.fr/meije/verification/index.html`.

[169] Markus Roggenbach, Lutz Schröder, and Till Mossakowski. Specifying real numbers in CASL. In Christine Choppy and Didier Bert, editors, *Recent Developments in Algebraic Development Techniques, 14th International Workshop, WADT'99*, volume 1827 of *Lecture Notes in Computer Science*. Springer, 2000.

[170] J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. PhD thesis, Universiteit Twente, 1999.

[171] J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001. Special issue on FMICS'99.

[172] J.E. Rooda. Discrete event simulation for the design and operation of grain terminals. *Int. Journal of Modelling and Simulation*, 1:246–249, 1981.

[173] J.E. Rooda. Simulation of logistics elements. Technical Report 34, Technische Hogeschool Twente, The Netherlands, September 1982. User manual.

[174] J.E. Rooda. Transport- en produktiesystemen, modelbouw en simulatie. *Transport & Opslag*, 6:30–35, June 1982.

[175] J.E. Rooda. The modelling of industrial systems. Lecture notes. Online document: `http://se.wtb.tue.nl/documentation/`, 2000.

[176] J.E. Rooda, P.F. Jansen, and M.E.A. Striekwold. Analyse des Verteiler-Systems einer Auktionszentrale für Blumen. *Fördern und Heben*, 32(10), October 1982.

[177] J.E. Rooda and N. van den Schilden. Simulation of maritime transport and distribution by sea-going barges: An application of multiple regression analysis and factor screening. *The International Journal of Storing and Handling Bulk Materials*, 2(4):813–824, December 1982.

[178] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[179] H.J.A. Rulkens, E.J.J. van Campen, J. van Herk, and J.E. Rooda. Batch size optimization of a furnace and pre-clean area by using dynamic simulations. In *Proceedings of the Advanced Semiconductor Manufacturing Conference*, pages 439–444, Boston, September 1998. ASMC.

[180] J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-01, SRI International, Menlo Park CA 94024 USA, March 1995.

[181] J. Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of* SPIN *Model Checking: 5th and 6th International* SPIN *Workshops*, volume 1680 of *Lecture Notes in Computer Science*, Trento, Italy, and Toulouse, France, July and September 1999. Springer-Verlag.

[182] S.A. Schneider. An operational semantics for timed CSP. In *Proceedings Chalmers Workshop on Concurrency, 1992*, pages 428–456. Report PMG-R63, Chalmers University of Technology and University of Göteborg, 1992.

[183] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, 1995.

[184] R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[185] N. Shankar. PVS: Combining specification, proof checking, and model checking. In M.K. Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, November 1996. Springer-Verlag.

[186] R.W. Sierenberg and O.B. de Gans. Personal Prosim: A fully integrated simulation environment. In W. Krug and A. Lehmann, editors, *ESS-92: Proc. of the 1992 European Simulation Symposium*, pages 167–173. Society for Computer Simulation, San Diego, 1992.

[187] R. De Simone. Higher level synchronizing devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[188] C.M.N. Tofts and G. Birtwistle. A denotational semantics for a process-based simulation language. *Modeling and Computer Simulation*, 8(3):281–305, 1998.

[189] Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18:25–68, 2001.

[190] Kenneth J. Turner, editor. *Using Formal Description Techniques, An Introduction to Estelle, Lotos and SDL*. Wiley Series in Communication and Distributed Systems. John Wiley & Sons, 1993.

[191] Irek Ulidowski and Shoji Yuen. Extending process languages with time. In M. Johnson, editor, *6th International Conference on Algebraic Methodology and Software Technology AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 524–538, Sydney, Australia, 1997. Springer.

[192] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2(2):274–302, 1995.

[193] C. Weise. An incremental formal semantics for PROMELA. In *Proceedings of the 3rd* SPIN *Workshop*, The Netherlands, April 1997.

[194] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2), 1997.

[195] B.P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, London, 1984.

[196] B.P. Zeigler, H. Praehofer, and T.G. Kim. *Theory of Modeling and Simulation.* Academic Press Ltd., London, U.K., 2000.

[197] The BCG (binary coded graphs) manual pages. Online document: `http://www.inrialpes.fr/vasy/cadp/man/bcg.html`.

States and stacks · A

We assume there is a countably infinite number of distinct identifiers, which are typically denoted by i, i', ... Identifiers can be used to denote *programming variables* or *channels*. Recall that programming variables are typically denoted by x, x', ... (see Chapter 3). Channels are typically denoted by m, m', ... Programming variable identifiers and channel identifiers are associated with values (also called constant expressions, see Chapter 3). Recall that values are typically denoted by c, c', ... The association of an identifier and a value is called a *valuation*.

Definition A.1 · (Valuation) *Let v be a valuation, i be an identifier, and c be a value. A valuation is a mapping from an identifier to a value with syntax $v ::= i \mapsto c$.*

Valuations are typically denoted by v, v', ... The notation $i \mapsto \bot$ denotes that there is a value c such that $i \mapsto c$. This notation allows the value of an identifier to be unspecified.

Valuations occur in *states*. States gather valuations so that programming variables or channels can be assigned a value. Typically, we use s, s', ... to denote states.

Definition A.2 · (State) *The empty state is denoted by λ_s. Further, let v be a valuation. A state s is a list of valuations with syntax*

$$\begin{aligned} s ::= {} & \lambda_s \\ & \mid v : s. \end{aligned}$$

By definition, states have unique identifiers. That is, each identifier occurs at most once in a state. In addition, all possible states are contained in the set State.

Note that the colon-symbol ':' is used as the construction operator for states.

If a state contains a valuation from a certain identifier to a value, then that identifier is in the *domain* of that state.

Definition A.3 · *Let s be a state, i be an identifier, and c be a value. The function dom, which returns the domain of a state, is defined by*

$$\begin{aligned} dom(\lambda_s) &= \emptyset, \\ dom(i \mapsto c : s) &= \{i\} \cup dom(s). \end{aligned}$$

The value corresponding to an identifier in a state can be changed by the *substitution* operator.

Definition A.4 · *Let s be a state, i and i' be identifiers, and c and c' be values. Substitution on states is defined by*

$$\begin{aligned} \lambda_s[c/i] &= \lambda_s, \\ (i \mapsto c : s)[c'/i] &= i \mapsto c' : s, \\ (i \mapsto c : s)[c'/i'] &= i \mapsto c : s[c'/i'] \quad \textit{if } i \neq i'. \end{aligned}$$

Note that an update can never add new valuations (and consequently new identifiers) to states. If the identifier to be updated does not occur in a valuation in a state, then substitution is the identity operation.

If e is an expression and s a state, then the evaluation of e in s is written as $s(e)$. A variable is *defined* in a state if and only if it is in the domain of the state. If it is not in the domain, it is *undefined* in that state. If all variables occurring in e are defined in s, then the result of $s(e)$ will be a value. If variables occurring in e are not defined in s, then the result of $s(e)$ does not have to be a value but can still be an expression containing variables. Evaluation of identifiers in states is defined below. Evaluation of expressions is not defined here.

Definition A.5 · *Let s be a state, i and i' be identifiers, and c be a value. Looking up identifiers in states is defined by*

$$\begin{aligned} \lambda_s(i) &= i, \\ (i \mapsto c : s)(i) &= c, \\ (i \mapsto c : s)(i') &= s(i') \quad \textit{if } i \neq i'. \end{aligned}$$

Two states are equivalent if for every identifier i evaluation of i in those two states has the same result.

Definition A.6 · *Let s and s' be states, and i be an identifier. Equivalence on states is defined by $s = s'$ if $\forall i : s(i) = s'(i)$.*

It can be proven easily that equivalence on states is an equivalence relation; it is reflexive ($s = s$ for all $s \in \mathit{State}$), symmetric ($s = s'$ if and only if $s' = s$

for all $s, s' \in \mathit{State}$), and transitive (if $s = s'$ and $s' = s''$ then $s = s''$, for all $s, s', s'' \in \mathit{State}$).

The substitution operator does not add new valuations to states. It only updates values of existing identifiers. Sometimes however, we do want an identifier to be added to a state if this identifier is not yet contained by that state. The *set* function does just this; it changes values of existing identifiers and adds new identifiers and their corresponding values to states.

Definition A.7 · *Let s and s' be states, i be an identifier, and c be a value. The set function on states is defined by*

$$\begin{aligned} \mathit{set}(s, \lambda_s) &= s, \\ \mathit{set}(s, i \mapsto c : s') &= \mathit{set}(s[c/i], s') && \text{if } i \in \mathit{dom}(s), \\ \mathit{set}(s, i \mapsto c : s') &= \mathit{set}(i \mapsto c : s, s') && \text{if } i \notin \mathit{dom}(s). \end{aligned}$$

States can be stacked in so-called *state stacks*. These state stacks are typically denoted by σ, σ', ... Usually, we refer to state stacks simply by the word stack. In a stack, the same identifier can occur more than once, but only in different states of the stack.

Definition A.8 · (State stack) *The empty state stack is denoted by λ_σ. Let σ be a state stack and s be a state. A state stack has syntax*

$$\begin{aligned} \sigma ::= {} & \lambda_\sigma \\ \mid {} & s :: \sigma. \end{aligned}$$

All possible state stacks are contained in the set Stack.

As can be seen, we use the double colon symbol '::' as the push operator for state stacks.

As for states, the function *dom* is also defined for stacks.

Definition A.9 · *Let σ be a stack, and s be a state. The function dom is defined by*

$$\begin{aligned} \mathit{dom}(\lambda_\sigma) &= \emptyset, \\ \mathit{dom}(s :: \sigma) &= \mathit{dom}(s) \cup \mathit{dom}(\sigma). \end{aligned}$$

The substitution operation is generalized to stacks as defined below.

Definition A.10 · *Let σ be a stack, s be a state, i be an identifier, and c be a value. Substitution on stacks is defined by*

$$\begin{aligned} \lambda_\sigma[c/i] &= \lambda_\sigma, \\ (s :: \sigma)[c/i] &= s[c/i] :: \sigma \quad \textit{if } i \in dom(s), \\ (s :: \sigma)[c/i] &= s :: \sigma[c/i] \quad \textit{if } i \notin dom(s). \end{aligned}$$

In the same way we evaluate expressions in states, we can also evaluate expressions in stacks. For instance, if e is an expression and σ a stack, then $\sigma(e)$ is the evaluation of e in σ. Variables in expressions are looked up starting in the top state of the stack. A variable in lower levels of a stack is made invisible by the same variable in a higher level of the stack.

Definition A.11 · *Let σ be a stack, s be a state, and i be an identifier. Looking up identifiers in stacks is defined by*

$$\begin{aligned} \lambda_\sigma(i) &= i, \\ (s :: \sigma)(i) &= s(i) \quad \textit{if } i \in dom(s), \\ (s :: \sigma)(i) &= \sigma(i) \quad \textit{if } i \notin dom(s). \end{aligned}$$

Definition A.12 · *Let σ and σ' be stacks, and i be an identifier. Equivalence on stacks is defined by*

$$\begin{aligned} \lambda_\sigma &= \lambda_\sigma, \\ s :: \sigma &= s' :: \sigma' \quad \textit{if } s = s' \wedge \sigma = \sigma'. \end{aligned}$$

It can be proven easily that equivalence on stacks is an equivalence relation.

Definition A.13 · *Let σ and σ' be stacks, and i be an identifier. Observational equivalence on stacks is defined by $\sigma \stackrel{\circ}{=} \sigma'$ if $\forall i : \sigma(i) = \sigma'(i)$.*

Next, we present some lemmas regarding states and stacks.

The order of valuations in states is irrelevant.

Lemma A.14 · *Let s be a state, i and i' be identifiers, and c and c' be values. Then $i \mapsto c : i' \mapsto c' : s = i' \mapsto c' : i \mapsto c : s$.*

Proof · (Lemma A.14) Recall that according to Definition A.2 states have unique identifiers. So, we know that $i \neq i'$. According to Definition A.6, proving that $i \mapsto c : i' \mapsto c' : s = i' \mapsto c' : i \mapsto c : s$ means proving that $\forall i'' : (i \mapsto c : i' \mapsto c' : s)(i'') = (i' \mapsto c' : i \mapsto c : s)(i'')$. With respect to i'', we distinguish three cases: $i'' = i$, $i'' = i'$, and $i'' \neq i \wedge i'' \neq i'$.

$i'' = i$: For the left-hand side we obtain

$$\begin{aligned}(i \mapsto c : i' \mapsto c' : s)(i'') &= (i \mapsto c : i' \mapsto c' : s)(i) \\ &= \{\text{Definition A.5}\} \\ &\quad c,\end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned}(i' \mapsto c' : i \mapsto c : s)(i'') &= (i' \mapsto c' : i \mapsto c : s)(i) \\ &= \{\text{Definition A.5}\} \\ &\quad (i \mapsto c : s)(i) \\ &= \{\text{Definition A.5}\} \\ &\quad c.\end{aligned}$$

$i'' = i'$: The proof is similar to the previous case.

$i'' \neq i \wedge i'' \neq i'$: For the left-hand side we obtain

$$\begin{aligned}(i \mapsto c : i' \mapsto c' : s)(i'') &= \{\text{Definition A.5}\} \\ &\quad (i' \mapsto c' : s)(i'') \\ &= \{\text{Definition A.5}\} \\ &\quad s(i''),\end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned}(i' \mapsto c' : i \mapsto c : s)(i'') &= \{\text{Definition A.5}\} \\ &\quad (i \mapsto c : s)(i'') \\ &= \{\text{Definition A.5}\} \\ &\quad s(i'').\end{aligned}$$

□

Lemma A.15 · *Let s and s' be states, i be an identifier, and c be a value. Then*

$$s = i \mapsto c : s' \Rightarrow dom(s') \subset dom(s).$$

Proof · (Lemma A.15) According to Definition A.2, states have unique identifiers. As a consequence, we know that $i \notin dom(s')$. Structural induction on s' gives us the following.

Basis $s' = \lambda_s$. This gives us $s = i \mapsto c : \lambda_s$. From Definition A.3 it follows that $dom(s) = i$ and $dom(s') = \emptyset$. So, $dom(s') \subset dom(s)$.

Inductive step $s' = i' \mapsto c' : s''$, with $s = i \mapsto c : s'' \Rightarrow dom(s'') \subset dom(s)$ the induction hypothesis. Recall that due to Definition A.2 we have that $i \neq i'$. Using the induction hypothesis, we find that $dom(s'') \subset dom(s')$. Combined with the knowledge that $i \notin dom(s')$ we now know that $i \notin dom(s'')$. Furthermore, using Definition A.3, we can derive that $dom(s) = \{i, i'\} \cup dom(s'')$ and $dom(s') = \{i'\} \cup dom(s'')$. Since $i \notin dom(s'')$, we have $dom(s') \subset dom(s)$. □

Substitution on a state does not change its domain.

Lemma A.16 · *Let s be a state, i be an identifier, and c be a value. In that case, $dom(s[c/i]) = dom(s)$.*

Proof · (Lemma A.16) We prove that $dom(s[c/i]) = dom(s)$ by structural induction on s.

Basis $s = \lambda_s$. Consider the following computation:

$$\begin{aligned} dom(s[c/i]) &= dom(\lambda_s[c/i]) \\ &= \{\text{Definition A.4}\} \\ &\quad dom(\lambda_s) \\ &= dom(s). \end{aligned}$$

Inductive step $s = i' \mapsto c' : s'$ with $dom(s'[c/i]) = dom(s')$ the induction hypothesis. We distinguish two cases: $i' = i$ and $i' \neq i$.

$i' = i$: Consider the following computation:

$$\begin{aligned} dom(s[c/i]) &= dom((i \mapsto c' : s')[c/i]) \\ &= \{\text{Definition A.4}\} \\ &\quad dom(i \mapsto c : s') \\ &= \{\text{Definition A.3}\} \\ &\quad \{i\} \cup dom(s') \\ &= dom(s). \end{aligned}$$

$i' \neq i$: Consider the following computation:

$$\begin{aligned} dom(s[c/i]) &= dom((i' \mapsto c' : s')[c/i]) \\ &= \{\text{Definition A.4}\} \\ &\quad dom(i' \mapsto c' : s'[c/i]) \\ &= \{\text{Definition A.3}\} \\ &\quad \{i'\} \cup dom(s'[c/i]) \\ &= \{\text{Induction hypothesis}\} \\ &\quad \{i'\} \cup dom(s') \\ &= dom(s). \end{aligned}$$

□

Substitution of value $s(i)$ for identifier i in state s is an identity operation on states provided that $i \in dom(s)$. If $i \notin dom(s)$ then every arbitrary substitution is an identity operation.

Lemma A.17 · *Let s be a state, i be an identifier, and c be a value. Then*

$$\begin{aligned} s[s(i)/i] &= s \quad \textit{if } i \in dom(s), \\ s[c/i] &= s \quad \textit{if } i \notin dom(s). \end{aligned}$$

Proof · (Lemma A.17) First, we prove that $s[s(i)/i] = s$ if $i \in dom(s)$. Suppose $i \in dom(s)$. According to Lemma A.14, we can say without loss of generality that $s = i \mapsto c : s'$ for some c and s'. Note that $s(i) = c$. Consider the following computation:

$$\begin{aligned} (i \mapsto c : s')[s(i)/i] &= (i \mapsto c : s')[s(i)/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto s(i) : s' \\ &= i \mapsto c : s' \\ &= s. \end{aligned}$$

Next, we prove that $s[c/i] = s$ if $i \notin dom(s)$. Structural induction on s gives us the following.

Basis $s = \lambda_s$. Consider the following computation:

$$\begin{aligned} s[c/i] &= \lambda_s[c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s \\ &= s. \end{aligned}$$

Inductive step $s = i' \mapsto c' : s'$, with $i' \neq i$ and $s'[c/i] = s'$ if $i \notin dom(s')$ the induction hypothesis. Using Lemma A.15 we obtain $i \notin dom(s')$. Now, consider the following computation:

$$\begin{aligned} s[c/i] &= (i' \mapsto c' : s')[c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i' \mapsto c' : s'[c/i] \\ &= \{\text{Induction hypothesis}\} \\ &\quad i' \mapsto c' : s' \\ &= s. \end{aligned}$$

□

For two or more substitutions on states we have that if two substitutions address the same identifier, only the last substitution is relevant. If they address different identifiers then their order is irrelevant.

Lemma A.18 · *Let s be a state, i and i' be identifiers, and c and c' be values. Then*

$$\begin{aligned} s[c/i][c'/i'] &= s[c'/i'] \quad \textit{if } i = i', \\ s[c/i][c'/i'] &= s[c'/i'][c/i] \quad \textit{if } i \neq i'. \end{aligned}$$

Proof · (Lemma A.18) We first prove that $s[c/i][c'/i'] = s[c'/i']$ if $i = i'$. Suppose $i = i'$. Structural induction on s gives us the following.

Basis $s = \lambda_s$. Consider the following computation:

$$\begin{aligned} s[c/i][c'/i'] &= \lambda_s[c/i][c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s[c'/i'] \\ &= s[c'/i']. \end{aligned}$$

Inductive step $s = i'' \mapsto c'' : s'$, with $s'[c/i][c'/i'] = s'[c'/i']$ if $i = i'$ the induction hypothesis. With respect to i'' we distinguish two cases: $i'' = i$ and $i'' \neq i$.

$i'' = i$: For the left-hand side we obtain

$$\begin{aligned} s[c/i][c'/i] &= (i \mapsto c'' : s')[c/i][c'/i] \\ &= \{\text{Definition A.4}\} \\ &\quad (i \mapsto c : s')[c'/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto c' : s', \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s[c'/i] &= (i \mapsto c'' : s')[c'/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto c' : s'. \end{aligned}$$

$i'' \neq i$: For the left-hand side we obtain

$$\begin{aligned} s[c/i][c'/i'] &= (i'' \mapsto c'' : s')[c/i][c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad (i'' \mapsto c'' : s'[c/i])[c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad i'' \mapsto c'' : s'[c/i][c'/i'] \\ &= \{\text{Induction hypothesis}\} \\ &\quad i'' \mapsto c'' : s'[c'/i'], \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s[c'/i'] &= (i'' \mapsto c'' : s')[c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad i'' \mapsto c'' : s'[c'/i']. \end{aligned}$$

Next, we prove that $s[c/i][c'/i'] = s[c'/i'][c/i]$ if $i \neq i'$. Suppose $i \neq i'$. Structural induction on s gives us the following.

Basis $s = \lambda_s$. For the left-hand side we obtain

$$\begin{aligned} s[c/i][c'/i'] &= \lambda_s[c/i][c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s[c'/i] \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s, \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s[c'/i'][c/i] &= \lambda_s[c'/i'][c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s[c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s. \end{aligned}$$

Inductive step $s = i'' \mapsto c'' : s'$, with $s'[c/i][c'/i'] = s'[c'/i'][c/i]$ if $i \neq i'$ the induction hypothesis. With respect to i'' we distinguish three cases: $i'' = i$, $i'' = i'$, and $i'' \neq i \wedge i'' \neq i'$.

$i'' = i$: For the left-hand side we obtain

$$\begin{aligned} s[c/i][c'/i'] &= (i \mapsto c'' : s')[c/i][c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad (i \mapsto c : s')[c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto c' : s'[c'/i'], \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s[c'/i'][c/i] &= (i \mapsto c'' : s'[c'/i'])[c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto c : s'[c'/i']. \end{aligned}$$

$i'' = i'$: The proof is similar to the previous case.

$i'' \neq i \wedge i'' \neq i'$: For the left-hand side we obtain

$$\begin{aligned} s[c/i][c'/i'] &= (i'' \mapsto c'' : s')[c/i][c'/i'] \\ &= \{\text{Definition A.4}\} \\ &\quad (i'' \mapsto c'' : s'[c/i])[c'/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i'' \mapsto c'' : s'[c/i][c'/i'], \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s[c'/i'][c/i] &= (i'' \mapsto c'' : s')[c'/i'][c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad (i'' \mapsto c'' : s'[c'/i'])[c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i'' \mapsto c'' : s'[c'/i'][c/i] \\ &= \{\text{Induction hypothesis}\} \\ &\quad i'' \mapsto c'' : s'[c/i][c'/i']. \end{aligned}$$

□

Lemma A.19 · *Let s be a state, i and i' be identifiers, and c be a value. Then*

$$\begin{aligned} s[c/i](i) &= c && \text{if } i \in dom(s), \\ s[c/i](i) &= i && \text{if } i \notin dom(s), \\ s[c/i](i') &= s(i') && \text{if } i \neq i'. \end{aligned}$$

Proof · (Lemma A.19) We first prove that $s[c/i](i) = c$ if $i \in dom(s)$. Suppose $i \in dom(s)$. According to Lemma A.14, we can say without loss of generality that $s = i \mapsto c' : s'$ for some c' and s'. Now, consider the following computation:

$$\begin{aligned} s[c/i](i) &= (i \mapsto c' : s')[c/i](i) \\ &= \{\text{Definition A.4}\} \\ &\quad (i \mapsto c : s')(i) \\ &= \{\text{Definition A.5}\} \\ &= c. \end{aligned}$$

Next, we prove that $s[c/i](i) = i$ if $i \notin dom(s)$. Suppose $i \notin dom(s)$. Structural induction on s gives us the following.

Basis $s = \lambda_s$. Consider the following computation:

$$\begin{aligned} s[c/i](i) &= \lambda_s[c/i](i) \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s(i) \\ &= \{\text{Definition A.5}\} \\ &\quad i. \end{aligned}$$

Inductive step $s = i' \mapsto c' : s'$, with $s'[c/i](i) = i$ if $i \notin dom(s')$ the induction hypothesis. Using Lemma A.15 we obtain $dom(s') \subset dom(s)$. Since $i \notin dom(s)$, this implies that also $i \notin dom(s')$. Further, it also implies that $i' \neq i$.

Now, consider the following computation:

$$\begin{aligned} s[c/i](i) &= (i' \mapsto c' : s')[c/i](i) \\ &= \{\text{Definition A.4}\} \\ &\quad (i' \mapsto c' : s'[c/i])(i) \\ &= \{\text{Definition A.5}\} \\ &\quad s'[c/i](i) \\ &= \{\text{Induction hypothesis}\} \\ &\quad i. \end{aligned}$$

Finally, we prove that $s[c/i](i') = s(i')$ if $i' \neq i$. Suppose that $i' \neq i$. Structural induction on s gives us the following.

Basis $s = \lambda_s$. For the left-hand side we obtain

$$\begin{aligned} s[c/i](i') &= \lambda_s[c/i](i') \\ &= \{\text{Definition A.4}\} \\ &\quad \lambda_s(i'), \end{aligned}$$

and for the right-hand side we obtain

$$s(i') = \lambda_s(i').$$

Inductive step $s = i'' \mapsto c' : s'$, with $s'[c/i](i') = s'(i')$ if $i' \neq i$ the induction hypothesis. With respect to i'', we distinguish three cases: $i'' = i$, $i'' = i'$, and $i'' \neq i \wedge i'' \neq i'$.

$i'' = i$: For the left-hand side we obtain

$$\begin{aligned} s[c/i](i') &= (i \mapsto c' : s')[c/i](i') \\ &= \{\text{Definition A.4}\} \\ &\quad (i \mapsto c : s'[c/i])(i') \\ &= \{\text{Definition A.5}\} \\ &\quad s'[c/i](i') \\ &= \{\text{Induction hypothesis}\} \\ &\quad s'(i'), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s(i') &= (i \mapsto c' : s')(i') \\ &= \{\text{Definition A.5}\} \\ &\quad s'(i'). \end{aligned}$$

$i'' = i'$: For the left-hand side we obtain

$$\begin{aligned} s[c/i](i') &= (i' \mapsto c' : s')[c/i](i') \\ &= \{\text{Definition A.4}\} \\ &\quad (i' \mapsto c' : s'[c/i])(i') \\ &= \{\text{Definition A.5}\} \\ &\quad c', \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s(i') &= (i' \mapsto c' : s')(i') \\ &= \{\text{Definition A.5}\} \\ &\quad c'. \end{aligned}$$

$i'' \neq i \wedge i'' \neq i'$: For the left-hand side we obtain

$$\begin{aligned} s[c/i](i') &= (i'' \mapsto c' : s')[c/i](i') \\ &= \{\text{Definition A.4}\} \\ &\quad (i'' \mapsto c : s'[c/i])(i') \\ &= \{\text{Definition A.5}\} \\ &\quad s'[c/i](i') \\ &= \{\text{Induction hypothesis}\} \\ &\quad s'(i'), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} s(i') &= (i'' \mapsto c' : s')(i') \\ &= \{\text{Definition A.5}\} \\ &\quad s'(i'). \end{aligned}$$

□

Lemma A.20 · *Let s and s' be states, c be a value, and i be an identifier. Then $set(i \mapsto c : s, s') = i \mapsto c : set(s, s')$ if $i \notin dom(s')$.*

Proof · (Lemma A.20) Structural induction on s' gives us the following.

Basis $s' = \lambda_s$. For the left-hand side we obtain

$$\begin{aligned} set(i \mapsto c : s, s') &= set(i \mapsto c : s, \lambda_s) \\ &= \{\text{Definition A.7}\} \\ &\quad i \mapsto c : s, \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} i \mapsto c : set(s, s') &= i \mapsto c : set(s, \lambda_s) \\ &= \{\text{Definition A.7}\} \\ &\quad i \mapsto c : s. \end{aligned}$$

Inductive step $s' = i' \mapsto c' : s''$, for some i', c', and s'' such that $i' \neq i$ and $i \notin dom(s'')$. Furthermore, we have the following induction hypothesis: $set(i \mapsto$

$c : s, s'') = i \mapsto c : set(s, s'')$ if $i \notin dom(s'')$. Now, for the left-hand side we obtain

$$\begin{aligned} set(i \mapsto c : s, s') &= set(i \mapsto c : s, i' \mapsto c' : s'') \\ &= \{\text{Definition A.7}\} \\ &\quad set(i' \mapsto c' : i \mapsto c : s, s'') \\ &= \{\text{Definition A.6}\} \\ &\quad set(i \mapsto c : i' \mapsto c' : s, s'') \\ &= \{\text{Induction hypothesis}\} \\ &\quad i \mapsto c : set(i' \mapsto c' : s, s''), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} i \mapsto c : set(s, s') &= i \mapsto c : set(s, i' \mapsto c' : s'') \\ &= \{\text{Definition A.7}\} \\ &\quad i \mapsto c : set(i' \mapsto c' : s, s''). \end{aligned}$$

□

Lemma A.21 · *Let s and s' be states. Then $dom(set(s, s')) = dom(s) \cup dom(s')$.*

Proof · (Lemma A.21) Structural induction on s' gives us the following.

Basis $s' = \lambda_s$. For the left-hand side we obtain

$$\begin{aligned} dom(set(s, s')) &= dom(set(s, \lambda_s)) \\ &= \{\text{Definition A.3}\} \\ &\quad dom(s), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} dom(s) \cup dom(s') &= dom(s) \cup dom(\lambda_s) \\ &= \{\text{Definition A.3}\} \\ &\quad dom(s) \cup \emptyset \\ &= dom(s). \end{aligned}$$

Inductive step $s' = i \mapsto c : s''$, with $dom(set(s, s'')) = dom(s) \cup dom(s'')$ the induction hypothesis. We distinguish two cases: $i \in dom(s)$ and $i \notin dom(s)$.

$i \in dom(s)$: For the left-hand side we obtain

$$\begin{aligned} dom(set(s, s')) &= dom(set(s, i \mapsto c : s'')) \\ &= \{\text{Definition A.7}\} \\ &\quad dom(set(s[c/i], s'')) \\ &= \{\text{Induction hypothesis}\} \\ &\quad dom(s[c/i]) \cup dom(s'') \\ &= \{\text{Lemma A.16}\} \\ &\quad dom(s) \cup dom(s'') \\ &= \{\text{Lemma A.14 and } \exists s''' : i \mapsto c : s'''\} \\ &\quad dom(i \mapsto c; s''') \cup dom(s'') \\ &= \{\text{Definition A.3}\} \\ &\quad \{i\} \cup dom(s''') \cup dom(s''), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} dom(s) \cup dom(s') &= dom(s) \cup dom(i \mapsto c : s'') \\ &= \{\text{Lemma A.14 and } \exists s''' : i \mapsto c : s'''\} \\ &\quad dom(i \mapsto c; s''') \cup dom(i \mapsto c : s'') \\ &= \{\text{Definition A.3}\} \\ &\quad \{i\} \cup dom(s''') \cup \{i\} \cup dom(s'') \\ &= \{i\} \cup dom(s''') \cup dom(s''). \end{aligned}$$

$i \notin dom(s)$: For the left-hand side we obtain

$$\begin{aligned} dom(set(s, s')) &= dom(set(s, i \mapsto c : s'')) \\ &= \{\text{Definition A.7}\} \\ &\quad dom(set(i \mapsto c : s, s'')) \\ &= \{\text{Induction hypothesis}\} \\ &\quad dom(i \mapsto c : s) \cup dom(s'') \\ &= \{\text{Definition A.3}\} \\ &\quad \{i\} \cup dom(s) \cup dom(s''), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} dom(s) \cup dom(s') &= dom(s) \cup dom(i \mapsto c : s'') \\ &= \{\text{Definition A.3}\} \\ &\quad dom(s) \cup \{i\} \cup dom(s'') \\ &= \{i\} \cup dom(s) \cup dom(s''). \end{aligned}$$

□

Lemma A.22 · *Let s and s' be states, c be a value, and i be an identifier. Then*

$$set(s, s')[c/i] = set(s, s'[c/i]) \text{ if } i \in dom(s'),$$
$$set(s, s')[c/i] = set(s[c/i], s') \text{ if } i \notin dom(s').$$

Proof · (Lemma A.22) First, we prove that $set(s, s')[c/i] = set(s, s'[c/i])$ if $i \in dom(s')$. Suppose $i \in dom(s')$. According to Lemma A.14, we can say without loss of generality that $s' = i \mapsto c' : s''$ for some c' and s'' such that, according to Lemma A.15, $i \notin dom(s'')$. Now, consider the following computation:

$$\begin{aligned} set(s, s')[c/i] &= set(s, s'[c/i]) \\ set(s, i \mapsto c' : s'')[c/i] &= set(s, (i \mapsto c' : s'')[c/i]) \\ &= \{\text{Definition A.4}\} \\ &\quad set(s, i \mapsto c : s''). \end{aligned}$$

We distinguish two cases: $i \in dom(s)$ and $i \notin dom(s)$.

$i \in dom(s)$: As a consequence, according to Lemma A.14, we can say without loss of generality that $s = i \mapsto c'' : s'''$ for some c'' and s''' such that, according to Lemma A.15, $i \notin dom(s''')$. For the left-hand side we obtain

$$\begin{aligned} set(s, i \mapsto c' : s'')[c/i] &= set(i \mapsto c'' : s''', i \mapsto c' : s'')[c/i] \\ &= \{\text{Definition A.7}\} \\ &\quad set(i \mapsto c' : s''', s'')[c/i] \\ &= \{\text{Lemma A.20}\} \\ &\quad (i \mapsto c' : set(s''', s''))[c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto c : set(s''', s''), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} set(s, i \mapsto c : s'') &= set(i \mapsto c'' : s''', i \mapsto c : s'') \\ &= \{\text{Definition A.7}\} \\ &\quad set(i \mapsto c : s''', s'') \\ &= \{\text{Lemma A.20}\} \\ &\quad i \mapsto c : set(s''', s''). \end{aligned}$$

$i \notin dom(s)$: For the left-hand side we obtain

$$\begin{aligned} set(s, i \mapsto c' : s'')[c/i] &= set(i \mapsto c' : s, s'')[c/i] \\ &= \{\text{Lemma A.20}\} \\ &\quad (i \mapsto c' : set(s, s''))[c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto c : set(s, s''), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} set(s, i \mapsto c : s'') &= set(i \mapsto c : s, s'') \\ &= \{\text{Lemma A.20}\} \\ &\quad i \mapsto c : set(s, s''). \end{aligned}$$

Next, we prove that $set(s, s')[c/i] = set(s[c/i], s')$ if $i \notin dom(s')$. Suppose $i \notin dom(s')$. We distinguish two cases: $i \in dom(s)$ and $i \notin dom(s)$.

$i \in dom(s)$: As a consequence, according to Lemma A.14 we can say without loss of generality that $s = i \mapsto c' : s''$ for some c' and s'' such that, according to Lemma A.15, $i \notin dom(s'')$. For the left-hand side we obtain

$$\begin{aligned} set(s, s')[c/i] &= set(i \mapsto c' : s'', s')[c/i] \\ &= \{\text{Lemma A.20}\} \\ &\quad (i \mapsto c' : set(s'', s'))[c/i] \\ &= \{\text{Definition A.4}\} \\ &\quad i \mapsto c : set(s'', s'), \end{aligned}$$

and for the right-hand side we obtain

$$\begin{aligned} set(s[c/i], s') &= set((i \mapsto c' : s'')[c/i], s') \\ &= \{\text{Definition A.4}\} \\ &\quad set(i \mapsto c : s'', s') \\ &= \{\text{Lemma A.20}\} \\ &\quad i \mapsto c : set(s'', s'). \end{aligned}$$

$i \notin dom(s)$: Here we have $i \notin dom(s)$ and $i \notin dom(s')$. According to Lemma A.21, this implies that also $i \notin dom(set(s, s'))$. Using Lemma A.17, we now immediately obtain $set(s, s')[c/i] = set(s, s')$. □

Lemma A.23 · *Let s and s' be states, and i be an identifier. Then*

$$\begin{aligned} set(s, s')(i) &= s'(i) \quad \textit{if } i \in dom(s'), \\ set(s, s')(i) &= s(i) \quad \textit{if } i \notin dom(s'). \end{aligned}$$

Proof · (Lemma A.23) We first prove that $set(s, s')(i) = s'(i)$ if $i \in dom(s')$. Suppose $i \in dom(s')$. According to Lemma A.14, we can say without loss of generality that $s' = i \mapsto c : s''$ and from Definition A.2, it follows that $i \notin dom(s'')$. For the left-hand side, we obtain $set(s, i \mapsto c : s'')(i)$, and for the right-hand side we obtain $(i \mapsto c : s'')(i)$, which equals c according to Definition A.5. So, we need to prove that $set(s, i \mapsto c : s'')(i) = c$. Structural induction on s'' gives us the following.

Basis $s'' = \lambda_s$. We distinguish two cases: $i \in dom(s)$ and $i \notin dom(s)$.

$i \in dom(s)$: Consider the following computation:

$$\begin{aligned} set(s, i \mapsto c : \lambda_s)(i) &= \{\text{Definition A.7}\} \\ &\quad set(s[c/i], \lambda_s)(i) \\ &= \{\text{Definition A.7}\} \\ &\quad s[c/i](i) \\ &= \{\text{Lemma A.19}\} \\ &\quad c. \end{aligned}$$

$i \notin dom(s)$: Consider the following computation:

$$\begin{aligned} set(s, i \mapsto c : \lambda_s)(i) &= \{\text{Definition A.7}\} \\ &\quad set(i \mapsto c : s, \lambda_s)(i) \\ &= \{\text{Definition A.7}\} \\ &\quad (i \mapsto c : s)(i) \\ &= \{\text{Definition A.5}\} \\ &\quad c. \end{aligned}$$

Inductive step $s'' = i' \mapsto c' : s'''$, with $set(s, i \mapsto c : s''')(i) = c$ the induction hypothesis. Now, consider the following computation:

$$\begin{aligned} set(s, i \mapsto c : s'')(i) &= set(s, i \mapsto c : i' \mapsto c' : s''')(i) \\ &= \{\text{Lemma A.14}\} \\ &\quad set(s, i' \mapsto c' : i \mapsto c : s''')(i). \end{aligned}$$

We distinguish two cases: $i' \in dom(s)$ and $i' \notin dom(s)$.

$i' \in dom(s)$: This gives us

$$\begin{aligned} set(s, i' \mapsto c' : i \mapsto c : s''')(i) &= set(s[c'/i'], i \mapsto c : s''')(i) \\ &= \{\text{Induction hypothesis}\} \\ &\quad c. \end{aligned}$$

$i' \notin dom(s)$: This gives us

$$\begin{aligned} set(s, i' \mapsto c' : i \mapsto c : s''')(i) &= set(i' \mapsto c' : s, i \mapsto c : s''')(i) \\ &= \{\text{Induction hypothesis}\} \\ &\quad c. \end{aligned}$$

Next, we prove that $set(s, s')(i) = s(i)$ if $i \notin dom(s')$. Suppose $i \notin dom(s')$. Structural induction on s' gives us the following.

Basis $s' = \lambda_s$. Consider the following computation:

$$\begin{aligned} set(s, s')(i) &= set(s, \lambda_s)(i) \\ &= \{\text{Definition A.7}\} \\ &\quad s(i). \end{aligned}$$

Inductive step $s' = i' \mapsto c : s''$, with $set(s, s'')(i) = s(i)$ if $i \notin dom(s'')$ the induction hypothesis. We have $i \notin dom(s'')$ since $i \notin dom(s')$ and Lemma A.15 gives $dom(s'') \subset dom(s')$. Consequently, also $i \notin dom(s'')$. Furthermore, since $i \notin dom(s')$, we know that $i' \neq i$.

Now, consider the equation $set(s, s')(i) = set(s, i' \mapsto c : s'')(i)$. We distinguish two cases: $i' \in dom(s)$ and $i' \notin dom(s)$.

$i' \in dom(s)$: This gives us

$$\begin{aligned} set(s, i' \mapsto c : s'')(i) &= \{\text{Definition A.7}\} \\ &\quad set(s[c/i'], s'')(i) \\ &= \{\text{Induction hypothesis}\} \\ &\quad s[c/i'](i) \\ &= \{\text{Lemma A.19}\} \\ &\quad s(i). \end{aligned}$$

$i' \notin dom(s)$: This gives us

$$\begin{aligned} set(s, i' \mapsto c : s'')(i) &= \{\text{Definition A.7}\} \\ & \quad set(i' \mapsto c : s, s'')(i) \\ &= \{\text{Induction hypothesis}\} \\ & \quad (i' \mapsto c : s)(i) \\ &= \{\text{Definition A.5}\} \\ &= s(i). \end{aligned}$$

□

Lemma A.24 · *Let s be a state and σ and σ' be stacks. Then*

$$\sigma = s :: \sigma' \Rightarrow dom(s) \subseteq dom(\sigma) \wedge dom(\sigma') \subseteq dom(\sigma).$$

Proof · (Lemma A.24) Suppose $\sigma = s :: \sigma'$. We distinguish two cases: $\sigma' = \lambda_\sigma$ and $\sigma' = s' :: \sigma''$.

$\sigma' = \lambda_\sigma$: Using Definition A.9, we can perform the following computations:

$$\begin{aligned} dom(s) &\subseteq dom(\sigma) \\ &\subseteq dom(s :: \lambda_\sigma) \\ &\subseteq dom(s) \cup dom(\lambda_\sigma) \\ &\subseteq dom(s) \cup \emptyset \\ &\subseteq dom(s), \end{aligned}$$

$$\begin{aligned} dom(\sigma') &\subseteq dom(\sigma) \\ dom(\lambda_\sigma) &\subseteq dom(s :: \lambda_\sigma) \\ \emptyset &\subseteq dom(s) \cup dom(\lambda_\sigma) \\ &\subseteq dom(s) \cup \emptyset \\ &\subseteq dom(s). \end{aligned}$$

$\sigma' = s' :: \sigma''$: Using Definition A.9, we can perform the following computations:

$$\begin{aligned} dom(s) &\subseteq dom(\sigma) \\ &\subseteq dom(s :: s' :: \sigma'') \\ &\subseteq dom(s) \cup dom(s') \cup dom(\sigma''), \end{aligned}$$

$$\begin{aligned} dom(\sigma') &\subseteq dom(\sigma) \\ dom(s' :: \sigma'') &\subseteq dom(s :: s' :: \sigma'') \\ dom(s') \cup dom(\sigma'') &\subseteq dom(s) \cup dom(s') \cup dom(\sigma''). \end{aligned}$$

□

Substitution on a stack does not change its domain.

Lemma A.25 · *Let σ be a stack, i be an identifier, c be a value, I be a set of identifiers, and c_i be a value for all $i \in I$. Then $dom(\sigma[c/i]) = dom(\sigma)$.*

Proof · (Lemma A.25) We prove that $dom(\sigma[c/i]) = dom(\sigma)$ by structural induction on σ.

Basis $\sigma = \lambda_\sigma$. Consider the following computation:

$$\begin{aligned} dom(\sigma[c/i]) &= dom(\lambda_\sigma[c/i]) \\ &= \{\text{Definition A.10}\} \\ &\quad dom(\lambda_\sigma) \\ &= dom(\sigma). \end{aligned}$$

Inductive step $\sigma = s :: \sigma'$ with $dom(\sigma'[c/i]) = dom(\sigma')$ the induction hypothesis. We distinguish two cases: $i \in dom(s)$, and $i \notin dom(s)$.

$i \in dom(s)$: Consider the following computation:

$$\begin{aligned} dom(\sigma[c/i]) &= dom((s :: \sigma')[c/i]) \\ &= \{\text{Definition A.10}\} \\ &\quad dom(s[c/i] :: \sigma') \\ &= \{\text{Definition A.9}\} \\ &\quad dom(s[c/i]) \cup dom(\sigma') \\ &= \{\text{Lemma A.16}\} \\ &\quad dom(s) \cup dom(\sigma') \\ &= dom(\sigma). \end{aligned}$$

$i \notin dom(s)$: Consider the following computation:

$$\begin{aligned} dom(\sigma[c/i]) &= dom((s :: \sigma')[c/i]) \\ &= \{\text{definition A.10}\} \\ &\quad dom(s :: \sigma'[c/i]) \\ &= \{\text{Definition A.9}\} \\ &\quad dom(s) \cup dom(\sigma'[c/i]) \\ &= \{\text{Induction hypothesis}\} \\ &\quad dom(s) \cup dom(\sigma') \\ &= dom(\sigma). \end{aligned}$$

□

Substitution of value $\sigma(i)$ for identifier i in stack σ is an identity operation on states provided that $i \in dom(\sigma)$. If $i \notin dom(\sigma)$ then every arbitrary substitution is an identity operation.

Lemma A.26 · *Let σ be a stack, i be an identifier, and c be a value. Then*

$$\begin{aligned} \sigma[\sigma(i)/i] &= \sigma \quad \textit{if } i \in dom(\sigma), \\ \sigma[c/i] &= \sigma \quad \textit{if } i \notin dom(\sigma). \end{aligned}$$

Proof · (Lemma A.26)

First, we prove that $\sigma[\sigma(i)/i] = \sigma$ if $i \in dom(\sigma)$. Suppose $i \in dom(\sigma)$. Then $\sigma = s :: \sigma'$ for some s and σ' such that $i \in dom(s)$ or $i \in dom(\sigma')$. Structural induction on σ' gives us the following.

Basis $\sigma' = \lambda_\sigma$. This gives us $i \in dom(s)$. Now, consider the computation:

$$\begin{aligned} \sigma[\sigma(i)/i] &= (s :: \lambda_\sigma)[(s :: \lambda_\sigma)(i)/i] \\ &= \{\text{Definition A.11}\} \\ &\quad (s :: \lambda_\sigma)[s(i)/i] \\ &= \{\text{Definition A.10}\} \\ &\quad s[s(i)/i] :: \lambda_\sigma \\ &= \{\text{Lemma A.17}\} \\ &\quad s :: \lambda_\sigma \\ &= \sigma. \end{aligned}$$

Inductive Step $\sigma' = s' :: \sigma''$, with $\sigma''[\sigma''(i)/i] = \sigma''$ if $i \in dom(\sigma'')$ the induction hypothesis. With respect to i, we distinguish three cases: $i \in dom(s)$, $i \notin dom(s) \wedge i \in dom(s')$, and $i \notin dom(s) \wedge i \notin dom(s')$.

$i \in dom(s)$: Consider the following computation:

$$\begin{aligned} \sigma[\sigma(i)/i] &= (s :: s' :: \sigma'')[(s :: s' :: \sigma'')(i)/i] \\ &= \{\text{Definition A.11}\} \\ &\quad (s :: s' :: \sigma'')[s(i)/i] \\ &= \{\text{Definition A.10}\} \\ &\quad s[s(i)/i] :: s' :: \sigma'' \\ &= \{\text{Lemma A.17}\} \\ &\quad s :: s' :: \sigma'' \\ &= \sigma. \end{aligned}$$

$i \notin dom(s) \wedge i \in dom(s')$: Consider the following computation:

$$\begin{aligned}
\sigma[\sigma(i)/i] &= (s :: s' :: \sigma'')[(s :: s' :: \sigma'')(i)/i] \\
&= \{\text{definition A.11}\} \\
&\quad (s :: s' :: \sigma'')[s'(i)/i] \\
&= \{\text{definition A.10}\} \\
&\quad s :: s'[s'(i)/i] :: \sigma'' \\
&= \{\text{lemma A.17}\} \\
&\quad s :: s' :: \sigma'' \\
&= \sigma.
\end{aligned}$$

$i \notin dom(s) \wedge i \notin dom(s')$: This gives us, $i \in dom(\sigma'')$. Now, consider the following computation:

$$\begin{aligned}
\sigma[\sigma(i)/i] &= (s :: s' :: \sigma'')[(s :: s' :: \sigma'')(i)/i] \\
&= \{\text{definition A.11}\} \\
&\quad (s :: s' :: \sigma'')[\sigma''(i)/i] \\
&= \{\text{definition A.10}\} \\
&\quad s :: s' :: \sigma''[\sigma''(i)/i] \\
&= \{\text{Induction hypothesis}\} \\
&\quad s :: s' :: \sigma'' \\
&= \sigma.
\end{aligned}$$

Next we proof that $\sigma[c/i] = \sigma$ if $i \notin dom(\sigma)$. Structural induction on σ gives us the following.

Basis $\sigma = \lambda_\sigma$. Now, according to Definition A.10, we have $\sigma[c/i] = \lambda_\sigma[c/i] = \lambda_\sigma = \sigma$.

Inductive step $\sigma = s :: \sigma'$ with $\sigma'[c/i] = \sigma'$ if $i \notin dom(\sigma')$ the induction hypothesis. Consider the following computation:

$$\begin{aligned}
\sigma[c/i] &= (s :: \sigma')[c/i] \\
&= \{\text{Definition A.10}\} \\
&\quad s[c/i] :: \sigma'[c/i] \\
&= \{\text{Lemma A.17 and induction hypothesis}\} \\
&\quad s :: \sigma'.
\end{aligned}$$

□

Experiments · B

In this appendix, we consider a small example of a production system to illustrate how to use the experiment environment. The system consists of a generator, a buffer, a machine, and an exit. The χ_σ specification reads (line numbers are listed for referencing):

```
1   % File: prodsys.chi
2   %
3   G(output: chan) = (skip; output!1 | skip; delay 2)*; deadlock
4
5   E(input: chan) = |[ x: int | (input?x)*; deadlock]|
6
7   B(input, output: chan, cap: int) =
8   |[ x: int, xs: list[int] = mtlist
9    | ( len(xs) < cap :-> input?x; xs := xs ++ (x:mtlist)
10     | len(xs) > 0 :-> output!hd(xs); xs := tl(xs)
11     )*; deadlock
12  ]|
13
14  M(input, output : chan, pt: real) =
15  |[ x : int | (input?x; delay pt; output!x)*; deadlock ]|
16
17  GBME(cap: int, pt: real) =
18  mp enc sa(*,*): ra(*,*): mtset
19         |[ ~gb, ~bm, ~me
20          | G(~gb) || B(~gb, ~bm, cap) || M(~bm, ~me, pt) || E(~me)
21         ]|
```

Lines starting with the '%' sign, lines 1–3 of the example, are comments. Generator `G` is defined on line 4. It makes a nondeterministic choice between sending a new product (represented by the integer 1) over its output channel, or delaying 2 time units. This is repeated infinitely many times.

Line 6 defines the exit process `E`. It repeatedly tries to receive a product over its input channel.

Buffer `B` is defined on lines 8–13. It stores received products in the list `xs`. If the length of this list is less than `cap`, representing the capacity of the buffer, the buffer is able to receive new products over its input channel. Similarly, if the list `xs` is not empty, the buffer is able to send a product over its output channel.

Machine `M` is defined on lines 15–16. It repeatedly executes a receive-delay-send loop. This loop repeatedly receives a product over its input channel, processes the product for `pt` time units (`pt` represents the process time of one product), and sends the product over its output channel.

The parallel composition of the previous processes is defined in process `GBME` on lines 18–22. Furthermore, this process also enforces maximal progress, using the `mp` operator, and it prohibits single send and receive actions, using the `enc` operator. Notice that instead of sets of actions, *action patterns* are used in the encapsulation operator. For example, `aa(i,*)` is an action pattern denoting the set $\{aa(x, e) \mid x \equiv \texttt{i} \wedge e \in \mathit{Expr}\}$.

The χ_σ engine is started in interactive mode by the command `iachi`. After the χ_σ engine has started, commands to read and parse χ_σ specifications, to instantiate process definitions, to simulate processes, to generate process graphs, etc., can be given. The following session illustrates the interactive mode of the χ_σ engine. Each line starting with the symbol '`>>>`' contains a command for the χ_σ engine. The other lines contain output of the χ_σ engine. End of line comments start with the hash sign '`#`'.

```
1   >>> m = Model("prodsys.chi")          # read and parse model
2   >>> s0 = m.instantiate("GBME(4, 2)")  # instantiate process
3   >>> sim = Simulator(s0)               # make simulator for process
4   >>> sim.run(100)                      # run 100 steps
5   >>> sim.reset()                       # reset simulator
6   >>> sim.stepFunction = sim.tracedStep # choose different step function
7   >>> sim.run(1)                        # run one step, produces output:
8   Option 0: tau
9   Option 1: tau
10   Step 0: tau
11
12   ------------------------------------------------------------------------
13   (mp ((enc sa(*, *) : ra(*, *) : mtset (|[ ~gb : ~bm : ~me : mtstate |
14   ((((((empty ; ~gb ! 1) ; (((skip ; ~gb ! 1) | (skip ; delay 2)) *)) ;
15   deadlock) || |[ x: int : xs: list[int] = mtlist : mtstate | (((((len
16   (xs) < 4) :-> (~gb ? x ; xs := (xs ++ (x : mtlist)))) | ((len(xs) > 0)
17    :-> (~bm ! hd(xs) ; xs := tl(xs)))) *) ; deadlock) ]|) || |[ x: int :
18    mtstate | ((((~bm ? x ; delay 2) ; ~me ! x) *) ; deadlock) ]|) || |[
```

```
19  x: int : mtstate | ((~me ? x *) ; deadlock) ]|) ]|))))
20  ------------------------------------------------------------------------
21
22  >>> sim.run(1)                          # run another step:
23  Option 0: ca(~gb, x, 1)
24  Step 1: ca(~gb, x, 1)
25
26  ------------------------------------------------------------------------
27  (mp ((enc sa(*, *) : ra(*, *) : mtset (|[ ~gb = 1 : ~bm : ~me :
28  mtstate | (((((empty ; (((skip ; ~gb ! 1) | (skip ; delay 2)) *)) ;
29  deadlock) || |[ x: int = 1 : xs: list[int] = mtlist : mtstate | (((
30  empty ; xs := (xs ++ (x : mtlist))) ; ((((len(xs) < 4) :-> (~gb ? x ;
31  xs := (xs ++ (x : mtlist)))) | ((len(xs) > 0) :-> (~bm ! hd(xs) ; xs
32  := tl(xs)))) *)) ; deadlock) ]|) || |[ x: int : mtstate | ((((~bm ? x
33  ; delay 2) ; ~me ! x) *) ; deadlock) ]|) || |[ x: int : mtstate | ((
34  ~me ? x *) ; deadlock) ]|) ]|))))
35  ------------------------------------------------------------------------
```

In line 1, the file `prodsys.chi` is read and parsed and the result, a χ_σ model, is stored in the programming variable `m`. Python programming variables are created on the fly, they do not have to be declared. In line 2, process definition `GBME` is instantiated with parameters 4 and 2, representing the buffer capacity and the process time, respectively. The instantiated process is stored in programming variable `s0`.

Line 3 creates a simulator for this process and stores it in programming variable `sim`. A simulator has different *step functions*: functions that perform one step (transition) of the process of the simulator. The default step function is called `quietStep`. This step function does not generate output, except for possible warnings and error messages. Line 4 tells `sim` to perform 100 consecutive steps. Since `sim` uses the default step function, no output is generated. After these steps, the process of the simulator is different from the originally instantiated process and it is possible to continue simulation from that point.

Line 5 resets the simulator. Now, the simulator is in its original state. Line 6 changes the step function of `sim` into `tracedStep`. This step function generates a trace of the simulation. A trace is a sequence of steps. Each step consists of a list of options, a selected option, and the resulting process. Line 7 instructs the simulator to perform 1 step. For instance, on line 8 and 9, two options are displayed, both being `tau` actions. In line 10, one of the options is selected as the first step, `Step 0`. In lines 12–20, a textual presentation of the process resulting from performing the selected option is displayed. With some effort, it is possible to

see that the selected option corresponded to the skip process of the first alternative of the generator G. In line 22, another step is performed. Note that here, only one option is possible. This option is the communication over channel ~gb. Lines 26–35 display the resulting process. The session is continued below.

```
36  >>> mc = ModelChecker(s0)             # make a model checker for s0
37  >>> pg = mc.getProcessGraph()         # compute process graph for s0
38  chi log(5): Total time: 2.350 sec
39  chi log(5): Average speed: 156.170 states/sec
40  >>> pg.nrOfStates()                   # show number of states
41  367
42  >>> pg.nrOfDeadlockStates()           # show number of deadlock states
43  0
44  >>> pg.nrOfTerminationStates()        # show number of termination states
45  0
46  >>> pgmin = pg.minimize()             # minimize process graph
47  >>> pgmin.nrOfStates()                # show number of states
48  84
49  >>> pgmin.writePSFile("gbme-min.ps")  # write PostScript file of
50                                         # minimized process graph
```

In line 36, a model checker if created for the original process GBME(4, 2), which was stored in programming variable s0. A χ_σ model checker can generate process graphs of (finite) χ_σ processes. Line 37 computes the process graph of GBME(4, 2) and stores it in programming variable pg. The output of this command, shows information about the state space generation process, for example, the total time needed and the number of states per second.

In Lines 40, 42, and 44, the numbers of states, deadlock states, and termination states are computed. We see that there are no deadlock and termination states in the process graph. In Line 47, the process graph is minimized and the result is stored in pgmin. By default, the minimize function reduces process graphs under strong bisimulation. If minimization under branching bisimulation is desired, use pg.minimize('b'), instead. As can be seen on line 48, the number of states is reduced considerably. In line 49, the minimized process graph is written to a PostScript file.

It is possible to attach user-defined *call-backs* to χ_σ transitions. Each time a simulator performs a transition, it also executes the call-backs attached to that transition. The parameters of the action or the delay of the transition are accessible from within the call-back. For example, it is possible to define a call-back that keeps track of the current simulation time. This call-back, which is attached to

delay transitions, has its own (Python) variable that stores the simulation time. Each time a delay transition is performed, the call-back is executed and it increases the value of the current time with the value of the delay. In many situations, such a 'current time call-back' is a practical alternative to using χ's current time expression τ.

Call-backs have several advantages over traces. Firstly, since call-backs are defined in ordinary Python code, all of Python's programming power is available. Secondly, call-backs are usually attached to a relatively small number of actions. This means that during most of the steps of a simulation, no call-back is executed. In order to generate traces, however, each simulation step has to write output to the screen. Therefore, traces are much more time consuming than call-backs. Finally, even though traces contain usually enough information, this does not mean that the information is readily accessible. In fact, since traces are available only after simulation, a considerable amount of parsing of traces is required in order to retrieve information from the traces. In contrast, call-backs have access to data structures representing χ_σ transitions at run time.

Membership equational logic · C

This appendix provides an introduction to membership equational logic (MEL) so that the MEL specifications presented in Chapter 2 can be understood.

MEL is a formalism for algebraic specification. It was developed by Meseguer [138, 43]. In this section we present the syntax and semantics of MEL. In Chapter C.1, we define the minimal MEL syntax. In Section C.2, the semantics of MEL is defined. In Section C.3, we define a more user-friendly syntax for MEL.

C.1 · Syntax of membership equational logic

MEL extends many other equational logics, including order-sorted equational logic [85] in a conservative way and is therefore an expressive algebraic framework. Traditionally in the algebraic specification community, *sort names* denote syntactic classes of terms. In MEL, we will identify these classes by so-called *kind names* and we use *sort names* for another notion, explained below. Kind names are used to categorise terms purely on their syntax. As we will see in Definition C.6, each term has a certain kind. Furthermore, the kind of a term can be derived from the syntax of the term. Kind names are unique identifiers written in some presumed alphabet. We will not give a definition of identifiers, but we do assume there are infinitely many different identifiers.

Definition C.1 · (Kind name) *A kind name is an identifier K.*

Sort names are used to subdivide the syntactic classes of terms defined by kind names. Sorts are useful to split up huge classes of terms into smaller subclasses. For example, the class of numbers can be divided into subclasses of real numbers, integer, etc. In addition, sorts allow for a natural treatment of partial functions. A sort name belongs to exactly one kind name, but a kind name can have more than one sort names. Thus, a sort name is an identifier qualified with a kind name. As

we will explain below, the notion of a term belonging to a sort is based on axioms and derivation rules and can in general not be determined purely from the syntax of a term. Furthermore, a term may have more than one sort (provided that all these sorts have the same kind) or no sort at all.

Definition C.2 · (Sort name) *A sort name is a qualified identifier $s.K$, where s is an identifier and K is a kind name.*

If the kind of a sort name $s.K$ can be determined from the context, we will leave out the kind qualification K and just write s.

Function names are built up from an identifier and a function type. A function type has the form *input kind names* $\rightarrow$ *output kind name*. The input kind names specify the kinds of the parameters and the output kind name specifies the kind of the result of the function. Constants are function names without parameters. There are no special 'constructor' function names; whether or not application of function names construct new terms depends on the equations (see below).

Definition C.3 · (Function name) *A function name consists of an identifier f and a function type $K_1\ K_2\ \ldots\ K_n \rightarrow K$ ($n \geq 0$), where $K_1, K_2, \ldots, K_n, K$ are kind names. It is written like: $f : K_1\ K_2\ \ldots\ K_n \rightarrow K$. Kind names $K_1, \ldots, K_n$ are called the* input kinds *or* input kind names *and K is called the* output kind *or* output kind name *of the function.*

A *signature* is a triple of sets containing kind names, sort names, and function names. Signatures define the basic syntactic elements of MEL specifications. It is required the kind names of the sorts and the kind names in the function types be defined in the signature.

Definition C.4 · (Signature) *A signature Ω is a triple $(\mathcal{K}, \mathcal{S}, \mathcal{F})$ where $\mathcal{K}$ is a set of kinds, $\mathcal{S}$ is a set of sort names, and $\mathcal{F}$ is a set of function names. Furthermore, for all $s.K \in \mathcal{S}$, we require that $K \in \mathcal{K}$; and for all function names $f : K_1 K_2 \ldots K_n \rightarrow K \in \mathcal{F}$, we require that $K_1, K_2, \ldots, K_n, K \in \mathcal{K}$.*

As mentioned above, signatures define the elementary syntax of MEL specifications. More complicated syntactic structures (terms) are composed from function names and *logical variables*, see Definition C.5. Logical variables are place holders and may be replaced by terms of the same kind. Consequently, logical variables have to belong to a certain kind.

Definition C.5 · (Logical variable) *A logical variable is a qualified identifier $x.K$, where x is an identifier and K is a kind name.*

As with sort names, we will leave out the kind qualification K of a variable $x.K$ if it can be determined from the context.

Terms are built up from function names and logical variables. The rules by which we construct terms are listed in Definition C.6. The (identifiers of) nullary function names are basic terms. Logical variables are basic terms, too. More complicated terms are constructed by function application of n-ary function names ($n > 0$). Note that the function type of function names does not occur in terms. The separate treatment of constants is merely for readability. The second rule and the third rule could be merged into one by taking $n \geq 0$.

Definition C.6 · (Terms) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and X a set of variables of kinds in $\mathcal{K}$. The set of terms over Ω parameterised by X is denoted by $Term_\Omega(X)$. The set $Term_\Omega(X)$ is recursively defined by the following rules.*

1. *Every variable $x.K \in X$ is a term of kind K: $x.K \in Term_\Omega(X)$.*
2. *Every nullary function name $f :\rightarrow K \in \mathcal{F}$ is a term of kind K: $f :\rightarrow K \in Term_\Omega(X)$.*
3. *If $t_1, t_2, \ldots, t_n \in Term_\Omega(X)$ are terms of kinds $K_1\ K_2\ \ldots\ K_n$ (for $n > 0$) respectively, and $f : K_1\ K_2\ \ldots\ K_n \rightarrow K \in \mathcal{F}$ is a function name, then $f(t_1, t_2, \ldots, t_n) \in Term_\Omega(X)$ is a term of kind K.*

Terms that do not contain variables are called *closed* terms.

Variables occurring in terms can be replaced by other terms (of the same kind). Formally, this is done by *substitutions*: functions from variables to terms. We assume substitutions are total functions. Since variables may be mapped onto themselves, this is not really a limitation. For example, if a variable x is undefined in a substitution θ, we define $\theta(x) = x$. If x occurs in a term t and we apply θ to t (see below), the resulting term will still have the original occurrences of x.

Definition C.7 · (Substitution) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and X a set of variables of kinds in $\mathcal{K}$. A* substitution *is a function $\theta : X \rightarrow Term_\Omega(X)$ such that for all $x.K \in X$ the term $\theta(x.K)$ is of kind K.*

We use the notation $\theta[t/x]$ to denote the following substitution:

$$\theta[t/x](x') = \begin{cases} t & \text{if } x = x' \\ \theta(x') & \text{if } x \neq x'. \end{cases}$$

Below, we will apply substitutions not only to variables, but also to terms. For example, we often write $\theta(t)$ for some term t. This represents the term t in which

all variables are replaced by their values according to θ. Thus, a substitution can be *extended* to terms.

Definition C.8 · (Extending a substitution) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and X a set of variables of kinds in Ω. Let $x \in X$ be a variable, $c :\rightarrow K$ a nullary function name in Ω; $t_1, \ldots, t_n$ be terms in $Term_\Omega(X)$; and f an n-ary function name such that $f(t_1, \ldots, t_n) \in Term_\Omega(X)$ is a term. The* extended substitution *$\overline{\theta}$ of a substitution $\theta : X \rightarrow Term_\Omega(X)$ is a function $\overline{\theta} : Term_\Omega(X) \rightarrow Term_\Omega(X)$ defined by $\overline{\theta}(x) = \theta(x)$, $\overline{\theta}(c) = c$, and $\overline{\theta}(f(t_1, \ldots, t_n)) = f(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n))$.*

Usually, we will implicitly extend a substitution and write θ instead of $\overline{\theta}$.

In MEL, there are two kinds of atomic formulas, namely, equations and membership assertions. An equation says that two terms are equal and a membership assertion says that a term has a certain sort. A sentence is a pair of an atomic formula and a *condition*, where a condition is a list of atomic formulas. The informal meaning of a sentence is that it should be *true* if the atomic formulas of the condition are *true*. That is, a sentence is an implication. As with atomic formulas, there are also two types of compound formulas, namely, conditional equations and conditional membership assertions.

Definition C.9 · (Formulas) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and X a set of variables of kinds in $\mathcal{K}$.* Atomic formulas *of membership equational logic are* equations *and* membership assertions. *We use the character a (possibly indexed) to range over equations or membership assertions.*
Let $t_1, t_2 \in Term_\Omega(X)$ be terms of the same kind.

1. *An* equation *has the form: $t_1 = t_2$.*

Let $t \in Term_\Omega(X)$ be a term of kind K and $s.K \in \mathcal{S}$ a sort name.

2. *A* membership assertion *has the form: $t : s.K$.*

Sentences *(or* compound formulas*) of membership equational logic are* conditional equations *and* conditional membership assertions.
Let $t_1, t_2 \in Term_\Omega(X)$ be terms and $a_1, \ldots, a_n$ ($n \geq 0$) be atomic formulas.

3. *A* conditional equation *has the form: $t_1 = t_2 \Leftarrow a_1, \ldots, a_n$.*

Let $t \in Term_\Omega(X)$ be a term of kind K, $s.K \in \mathcal{S}$ a sort name, and $a_1, \ldots, a_n$ ($n \geq 0$) equations or membership assertions.

4. *A* conditional membership assertion *has the form: $t : s.K \Leftarrow a_1, \ldots, a_n$.*

We now have all ingredients to construct *MEL specifications*. Note that there are also *MEL theories* and *MEL views*, but they are not introduced until Section C.3. A MEL specification is a signature, a set of equations, and a set of membership assertions. The terms occurring in the equations and membership assertions should comply to the signature of the specification.

Definition C.10 · (Membership equational specification) *A membership equational specification is a tuple* (Ω, E, M) *where* Ω *is a signature,* E *is a set of equations and conditional equations over* Ω*, and* M *is a set of membership assertions and conditional membership assertions over* Ω*.*

A MEL specification defines the syntax of terms. In addition, a MEL specification defines which terms are considered equal and which terms belong to a certain sort. So, the sentences of a MEL specification play the role of axioms. Using the axioms of a MEL specification, it is sometimes possible to derive that two terms are equal in that specification, or that a term belongs to a certain sort. As in any formal logic, the derivation is a purely syntactic game governed by *derivation rules*.

Definition C.11 · (Derivation in MEL) *Let* $T = (\Omega, E, M)$ *be a membership equational specification with* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$*,* X *a set of variables of kinds in* $\mathcal{K}$*;* $f : K_1 \dots K_n \to K$ *an* n*-ary function name,* t*,* t'*,* t''*,* $t_1 \dots t_n$*,* $r \in Term_\Omega(X)$*;* $a_1, \dots, a_n$ *equations or membership assertions;* K*,* $K' \in \mathcal{K}$*;* $s.K$*,* $s.K' \in \mathcal{S}$*; and* θ *a substitution. The MEL derivation relation* $\vdash$*' (for* $0 < i \leq n$*) is defined by*

1. *equational axiom: if* $t = t' \in E$ *then* $T \vdash \theta(t) = \theta(t')$*,*
2. *membership axiom: if* $r : s.K \in M$ *then* $T \vdash \theta(r) : s.K$*,*
3. *conditional equational axiom: if* $t = t' \Leftarrow a_1, \dots, a_n \in E$ *and* $T \vdash \theta(a_i)$*, then* $T \vdash \theta(t) = \theta(t')$*,*
4. *conditional membership axiom: if* $r : s.K \Leftarrow a_1, \dots, a_n \in M$ *and* $T \vdash \theta(a_i)$*, then* $T \vdash \theta(r) : s.K$*,*
5. *subject reduction: if* $T \vdash t' : s.K$ *and* $T \vdash t = t'$ *then* $T \vdash t : s.K$*,*
6. *reflexivity:* $T \vdash t = t$*,*
7. *symmetry: if* $T \vdash t = t'$ *then* $T \vdash t' = t$*,*
8. *transitivity: if* $T \vdash t = t'$ *and* $T \vdash t' = t''$*, then* $T \vdash t = t''$*,*
9. *congruence: if* $T \vdash t_i = t_i'$*, then* $T \vdash f(t_1, \dots, t_n) = f(t_1', \dots, t_n')$*.*

C.2 · Semantics of membership equational logic

We will now define the semantics of membership equational logic. In literature on membership equational logic, the semantics is usually defined using category theory. Since we do not presume the reader is familiar with category theory, we define the semantics using set theory only.

The mathematical structures we use to interpret a MEL specification are algebras. An algebra is a system of sets and functions satisfying certain properties. Firstly, there should be a set for every kind name of the MEL specification. Secondly, there should be a set for every sort name of the MEL specification such that it is a subset of the set for the kind of the sort name. Finally, there should be a function for every function name of the MEL specification, such that the domain of the function is the Cartesian product of the sets for the input kinds of the function name and the range of the function is the set for the kind of the output kind of the function name. It follows that algebras depend on the signatures of a MEL specification. Therefore, we call them Ω *Algebras*. For a particular signature Ω, there are usually many Ω algebras.

Definition C.12 · (Ω Algebra) *An Ω Algebra A for a signature $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$, is a triple $(\mathcal{K}^A, \mathcal{S}^A, \mathcal{F}^A)$ of sets, where for each $K \in \mathcal{K}$ there is a set $K^A \in \mathcal{K}^A$; for each $s.K \in \mathcal{S}$ there is a set $(s.K)^A \in \mathcal{S}^A$ such that $(s.K)^A \subseteq K^A$; and for each $f : K_1\, K_2\, \ldots\, K_n \to K \in \mathcal{F}$ there is a function $f^A : K_1^A \times \ldots \times K_n^A \to K^A \in \mathcal{F}^A$.*

Note that nullary function names, $c :\to K$, are mapped to elements of the set K^A, that is, c is mapped to an element $c^A \in K^A$.

Knowing the structure of Ω algebras, it is not surprising to see that the K^A sets are the interpretations of the kind names of a MEL specification, the $s.K^A$ sets are the interpretations of the sort names, and the f^A functions are the interpretations of the function names. This means that terms of a certain kind K (or sort $s.K$) are interpreted as elements of the corresponding set K^A or $(s.K^A)$. The variables that may occur in the axioms of a MEL specification stand for arbitrary terms. The usual way to deal with variables in formal logics is by *valuations*: functions from (syntactic) variables to (semantic) elements of algebras.

Definition C.13 · (Valuation) *Let X be a set of variables and A an Ω algebra for signature $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$. A valuation is a function $v : X \to \bigcup_{K \in \mathcal{K}} K^A$, such that $v(x.K) \in K^A$ for all $x \in X$.*

The interpretation of terms of a MEL specification in an Ω algebra can now be defined recursively in a straightforward way. We use the usual notation $[\![t]\!]_v^A$ to denote the interpretation of a term t in the Ω algebra A with valuation v. Usually the algebra A is known from the context and we just write $[\![t]\!]_v$. As in the definition of terms, Definition C.6, we treat constants separately, but this is merely for readability.

Definition C.14 · (Interpretation) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature, $x \in X$ be a variable, $c :\rightarrow K \in \mathcal{F}$ a nullary function name, and $f : K_1 \dots K_n \rightarrow K' \in \mathcal{F}$ a n-ary function name. The interpretation of a term $t \in Term_\Omega(X)$ in an Ω algebra A is denoted by $[\![t]\!]_v^A$, where v is a valuation. It is defined by*

1. $[\![x]\!]_v^A = v(x)$,
2. $[\![c]\!]_v^A = c^A$,
3. $[\![f(t_1, \dots, t_n)]\!]_v^A = f^A([\![t_1]\!]_v^A, \dots, [\![t_n]\!]_v^A)$.

A *model* of a MEL specification is an Ω algebra in which all axioms of the specification are satisfied. That is, if $t : s.K$ and $t = t'$ are axioms in a MEL specification, then for an Ω algebra to be a model, $[\![t]\!]_v \in (s.K)^A$ and $[\![t]\!]_v = [\![t']\!]_v$ should hold for all valuations v. Below, we will use the notation $[\![a]\!]_v$ where a is a membership or equational axiom. If a is a membership axiom $t : s.K$, the notation $[\![a]\!]_v$ is an abbreviation for $[\![t]\!]_v \in (s.K)^A$. If a is an equational axiom $t = t'$, the notation $[\![a]\!]_v$ is an abbreviation for $[\![t]\!]_v = [\![t']\!]_v$

Definition C.15 · (Membership equational model) *Let $T = (\Omega, E, M)$ be an equational specification with $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$. An Ω algebra A is a* model *of T if it satisfies the following conditions:*

1. *if $t_1 = t_2 \in E$, then $[\![t_1]\!]_v = [\![t_2]\!]_v$ should hold for all valuations v,*
2. *if $t : s.K \in M$, then $[\![t]\!]_v \in (s.K)^A$ should hold for all valuations v,*
3. *if $t_1 = t_2 \Leftarrow a_1, \dots a_n \in E$ and $[\![a_1]\!]_v$, …, $[\![a_n]\!]_v$ hold, then $[\![t_1]\!]_v = [\![t_2]\!]_v$ should hold for all valuations v,*
4. *if $t : s.K \Leftarrow a_1, \dots a_n \in M$ and $[\![a_1]\!]_v$, …, $[\![a_n]\!]_v$ hold, then $[\![t]\!]_v \in (s.K)^A$ should hold for all valuations v.*

We use the notations $A, v \models t = t'$ and $A, v \models t : s.K$ to denote that $[\![t]\!]_v = [\![t']\!]_v$ and $[\![t]\!]_v \in (s.K)^A$ hold for a particular algebra A and valuation v. If it holds for all valuations, we write $A \models t = t'$ and $A \models t : s$, respectively.

As mentioned above, a signature Ω usually admits more than one Ω algebra. Some of the algebras have the same structure, meaning that the result of a function application in one algebra equals the result of the corresponding function in the other algebra. A *homomorphism* is a function between Ω algebras that preserves these structural properties. As such, homomorphisms are well suited to compare different Ω algebras.

Definition C.16 · (Homomorphism) *Let A and A' be Ω algebras, where $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ and let $c_1, \ldots, c_n \in \bigcup_{K \in \mathcal{K}} K^A$. A function $h : \bigcup_{K \in \mathcal{K}} K^A \to \bigcup_{K \in \mathcal{K}} K^{A'}$ is a homomorphism from A to A' if it satisfies*

$$h(f^A(c_1, \ldots, c_n)) = f^{A'}(h(c_1), \ldots, f(c_n)).$$

Usually, we write $h : A \to A'$ to denote a homomorphism h from A to A'. A homomorphism that is surjective and injective is called an *isomorphism*. If there exists an isomorphism between two Ω algebras, they are isomorphic. This means that the algebras only differ in representation, if they differ at all, but not in structure.

A well known concept from algebraic specification is *initiality of an Ω algebra* with respect to a MEL specification. An Ω algebra A is *initial* for a MEL specification with signature Ω if there exists a unique homomorphism from A to any Ω algebra A'.

Definition C.17 · (Initiality) *An Ω algebra A is* initial *if for any Ω algebra A' there is a unique homomorphism $h : A \to A'$.*

It can be shown that for any membership equational specification $T = (\Omega, E, L)$, there exists a unique initial Ω algebra for T, modulo isomorphism. Therefore, it is legitimate to speak about *the* initial algebra of a membership equational specification.

Initial algebras are useful since they have some interesting properties. First of all, every element in an initial algebra has a closed term representation in the concerning MEL specification. This property is called 'no junk' and can be paraphrased as "there exist no elements in the initial algebra that cannot be represented by closed terms in the MEL specification." In addition, in an initial algebra two closed terms are equal if and only if they can be proven equal by formal derivation in the concerning MEL specification. This property is called 'no confusion' and can be paraphrased as "everything that holds in the MEL specification, holds in the initial algebra of the specification and vice versa."

Property C.18 · (No junk and no confusion) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and A an initial algebra for a membership equational specification $T = (\Omega, E, M)$. Then A has the following three properties,*

1. *no junk: for each element c of a set K^A, there is a closed term $t \in Term_\Omega$, such that $[\![t]\!] = c$,*
2. *no confusion: for all closed terms $t, t' \in Term_\Omega$: $A \models t = t'$ if and only if $T \vdash t = t'$,*
3. *no sort confusion: for all closed terms $t \in Term_\Omega$ and sort names $s.K \in \mathcal{S}$: $A \models t : s.K$ if and only if $T \vdash t : s.K$.*

These properties are well known in theory about algebraic specifications and therefore, we do not provide their proofs.

Summarising the discussion so far, we have achieved the following. Ω algebras are mathematical structures of sets and functions such that every kind name and sort name of the signature corresponds to a set and every function name corresponds to a function. Furthermore, the sets for the sort names are subsets of the sets of their corresponding kind names. Terms of a MEL specification $T = (\Omega, E, M)$ can be interpreted in an Ω algebra. Basic terms and nullary function names (constants) are interpreted as nullary functions and compound terms are interpreted recursively as the function applications of the function names in the term. Valuations are used to deal with the interpretation of variables occurring in the terms. An Ω algebra is a model of the MEL specification T, if the (interpretation of the) axioms in E and M hold in the algebra. In general, there is more than one model for a MEL specification. An initial algebra is an Ω algebra with 'no junk' and 'no confusion'. Every MEL specification has a unique (modulo isomorphism) initial algebra.

C.3 · A specification language for MEL

In this section, we describe a specification language for MEL. The specifications of Chapter 2 are written in this language. The language alleviates the writing process of MEL specifications and it defines a uniform format, which makes it easier to understand MEL specifications. Furthermore, the language provides some structuring mechanisms by which specifications can be built up from other

specifications. We explain how specifications written in this language should be interpreted in terms of the definitions of Sections C.1 and C.2.

We use the keywords **spec** and **end** to define a membership equational specification. By convention, the name of a specification is written in (small) capitals. The contents of a specification is split up into so-called *theory-sections*. Successive theory-sections are separated by white space. The general form is given below.

spec THEORYNAME
⟨*theory section*⟩
⋮
⟨*theory section*⟩
end

The basic theory-sections are *kinds*, *operators*, *variables*, *memberships*, and *equalities*. Each theory-section starts with a keyword and ends with a '.' (period). Theory-section keywords have a single and a plural form. For example, the *kinds* section starts with either **kind** or **kinds**. There is no semantical difference between the single and plural version of a theory-section keyword; choices to use one form instead of the other should be based on aesthetic grounds. In a theory-section, a sequence of comma separated entities is defined. For example, in a kind theory-section, kind names are defined, in an operator theory-section, operators are defined, etc.

We first describe the **kind** theory-section. In this section we define kind names, as in:

$$\begin{array}{ll} \textbf{kinds} & K_1[s_{1,1}, \ldots, s_{1,n_1}], \\ & \vdots \\ & K_m[s_{m,1}, \ldots, s_{m,n_m}]. \end{array}$$

This theory-section defines kind names K_i (where $0 < i \leq m$) and sort names $s_{i,j}$ (where $0 < i \leq m$ and $0 < j \leq n_i$). Each sort name $s_{i,j}$ is implicitly qualified with the kind name K_i. That is, the full sort names are actually $s_{i,j}.K_i$ (see Definition C.2). However, we almost never use this explicit notation. As an example of a **kind** section, we give a definition of the 'Number' kind with sorts nat, int, and rat (See also Sections 2.3–2.5):

kind Number[nat, int, rat].

Function names are defined in theory-sections starting with the keyword **operator**(**s**):

$$\textbf{operator } f : K_1 \dots K_n \to K.$$

Here, K, K_i $(0 < i \leq n)$ are kind names. Each kind name occurring in a function name definition should be defined in a kind theory-section. Recall from Section C.1 that there is no difference between constructors and operators. Therefore, instead of using the keyword **operator**, one can also use **constructor**.

Note that operators defined like this have a prefix syntax. In order to define operators with a non-prefix syntax, the underscore symbol '_' can be used. For example, the boolean infix operator $\wedge$ can be defined by the following function name definition:

$$\textbf{operator } \_ \wedge \_ : BB \to B.$$

Here, we assume that B is the kind name of the booleans. Using underscores, arbitrary 'mixfix' operators can be defined. For example,

$$\textbf{operator } \mathit{if\_then\_else\_} : B\ B\ B \to B.$$

The number of underscores in a function name definition should be 0 (zero) if the function name is written in prefix notation, and equal to the arity of the function name if it is written in mixfix notation.

Theory-sections starting with the keyword **var**(**s**) define variables of certain kinds. The variables are used to define membership axioms and equational axioms (see below). For example, the following theory-section defines three variables, two of kind K_1, and one of kind K_2:

$$\begin{aligned} \textbf{vars} \quad & x,\ y\ : K_1, \\ & x'\ : K_2. \end{aligned}$$

Sections starting with the keyword **membership**(**s**) define membership assertions (see items 2 and 4 of Definition C.9). Since these membership assertions are supposed to hold in any model of the specification, they are sometimes called (membership) axioms. Consider, for example, the following membership axioms:

$$\begin{aligned} \textbf{membership} \quad & t : s, \\ & t : s \Leftarrow a_1, \dots, a_n. \end{aligned}$$

Here, t is a term built up from function names and variables defined in **operator** sections and **var** sections, respectively; s is a sort name defined in a **kind** section or in a **sort** section; and $a_1, \ldots, a_n$ are unconditional equations or membership assertions, possibly containing variables defined in **var** sections.

Theory-sections starting with the keyword **equation**(**s**) define equations (see item 1 and 3 of Definition C.9). Since these equations are supposed to hold in any model of the specification, they are sometimes called (equational) axioms:

$$\begin{aligned} \textbf{equation} \quad & t_1 = t_2, \\ & t_1 = t_2 \Leftarrow a_1, \ldots, a_n. \end{aligned}$$

The terms t_1 and t_2 are built up from function names and variables defined in **operator** sections and **var** sections, respectively and $a_1, \ldots, a_n$ are unconditional equations or membership assertions possibly containing variables defined in **var** sections.

With the keyword **sort**(**s**), sort names can be introduced without defining a kind name explicitly. Using this keyword, the specification writer does not have to bother about the actual kind names; he can just assume that for every sort name he defines, there exists a kind name. In order to make the kind names explicit, only one fresh kind name has to be added to the specification. All sort names without a kind name belong to this new kind name. In the specifications of Chapter 2, this approach to define sort names is preferred over the explicit kind name approach.

The keyword **subsorts**(**s**) starts theory-sections in which *subsort relations* are defined. As we will explain below, subsort relations are not new features, since they can be defined in terms of membership assertions. In fact, subsort relations are just syntactic sugar. Consider, for example, the following definitions:

$$\begin{aligned} \textbf{subsorts} \quad & s_1 < s_2, \\ & s_1 < s_3 < s_4. \end{aligned}$$

Each sort name occurring in a subsort section, has to be defined in a kind theory-section or a sort theory-section. A subsort declaration is a conditional membership axiom in disguise. For example, to get the same results with a membership theory-

section only, we could write

$$\begin{aligned} \textbf{memberships} \quad & t : s_2 \Leftarrow t : s_1, \\ & t : s_3 \Leftarrow t : s_1, \\ & t : s_4 \Leftarrow t : s_3. \end{aligned}$$

Membership assertions and equations of a MEL specification may be labelled for referencing purposes. In fact, we have labelled each membership assertion and each equations of the specifications given in Chapter 2. A label is a string enclosed in square brackets, like [LABEL0], and it precedes the membership assertion or equation it labels. To prevent labels from standing out in the text of a MEL specification, we usually use a smaller font for labels:

$$\begin{aligned} \textbf{equation} \quad & {\scriptstyle [E1]} \quad s\ p\ 0 = 0, \\ & {\scriptstyle [E2]} \quad p\ s\ 0 = 0. \end{aligned}$$

As a notational convention, function names may be defined using sort names instead of kind names. For example, the following theory-sections are allowed in MEL specifications:

kind Number[int].
operator *succ* : int → int.

This is an abbreviation for the theory-sections:

kind Number[int].
operator *succ* : Number → Number.
var n : Number.
membership $succ(n)$: int ⇐ n : int.

That is, a unary operator *succ* from the kind Number to the kind Number is defined together with a membership axioms stating that $succ(n)$ is of sort int if n is of sort int.

Similarly, variables can be defined using sort names instead of kind names. The following theory-sections together are equivalent to the previous two examples:

operator *succ* : Number → Number.

var n : int.

membership $succ(n)$: int.

Here, it looks like the membership axiom is unconditional, but this is not really the case; every axiom in which the variable n occurs is conditional and one of the conditions is n : int.

In order to structure specifications, that is composing specifications from (smaller) specifications, there are theory-sections to import specifications. The keyword of these sections is **protecting**. Suppose S is the name of a specification, then it can be imported in another specification by the section

protecting S.

The importing specification can use all sorts, constructors, and operators of S.

Now, we explain MELs mechanism to parameterize specifications. The bottom up approach would be to first describe MEL theories, then MEL views, and finally parameterized specifications and theories. However, since MEL theories and MEL views are just means to implement parameterisation, we start by describing parameterised specifications. After that, we describe MEL theories and MEL views. Finally, we describe the difference between MEL theories and ordinary MEL specifications.

A powerful structuring mechanism for MEL specifications is parameterisation. MEL specifications can be parameterised by other MEL specifications. For example, a specification of finite sets could be parameterised by a *theory* of elements (see also Section 2.9):

spec SET[X :: ELEMENT] ... **end**

where ELEMENT is a MEL theory describing properties of set-elements needed to specify SET. The only property required to specify finite sets, is that there is a sort of elements (of the set). Therefore, the MEL theory ELEMENT-EXAMPLE is very simple:

theory ELEMENT-EXAMPLE

sort E.
end

Note that MEL theories start with the **theory** keyword. To get a specification of, say, sets of booleans, one can instantiate SET with BOOL, as in SET[BOOL]. Informally, this means that whenever a sort, constructor, or operator of MEL is used in SET, it is replaced by the 'corresponding' sort, constructor, or operator of BOOL. In order to explain what we mean by 'corresponding' in this context, we have to define a mapping from ELEMENT-EXAMPLE to BOOL. Such mappings are called MEL views. A view from ELEMENT-EXAMPLE to BOOL has to map the sort elt onto a sort of BOOL. In general, there are many views from a theory to a specification (or to another theory). However, since BOOL has only one sort, there is no choice. Consequently, the view is defined by

view BOOL-EXAMPLE
from ELEMENT-EXAMPLE **to** BOOL
sort elt **to** bool.
end

In the formal discussion of Sections C.1 and C.2, parameterisation was not described. The reason for this omission is that parameterisation is probably best described formally in a categorical setting and that would be outside the scope of this document. A general treatment of parameterisation of algebraic specifications can be found in [9]. For a formal discussion on parameterisation in MEL we refer to [138, 43]. Informally, the interpretation of a parameterised theory is a function from the class of algebras for its parameter to the class of algebras for the specification you get after substituting the formal parameter by an actual specification.

Note that the semantics of a MEL theory cannot be the initial semantics, since that would be too restrictive. For instance, the initial semantics of ELEMENT-EXAMPLE is an algebra wit just one empty set onto which the sort elt is mapped. However, this interpretation does not allow us to map the sort elt onto, for example, the sort bool of BOOL, since that sort has a set with two elements as its interpretation in the initial algebra of BOOL. A solution is to allow all algebras that have (at least) a set for each sort name and a function for each operator as valid interpretations of MEL theories. This kind of 'loose semantics' is used frequently in algebraic specifications, since it gives us opportunities to leave (parts of) specifications open

to many interpretations. Only after complete instantiation, will the initial algebra semantics hold.

Consequently, there is a clear distinction between MEL specifications and MEL theories: the former have an initial algebra semantics whereas the latter have a loose semantics. As such, the MEL specifications should be used to define concrete data types and MEL theories should be used to define formal parameters of parameterised specifications or theories. Of course, there subtleties involved in situations where MEL specifications include (using the **include** keyword) MEL theories and vice versa, or if a parameterised specification or theory is instantiated with another parameterised specification or theory. A thorough treatment of these subtleties is outside the scope of this document.

List of definitions and lemmas · D

Definition 4.1 [p. 56] · (LTS) *An LTS is a triple* $(S, R_{S\times S}, R_S)$*, with* S *a set of states,* $R_{S\times S}$ *a set of binary relations on states, and* R_S *a set of unary relations on states.*

Definition 4.2 [p. 57] · (χ_σ-LTS) *A* χ_σ*-LTS, is an LTS* $(S, R_{S\times S}, R_S)$*, such that*

- $S \subseteq C(P)$,
- *all* $r \in R_{S\times S}$ *are binary relations on closed* χ_σ *processes given by triples*
  - $(\sigma, a, \sigma') \in \mathit{Stack} \times \mathit{Action} \times \mathit{Stack}$ *or*
  - $(\sigma, d, \sigma') \in \mathit{Stack} \times R_{>0} \times \mathit{Stack}$*, and*
- *all* $r \in R_S$ *are unary relations on closed* χ_σ *processes given by stacks* $\sigma \in \mathit{Stack}$.

Definition 4.3 [p. 59] · (Formulas) *A formula has one of the following forms, where* $e_b \in \mathrm{bool}$*,* $p, p' \in P$*,* $\sigma, \sigma' \in \mathit{Stack}$*,* $a \in \mathit{Action}$*, and* $d \in R_{>0}$*:*

1. $\textsc{true}(e_b)$,
2. $\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$,
3. $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle$,
4. $\langle\, p, \sigma \,\rangle\!\downarrow$,
5. $\neg\exists p' \in C(P), \sigma', a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$,
6. $\neg\exists p' \in C(P), \sigma', d : \langle\, p, \sigma \,\rangle \xrightarrow{d} \langle\, p', \sigma' \,\rangle$,
7. $\neg\langle\, p, \sigma \,\rangle\!\downarrow$.

Definition 4.4 [p. 59] · (Deduction rule) *A deduction rule consists of a set of formulas H and a formula c. H is the set of* hypotheses *and c is the* conclusion. *Furthermore, c is of the form 2, 3, or 4 of Definition 4.3. A deduction rule is denoted by $\frac{H}{c}$.*

Definition 4.5 [p. 64] · (Strong Bisimulation) *A strong bisimulation on processes is a relation $R \in P \times P$ such that for all $(p, q) \in R$ the following holds:*

1. $\forall \sigma : \langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\!\downarrow$,
2. $\forall \sigma, a, p', \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \wedge (p', q') \in R$,
3. $\forall \sigma, a, q', \sigma' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \wedge (p', q') \in R$,
4. $\forall \sigma, d, p', \sigma' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \Rightarrow \exists q' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \wedge (p', q') \in R$,
5. $\forall \sigma, d, q', \sigma' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \Rightarrow \exists p' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \wedge (p', q') \in R$.

Two processes p and q are strongly bisimilar, denoted by $p \leftrightarrows q$, if there exists a bisimulation relation R such that $(p, q) \in R$.

Definition 4.6 [p. 65] · (Bisimulation up to $\leftrightarrows$) *A relation $R \in P \times P$ on processes is a 'bisimulation up to $\leftrightarrows$' if for all $(p, q) \in R$ the following holds:*

1. $\forall \sigma : \langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, q, \sigma \,\rangle\!\downarrow$,
2. $\forall \sigma, a, p', \sigma' : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \Rightarrow$
   $\exists p'', q', q'' : p' \leftrightarrows p'' \wedge \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \wedge q' \leftrightarrows q'' \wedge (p'', q'') \in R$,
3. $\forall \sigma, a, q', \sigma' : \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle \Rightarrow$
   $\exists q'', p', p'' : q' \leftrightarrows q'' \wedge \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle \wedge p' \leftrightarrows p'' \wedge (p'', q'') \in R$,
4. $\forall \sigma, d, p', \sigma' : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \Rightarrow$
   $\exists p'', q', q'' : p' \leftrightarrows p'' \wedge \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \wedge q' \leftrightarrows q'' \wedge (p'', q'') \in R$,
5. $\forall \sigma, d, q', \sigma' : \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle \Rightarrow$
   $\exists q'', p', p'' : q' \leftrightarrows q'' \wedge \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \wedge p' \leftrightarrows p'' \wedge (p'', q'') \in R$.

Lemma 4.7 [p. 65] · *Let R be a 'bisimulation up to $\leftrightarrows$' relation and p and q be processes. If $(p, q) \in R$ then $p \leftrightarrows q$.*

Definition 4.8 [p. 68] · (Atomic processes) *The atomic processes of χ_σ have the following signature with $Expr_R$ the set of real number expressions:*

$$\begin{array}{rcl} \varepsilon & : & P, \\ \delta & : & P, \\ \mathsf{skip} & : & P, \\ \Delta\_ & : & Expr_R \to P, \\ \_:=\_ & : & Id \times Expr \to P, \\ \_!\_ & : & Channel \times Expr \to P, \\ \_?\_ & : & Channel \times Id \to P. \end{array}$$

The deduction rules for χ_σ's atomic processes are listed in Table 4.1.

Lemma 4.9 [p. 69] · $\Delta 0 \leftrightarrows \varepsilon$.

Definition 4.10 [p. 70] · (Guard operator) *The guard operator has the following signature with* bool *the set of boolean expressions according to specification* BOOL *from Section 2.2:*

$$\_:\!\to\_ \quad : \quad \mathrm{bool} \times P \to P.$$

The deduction rules for the guard operator are listed in Table 4.2.

Lemma 4.11 [p. 71] · *Let p be a process, then*

$$true :\!\to p \leftrightarrows p.$$

Lemma 4.12 [p. 72] · *Let p be a process, then*

$$false :\!\to p \leftrightarrows \delta.$$

Definition 4.13 [p. 73] · (Alternative composition operator) *The alternative composition operator has the following signature:*

$$\_ \between \_ \quad : \quad P \times P \to P.$$

The deduction rules for the alternative composition operator are listed in Table 4.3.

Lemma 4.14 [p. 73] · *Let p be a process, then*

$$p \between \delta \leftrightarrows p.$$

Lemma 4.15 [p. 74] · *Let p be a process, then*

$$p \parallel p \leftrightarroweq p.$$

Lemma 4.16 [p. 75] · *Let p and q be processes, then*

$$p \parallel q \leftrightarroweq q \parallel p.$$

Lemma 4.17 [p. 75] · *Let p, q, and r be processes, then*

$$(p \parallel q) \parallel r \leftrightarroweq p \parallel (q \parallel r).$$

Definition 4.18 [p. 76] · (Sequential composition operator) *The sequential composition operator has the following signature:*

$$\_ \,;\, \_ \;:\; P \times P \to P.$$

The deduction rules for the sequential composition operator are listed in Table 4.4.

Lemma 4.19 [p. 77] · *Let p be a process, then*

$$p \,;\, \varepsilon \leftrightarroweq p.$$

Lemma 4.20 [p. 78] · *Let p be a process, then*

$$\varepsilon \,;\, p \leftrightarroweq p.$$

Lemma 4.21 [p. 79] · *Let p be a process, then*

$$\delta \,;\, p \leftrightarroweq \delta.$$

Lemma 4.22 [p. 80] · *Let p, q, and r be processes, then*

$$(p \,;\, q) \,;\, r \leftrightarroweq p \,;\, (q \,;\, r).$$

Lemma 4.23 [p. 83] · *Let p, q, and r be processes, then*

$$(p \parallel q) \,;\, r \leftrightarroweq p \,;\, r \parallel q \,;\, r.$$

Definition 4.24 [p. 89] · (Repetition operator) *The repetition operator has the following signature:*

$$\_^* \quad : \quad P \to P.$$

The deduction rules for the repetition operator are listed in Table 4.5.

Lemma 4.25 [p. 90] · $\delta^* \leftrightarrows \varepsilon$.

Lemma 4.26 [p. 90] · *Let p be a process, then*

$$p^* \leftrightarrows p \ ; \ p^* \between \varepsilon.$$

Definition 4.27 [p. 92] · (Parallel composition operator) *The parallel composition operator has the following signature:*

$$\_ \parallel \_ \quad : \quad P \times P \to P.$$

The deduction rules for the parallel composition operator are listed in Table 4.6.

Lemma 4.28 [p. 93] · *Let p be a process, then*

$$\varepsilon \parallel p \leftrightarrows p.$$

Lemma 4.29 [p. 94] · *Let p and q be processes, then*

$$p \parallel q \leftrightarrows q \parallel p.$$

Lemma 4.30 [p. 94] · *Let p, q, and r be processes, then*

$$(p \parallel q) \parallel r \leftrightarrows p \parallel (q \parallel r).$$

Definition 4.31 [p. 96] · (State operator) *The state operator has the following signature:*

$$[\![\_ \mid \_]\!] \quad : \quad State \times P \to P.$$

The deduction rules for the state operator are listed in Table 4.7.

Lemma 4.32 [p. 97] · *Let p be a process, then*

$$[\![\lambda_s \mid p]\!] \leftrightarrows p.$$

Lemma 4.33 [p. 98] · *Let s_0 and s_1 be states and let p be a process, then*

$$[\![s_0 \mid [\![s_1 \mid p]\!]]\!] \leftrightarrows [\![set(s_0, s_1) \mid p]\!].$$

Lemma 4.34 [p. 101] · *Let s and s' be states, p and p' be processes, σ and σ' be stacks, a be an action, and d be a positive real number. Then*

$$\langle [\![s \mid p]\!], \sigma \rangle \xrightarrow{a} \langle [\![s' \mid p']\!], \sigma' \rangle \Rightarrow dom(s) = dom(s'),$$
$$\langle [\![s \mid p]\!], \sigma \rangle \overset{d}{\longmapsto} \langle [\![s' \mid p']\!], \sigma' \rangle \Rightarrow dom(s) = dom(s').$$

Lemma 4.35 [p. 102] · *Let p and p' be a processes, s and s' be states, σ and σ' be stacks, i be an identifier, c be a value, a be an action, and d be a positive real value. Then for $i \in dom(s)$ we have*

$$\langle [\![s \mid p]\!], \sigma \rangle\downarrow \Leftrightarrow \langle [\![s \mid p]\!], \sigma[c/i] \rangle\downarrow,$$
$$\langle [\![s \mid p]\!], \sigma \rangle \xrightarrow{a} \langle [\![s' \mid p']\!], \sigma' \rangle \Leftrightarrow \langle [\![s \mid p]\!], \sigma[c/i] \rangle \xrightarrow{a} \langle [\![s' \mid p']\!], \sigma'[c/i] \rangle,$$
$$\langle [\![s \mid p]\!], \sigma \rangle \overset{d}{\longmapsto} \langle [\![s' \mid p']\!], \sigma' \rangle \Leftrightarrow \langle [\![s \mid p]\!], \sigma[c/i] \rangle \overset{d}{\longmapsto} \langle [\![s' \mid p']\!], \sigma'[c/i] \rangle.$$

Definition 4.36 [p. 103] · (Encapsulation operator) *The encapsulation operator has the following signature:*

$$\partial_{\_} \quad : \quad \mathcal{P}(Action) \times P \to P.$$

The deduction rules for the encapsulation operator are listed in Table 4.8.

Lemma 4.37 [p. 103] · *Let A and A' be sets of actions and let p be a process, then*

$$\partial_A(\partial_{A'}(p)) \leftrightarrows \partial_{A \cup A'}(p).$$

Definition 4.38 [p. 105] · (Maximal progress operator) *The maximal progress operator has the following signature:*

$$\pi \quad : \quad P \to P.$$

The deduction rules for the maximal progress operator are listed in Table 4.9.

Lemma 4.39 [p. 105] · *Let p be a process, then*

$$\pi(\pi(p)) \leftrightarrows \pi(p).$$

Definition 4.40 [p. 106] · (Abstraction operator) *The abstraction operator has the following signature:*

$$\tau\_ \quad : \quad \mathcal{P}(Action) \times P \to P.$$

The deduction rules for the abstraction operator are listed in Table 4.10.

Lemma 4.41 [p. 107] · *Let A and A' be sets of actions and let p be a process, then*

$$\tau_A(\tau_{A'}(p)) \leftrightarrows \tau_{A \cup A'}(p).$$

Definition 4.42 [p. 109] · (Stratification) *A mapping S from positive formulas to natural numbers is a* stratification *for the SOS of χ_σ if for every deduction rule $\frac{H}{c}$ and every closed substitution θ,*

- *for $h \in H$ of the forms 1–4 of Definition 4.3 (the positive hypotheses), $S(\theta(h)) \leq S(\theta(c))$; and*
- *for $h \in H$ of the forms 5–7 of Definition 4.3 (the negative hypotheses):*
  form 5: if $h = \neg\exists p' \in C(P), \sigma', a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, then for all closed terms p': $S(\langle\, \theta(p), \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle) < S(\theta(c))$; and
  form 6: if $h = \neg\exists p' \in C(P), \sigma', a : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle$, then for all closed terms p': $S(\langle\, \theta(p), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle) < S(\theta(c))$; and
  form 7: if $h = \neg\langle\, p, \sigma \,\rangle\!\downarrow$, $S(\langle\, \theta(p), \sigma \,\rangle\!\downarrow) < S(\theta(c))$; respectively.

A TSS with a stratification is stratifiable.

Definition 4.43 [p. 110] · *Let p and p' be processes. The function $ops : P \to N$ is defined recursively by*

- $ops(p) = 0$, *if* $p \in \{\delta, \varepsilon, \mathsf{skip}, x := e, m\,!\,e, m\,?\,e, \Delta e\}$,
- $ops(e :\rightarrow p) = 1 + ops(p)$,
- $ops(p \between p') = 1 + ops(p) + ops(p')$,
- $ops(p \;;\; p') = 1 + ops(p) + ops(p')$,
- $ops(p^*) = 1 + ops(p)$,
- $ops(p \between p') = 1 + ops(p) + ops(p')$,

- $ops([\![\, s \mid p \,]\!]) = 1 + ops(p)$,
- $ops(\partial_A(p)) = 1 + ops(p)$,
- $ops(\pi(p) = 1 + ops(p)$,
- $ops(\tau_A(p)) = 1 + ops(p)$.

Definition 4.44 [p. 110] · *Let p and p' be closed process terms, σ and σ' be stacks, a an action, and d a positive real number. The function S from positive formulas to natural numbers is defined by*

1. $S(\text{TRUE}(e)) = 0$,
2. $S(\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle) = ops(p)$,
3. $S(\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle) = ops(p)$,
4. $S(\langle\, p, \sigma \,\rangle\!\downarrow) = ops(p)$.

Lemma 4.45 [p. 110] · *The TSS of χ_σ is stratifiable.*

Lemma 4.46 [p. 113] · *Let p and p_d be processes; σ and σ_d be stacks; and d and d' be positive real numbers such that $d' < d$. Then*

$$\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle \Rightarrow \exists p_{d'}, \sigma_{d'} : \langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle.$$

Lemma 4.47 [p. 113] · (Time factorisation) *Let e and e' be expressions, such that $e \geq 0$ and $e' \geq 0$, and let p be a process, then*

$$\Delta e \;;\; p \,[\![\, \Delta e + e' \leftrightarrows \Delta e \;;\; (p \,[\![\, \Delta e').$$

Lemma 4.48 [p. 116] · (Time determinism) *Let p, p', and p'' be processes; σ, σ', and σ'' be stacks; and d be a positive real number. Then*

$$(\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle \wedge \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p'', \sigma'' \,\rangle) \Rightarrow (p' \equiv p'' \wedge \sigma' \equiv \sigma'').$$

Lemma 4.49 [p. 117] · *Let p and p' be processes; σ and σ' be stacks such that $m \in dom(\sigma)$, then*

$$\begin{aligned}
\langle\, p, \sigma \,\rangle \xrightarrow{\tau} \langle\, p', \sigma' \,\rangle &\Rightarrow \sigma' = \sigma, \\
\langle\, p, \sigma \,\rangle \xrightarrow{aa(x,c)} \langle\, p', \sigma' \,\rangle &\Rightarrow \sigma' = \sigma \vee \sigma' = \sigma[c/x], \\
\langle\, p, \sigma \,\rangle \xrightarrow{sa(m,c)} \langle\, p', \sigma' \,\rangle &\Rightarrow \sigma' = \sigma \vee \sigma' = \sigma[c/m], \\
\langle\, p, \sigma \,\rangle \xrightarrow{ra(m,x)} \langle\, p', \sigma' \,\rangle &\Rightarrow \sigma' = \sigma \vee \sigma' = \sigma[\sigma(m)/x], \\
\langle\, p, \sigma \,\rangle \xrightarrow{ca(m,x,c)} \langle\, p', \sigma' \,\rangle &\Rightarrow \sigma' = \sigma \vee \sigma' = \sigma[c/m][c/x], \\
\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle &\Rightarrow \sigma' = \sigma.
\end{aligned}$$

Lemma 4.50 [p. 118] · *Let p and p' be processes, σ_0, σ_0', σ_1, and σ_1' be stacks, a be an action, and d be a positive real number, then*

$$\begin{gathered}
(\langle\, p, \sigma_0 \,\rangle \xrightarrow{a} \langle\, p', \sigma_0' \,\rangle \wedge \langle\, p, \sigma_1 \,\rangle \xrightarrow{a} \langle\, p', \sigma_1' \,\rangle) \vee \\
(\langle\, p, \sigma_0 \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma_0' \,\rangle \wedge \langle\, p, \sigma_1 \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma_1' \,\rangle) \\
\Rightarrow \\
(\sigma_0' = \sigma_0 \wedge \sigma_1' = \sigma_1) \vee (\exists c, i : \sigma_0' = \sigma_0[c/i] \wedge \sigma_1' = \sigma_1[c/i]) \vee \\
(\exists c, c', i, i' : \sigma_0' = \sigma_0[c/i][c'/i'] \wedge \sigma_1' = \sigma_1[c/i][c'/i']).
\end{gathered}$$

Lemma 4.51 [p. 119] · *Let p and p' be processes, σ and σ' be stacks, a an action and d a positive real number, then*

$$\begin{aligned}
\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle &\Rightarrow dom(\sigma) = dom(\sigma'), \\
\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle &\Rightarrow dom(\sigma) = dom(\sigma').
\end{aligned}$$

Lemma 4.52 [p. 120] · *Let p and p' be processes and let σ and σ' be stacks such that $\sigma \doteq \sigma'$. Then we have*

$$\langle\, p, \sigma \,\rangle\!\downarrow \Leftrightarrow \langle\, p, \sigma' \,\rangle\!\downarrow.$$

Furthermore, let a be an action and d a positive real number, then

$$\begin{aligned}
\exists \sigma : \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma \,\rangle &\Leftrightarrow \exists \sigma' : \langle\, p, \sigma' \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle, \\
\exists \sigma : \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma \,\rangle &\Leftrightarrow \exists \sigma' : \langle\, p, \sigma' \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle.
\end{aligned}$$

Lemma 4.53 [p. 120] · (Time confluence) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p_d, \sigma_d \,\rangle \wedge \langle\, p, \sigma \,\rangle \overset{d'}{\longmapsto} \langle\, p_{d'}, \sigma_{d'} \,\rangle$, then $\langle\, q, \sigma' \,\rangle \overset{d-d'}{\longmapsto} \langle\, r, \sigma'' \,\rangle$.*

Lemma 4.54 [p. 120] · (Preservation of terminations) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle \wedge \langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, then $\langle p_{d'}, \sigma_{d'} \rangle\downarrow \Rightarrow \langle p, \sigma \rangle\downarrow \wedge \langle p_d, \sigma_d \rangle\downarrow$.*

Lemma 4.55 [p. 121] · (Preservation of action transitions) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle \wedge \langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, then $\forall p_{d',a}, \sigma_{d',a}, a : \langle p_{d'}, \sigma_{d'} \rangle \xrightarrow{a} \langle p_{d',a}, \sigma_{d',a} \rangle \Rightarrow \exists p_a, \sigma_a, p_{d,a}, \sigma_{d,a} : \langle p, \sigma \rangle \xrightarrow{a} \langle p_a, \sigma_a \rangle \wedge \langle p_d, \sigma_d \rangle \xrightarrow{a} \langle p_{d,a}, \sigma_{d,a} \rangle$.*

Lemma 4.56 [p. 121] · (Undelayability of terminations) *Let p, p_d, and $p_{d'}$ be processes; σ, σ_d, and $\sigma_{d'}$ be stacks; and d and d' be positive real numbers such that $d' < d$. If $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p_d, \sigma_d \rangle \wedge \langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle p_{d'}, \sigma_{d'} \rangle$, then $\langle p_{d'}, \sigma_{d'} \rangle \not\downarrow$.*

Definition 7.1 [p. 160] · (SOS computer) *The SOS computer function $sc : P \to \mathcal{P}(P \times Stack) \cup \mathcal{P}(P \times Stack \times (Action \cup R_{>0}) \times P \times Stack)$ is defined by*

$$\begin{aligned} sc(p) = {} & \{(p, \sigma) \mid \langle p, \sigma \rangle\downarrow, \sigma \in Stack\} \\ & \cup \{(p, \sigma, a, p'\sigma') \mid \langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle, \sigma, \sigma' \in Stack, p' \in P, a \in Action\} \\ & \cup \{(p, \sigma, d, p'\sigma') \mid \langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p', \sigma' \rangle, \sigma, \sigma' \in Stack, p' \in P, d \in R_{>0}\}. \end{aligned}$$

Lemma 7.2 [p. 161] · *Let p and p' be processes, a be an action, and d be a positive real number, then*

$$\begin{aligned} & \langle p, \lambda_\sigma \rangle\downarrow \Rightarrow \forall \sigma : \langle p, \sigma \rangle\downarrow, \\ & \langle p, \lambda_\sigma \rangle \xrightarrow{a} \langle p', \lambda_\sigma \rangle \Rightarrow \forall \sigma : \exists \sigma' : \langle p, \sigma \rangle \xrightarrow{a} \langle p, \sigma' \rangle, \\ & \langle p, \lambda_\sigma \rangle \overset{d}{\longmapsto} \langle p', \lambda_\sigma \rangle \Rightarrow \forall \sigma : \exists \sigma' : \langle p, \sigma \rangle \overset{d}{\longmapsto} \langle p, \sigma' \rangle. \end{aligned}$$

Lemma 7.3 [p. 162] · *Let $p \in P$ be a process and $\sigma \in Stack$ be a stack. Then $|\{(p, \sigma) \in sc(p, \sigma) \mid \langle p, \sigma \rangle\downarrow\}| \leq 1$.*

Lemma 7.4 [p. 162] · (Finite number of actions) *Let $p \in P$ be a process and $\sigma \in Stack$ be a stack, then $|Action(p, \sigma)| \in N$.*

Definition 7.5 [p. 165] · (Action computation) *The function $ac : P \times Stack \to \mathcal{P}(P \times Stack \times Action \times P \times Stack)$ is defined in Table 7.1.*

Lemma 7.6 [p. 165] · (Correctness of ac) *Let $p, p' \in P$ be processes, let $a \in Action$ be an action, and let $\sigma, \sigma' \in Stack$ be stacks, then*

$$(p, \sigma, a, p', \sigma') \in ac(p, \sigma) \Leftrightarrow \langle p, \sigma \rangle \xrightarrow{a} \langle p', \sigma' \rangle.$$

Lemma 7.7 [p. 170] · *Let p be a maximal progress process: $p \equiv \pi(p')$ for some process p'. Suppose $\langle p, \sigma \rangle \overset{d}{\longmapsto} \langle r, \sigma'' \rangle$ and $\langle p, \sigma \rangle \overset{d'}{\longmapsto} \langle q, \sigma' \rangle$ and $d' < d$. Then $\langle q, \sigma' \rangle \overset{d-d'}{\longmapsto} \langle r, \sigma'' \rangle$ and $\langle q, \sigma' \rangle \not\downarrow$ and $\langle q, \sigma' \rangle \not\mapsto$.*

Definition 7.8 [p. 170] · (Unique delay value) *Let $d_0 \in R_{>0}$ be an arbitrary positive real number. The function $D_{d_0} : P \times Stack \to R_{\geq 0}$ is defined in Table 7.2.*

Lemma 7.9 [p. 170] · (Valid unique delay value) *Let $p \in P$ be a process, $\sigma \in Stack$ be a stack, and $d_0 \in R_{>0}$ be a positive real number. Then, if $D_{d_0}(p, \sigma) = 0$ then $\langle p, \sigma \rangle \not\mapsto$, and if $D_{d_0}(p, \sigma) > 0$ then $\exists p' \in P, \sigma' \in Stack : \langle p, \sigma \rangle \overset{D_{d_0}(p,\sigma)}{\longmapsto} \langle p', \sigma' \rangle$.*

Definition 7.10 [p. 177] · (Delay computation) *Let $d_0 \in R_{>0}$ be a positive real number. The function $dc_{d_0} : P \times Stack \to (P \times Stack \times R_{>0} \times P \times Stack)$ is defined by*

$$\begin{aligned} dc_{d_0}(p, \sigma) = \{ & (p, \sigma, D_{d_0}(p, \sigma), p', \sigma') \\ & \mid D_{d_0}(p, \sigma) > 0 \wedge (p, \sigma, D_{d_0}(p, \sigma), p', \sigma') \in sc(p, \sigma) \\ & \}. \end{aligned}$$

Lemma 7.11 [p. 177] · *Let $p \in P$ be a process and $\sigma \in Stack$ be a stack, then $|dc_{d_0}(p, \sigma)| \leq 1$.*

Definition 7.12 [p. 177] · (Finite SOS computer) *Let d_0 be a positive real number. The finite SOS computer function $sc'_{d_0} : P \to \mathcal{P}(P \times Stack) \cup \mathcal{P}(P \times Stack \times (Action \cup R_{>0}) \times P \times Stack)$ is defined by*

$$sc'_{d_0}(p) = \{(p, \lambda_\sigma) \mid (p, \lambda_\sigma) \in sc(p)\} \cup ac(p, \lambda_\sigma) \cup dc_{d_0}(p, \lambda_\sigma).$$

Definition A.1 [p. 249] · (Valuation) *Let v be a valuation, i be an identifier, and c be a value. A valuation is a mapping from an identifier to a value with syntax $v ::= i \mapsto c$.*

Definition A.2 [p. 249] · (State) *The empty state is denoted by λ_s. Further, let v be a valuation. A state s is a list of valuations with syntax*

$$\begin{aligned} s ::= & \lambda_s \\ & \mid v : s. \end{aligned}$$

By definition, states have unique identifiers. That is, each identifier occurs at most once in a state. In addition, all possible states are contained in the set State.

Definition A.3 [p. 250] · *Let s be a state, i be an identifier, and c be a value. The function dom, which returns the domain of a state, is defined by*

$$\begin{aligned} dom(\lambda_s) &= \emptyset, \\ dom(i \mapsto c : s) &= \{i\} \cup dom(s). \end{aligned}$$

Definition A.4 [p. 250] · *Let s be a state, i and i' be identifiers, and c and c' be values. Substitution on states is defined by*

$$\begin{aligned} \lambda_s[c/i] &= \lambda_s, \\ (i \mapsto c : s)[c'/i] &= i \mapsto c' : s, \\ (i \mapsto c : s)[c'/i'] &= i \mapsto c : s[c'/i'] \quad \text{if } i \neq i'. \end{aligned}$$

Definition A.5 [p. 250] · *Let s be a state, i and i' be identifiers, and c be a value. Looking up identifiers in states is defined by*

$$\begin{aligned} \lambda_s(i) &= i, \\ (i \mapsto c : s)(i) &= c, \\ (i \mapsto c : s)(i') &= s(i') \quad \text{if } i \neq i'. \end{aligned}$$

Definition A.6 [p. 250] · *Let s and s' be states, and i be an identifier. Equivalence on states is defined by $s = s'$ if $\forall i : s(i) = s'(i)$.*

Definition A.7 [p. 251] · *Let s and s' be states, i be an identifier, and c be a value. The set function on states is defined by*

$$\begin{aligned} set(s, \lambda_s) &= s, \\ set(s, i \mapsto c : s') &= set(s[c/i], s') \qquad \text{if } i \in dom(s), \\ set(s, i \mapsto c : s') &= set(i \mapsto c : s, s') \quad \text{if } i \notin dom(s). \end{aligned}$$

Definition A.8 [p. 251] · (State stack) *The empty state stack is denoted by λ_σ. Let σ be a state stack and s be a state. A state stack has syntax*

$$\begin{aligned} \sigma ::= {} & \lambda_\sigma \\ \mid {} & s :: \sigma. \end{aligned}$$

All possible state stacks are contained in the set Stack.

Definition A.9 [p. 251] · *Let σ be a stack, and s be a state. The function dom is defined by*

$$\begin{aligned} dom(\lambda_\sigma) &= \emptyset, \\ dom(s :: \sigma) &= dom(s) \cup dom(\sigma). \end{aligned}$$

Definition A.10 [p. 252] · *Let σ be a stack, s be a state, i be an identifier, and c be a value. Substitution on stacks is defined by*

$$\begin{aligned} \lambda_\sigma[c/i] &= \lambda_\sigma, \\ (s :: \sigma)[c/i] &= s[c/i] :: \sigma \quad \textit{if } i \in \textit{dom}(s), \\ (s :: \sigma)[c/i] &= s :: \sigma[c/i] \quad \textit{if } i \notin \textit{dom}(s). \end{aligned}$$

Definition A.11 [p. 252] · *Let σ be a stack, s be a state, and i be an identifier. Looking up identifiers in stacks is defined by*

$$\begin{aligned} \lambda_\sigma(i) &= i, \\ (s :: \sigma)(i) &= s(i) \quad \textit{if } i \in \textit{dom}(s), \\ (s :: \sigma)(i) &= \sigma(i) \quad \textit{if } i \notin \textit{dom}(s). \end{aligned}$$

Definition A.12 [p. 252] · *Let σ and σ' be stacks, and i be an identifier. Equivalence on stacks is defined by*

$$\begin{aligned} \lambda_\sigma &= \lambda_\sigma, \\ s :: \sigma &= s' :: \sigma' \quad \textit{if } s = s' \wedge \sigma = \sigma'. \end{aligned}$$

Definition A.13 [p. 252] · *Let σ and σ' be stacks, and i be an identifier. Observational equivalence on stacks is defined by $\sigma \stackrel{\circ}{=} \sigma'$ if $\forall i : \sigma(i) = \sigma'(i)$.*

Lemma A.14 [p. 252] · *Let s be a state, i and i' be identifiers, and c and c' be values. Then $i \mapsto c : i' \mapsto c' : s = i' \mapsto c' : i \mapsto c : s$.*

Lemma A.15 [p. 253] · *Let s and s' be states, i be an identifier, and c be a value. Then*

$$s = i \mapsto c : s' \Rightarrow \textit{dom}(s') \subset \textit{dom}(s).$$

Lemma A.16 [p. 254] · *Let s be a state, i be an identifier, and c be a value. In that case, $\textit{dom}(s[c/i]) = \textit{dom}(s)$.*

Lemma A.17 [p. 255] · *Let s be a state, i be an identifier, and c be a value. Then*

$$\begin{aligned} s[s(i)/i] &= s \quad \textit{if } i \in \textit{dom}(s), \\ s[c/i] &= s \quad \textit{if } i \notin \textit{dom}(s). \end{aligned}$$

Lemma A.18 [p. 256] · *Let s be a state, i and i' be identifiers, and c and c' be values. Then*

$$\begin{aligned} s[c/i][c'/i'] &= s[c'/i'] && \text{if } i = i', \\ s[c/i][c'/i'] &= s[c'/i'][c/i] && \text{if } i \neq i'. \end{aligned}$$

Lemma A.19 [p. 258] · *Let s be a state, i and i' be identifiers, and c be a value. Then*

$$\begin{aligned} s[c/i](i) &= c && \text{if } i \in dom(s), \\ s[c/i](i) &= i && \text{if } i \notin dom(s), \\ s[c/i](i') &= s(i') && \text{if } i \neq i'. \end{aligned}$$

Lemma A.20 [p. 261] · *Let s and s' be states, c be a value, and i be an identifier. Then $set(i \mapsto c : s, s') = i \mapsto c : set(s, s')$ if $i \notin dom(s')$.*

Lemma A.21 [p. 262] · *Let s and s' be states. Then $dom(set(s, s')) = dom(s) \cup dom(s')$.*

Lemma A.22 [p. 264] · *Let s and s' be states, c be a value, and i be an identifier. Then*

$$\begin{aligned} set(s, s')[c/i] &= set(s, s'[c/i]) \text{ if } i \in dom(s'), \\ set(s, s')[c/i] &= set(s[c/i], s') \text{ if } i \notin dom(s'). \end{aligned}$$

Lemma A.23 [p. 265] · *Let s and s' be states, and i be an identifier. Then*

$$\begin{aligned} set(s, s')(i) &= s'(i) && \text{if } i \in dom(s'), \\ set(s, s')(i) &= s(i) && \text{if } i \notin dom(s'). \end{aligned}$$

Lemma A.24 [p. 268] · *Let s be a state and σ and σ' be stacks. Then*

$$\sigma = s :: \sigma' \Rightarrow dom(s) \subseteq dom(\sigma) \wedge dom(\sigma') \subseteq dom(\sigma).$$

Lemma A.25 [p. 269] · *Let σ be a stack, i be an identifier, c be a value, I be a set of identifiers, and c_i be a value for all $i \in I$. Then $dom(\sigma[c/i]) = dom(\sigma)$.*

Lemma A.26 [p. 270] · *Let σ be a stack, i be an identifier, and c be a value. Then*

$$\begin{aligned} \sigma[\sigma(i)/i] &= \sigma && \text{if } i \in dom(\sigma), \\ \sigma[c/i] &= \sigma && \text{if } i \notin dom(\sigma). \end{aligned}$$

Definition C.1 [p. 279] · (Kind name) *A kind name is an identifier K.*

Definition C.2 [p. 280] · (Sort name) *A sort name is a qualified identifier $s.K$, where s is an identifier and K is a kind name.*

Definition C.3 [p. 280] · (Function name) *A function name consists of an identifier f and a function type $K_1\ K_2\ \ldots\ K_n \to K$ $(n \geq 0)$, where $K_1, K_2, \ldots, K_n, K$ are kind names. It is written like: $f : K_1\ K_2\ \ldots\ K_n \to K$. Kind names $K_1, \ldots, K_n$ are called the* input kinds *or* input kind names *and K is called the* output kind *or* output kind name *of the function.*

Definition C.4 [p. 280] · (Signature) *A signature Ω is a triple $(\mathcal{K}, \mathcal{S}, \mathcal{F})$ where $\mathcal{K}$ is a set of kinds, $\mathcal{S}$ is a set of sort names, and $\mathcal{F}$ is a set of function names. Furthermore, for all $s.K \in \mathcal{S}$, we require that $K \in \mathcal{K}$; and for all function names $f : K_1\ K_2\ \ldots\ K_n \to K \in \mathcal{F}$, we require that $K_1, K_2, \ldots, K_n, K \in \mathcal{K}$.*

Definition C.5 [p. 280] · (Logical variable) *A logical variable is a qualified identifier $x.K$, where x is an identifier and K is a kind name.*

Definition C.6 [p. 281] · (Terms) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and X a set of variables of kinds in $\mathcal{K}$. The set of terms over Ω parameterised by X is denoted by $Term_\Omega(X)$. The set $Term_\Omega(X)$ is recursively defined by the following rules.*

1. *Every variable $x.K \in X$ is a term of kind K: $x.K \in Term_\Omega(X)$.*
2. *Every nullary function name $f :\to K \in \mathcal{F}$ is a term of kind K: $f :\to K \in Term_\Omega(X)$.*
3. *If $t_1, t_2, \ldots, t_n \in Term_\Omega(X)$ are terms of kinds $K_1\ K_2\ \ldots\ K_n$ (for $n > 0$) respectively, and $f : K_1\ K_2\ \ldots\ K_n \to K \in \mathcal{F}$ is a function name, then $f(t_1, t_2, \ldots, t_n) \in Term_\Omega(X)$ is a term of kind K.*

Definition C.7 [p. 281] · (Substitution) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and X a set of variables of kinds in $\mathcal{K}$. A* substitution *is a function $\theta : X \to Term_\Omega(X)$ such that for all $x.K \in X$ the term $\theta(x.K)$ is of kind K.*

Definition C.8 [p. 282] · (Extending a substitution) *Let $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ be a signature and X a set of variables of kinds in Ω. Let $x \in X$ be a variable, $c :\to K$ a nullary function name in Ω; $t_1, \ldots, t_n$ be terms in $Term_\Omega(X)$; and f an n-ary function*

name such that $f(t_1, \ldots, t_n) \in Term_\Omega(X)$ *is a term. The* extended substitution $\overline{\theta}$ *of a substitution* $\theta : X \to Term_\Omega(X)$ *is a function* $\overline{\theta} : Term_\Omega(X) \to Term_\Omega(X)$ *defined by* $\overline{\theta}(x) = \theta(x)$, $\overline{\theta}(c) = c$, *and* $\overline{\theta}(f(t_1, \ldots, t_n)) = f(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n))$.

Definition C.9 [p. 282] · (Formulas) *Let* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ *be a signature and* X *a set of variables of kinds in* $\mathcal{K}$. Atomic formulas *of membership equational logic are* equations *and* membership assertions. *We use the character a (possibly indexed) to range over equations or membership assertions.*

Let $t_1, t_2 \in Term_\Omega(X)$ *be terms of the same kind.*

1. *An* equation *has the form:* $t_1 = t_2$.

Let $t \in Term_\Omega(X)$ *be a term of kind* K *and* $s.K \in \mathcal{S}$ *a sort name.*

2. *A* membership assertion *has the form:* $t : s.K$.

Sentences *(or* compound formulas*) of membership equational logic are* conditional equations *and* conditional membership assertions.

Let $t_1, t_2 \in Term_\Omega(X)$ *be terms and* $a_1, \ldots, a_n$ $(n \geq 0)$ *be atomic formulas.*

3. *A* conditional equation *has the form:* $t_1 = t_2 \Leftarrow a_1, \ldots, a_n$.

Let $t \in Term_\Omega(X)$ *be a term of kind* K, $s.K \in \mathcal{S}$ *a sort name, and* $a_1, \ldots, a_n$ $(n \geq 0)$ *equations or membership assertions.*

4. *A* conditional membership assertion *has the form:* $t : s.K \Leftarrow a_1, \ldots, a_n$.

Definition C.10 [p. 283] · (Membership equational specification) *A membership equational specification is a tuple* (Ω, E, M) *where* Ω *is a signature,* E *is a set of equations and conditional equations over* Ω, *and* M *is a set of membership assertions and conditional membership assertions over* Ω.

Definition C.11 [p. 283] · (Derivation in MEL) *Let* $T = (\Omega, E, M)$ *be a membership equational specification with* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$, X *a set of variables of kinds in* $\mathcal{K}$; $f : K_1 \ldots K_n \to K$ *an* n*-ary function name,* t, t', t'', $t_1 \ldots t_n$, $r \in Term_\Omega(X)$; $a_1, \ldots, a_n$ *equations or membership assertions;* K, $K' \in \mathcal{K}$; $s.K$, $s.K' \in \mathcal{S}$; *and* θ *a substitution. The MEL derivation relation* $\vdash'$ *(for* $0 < i \leq n$*) is defined by*

1. *equational axiom: if* $t = t' \in E$ *then* $T \vdash \theta(t) = \theta(t')$,
2. *membership axiom: if* $r : s.K \in M$ *then* $T \vdash \theta(r) : s.K$,
3. *conditional equational axiom: if* $t = t' \Leftarrow a_1, \ldots, a_n \in E$ *and* $T \vdash \theta(a_i)$, *then* $T \vdash \theta(t) = \theta(t')$,
4. *conditional membership axiom: if* $r : s.K \Leftarrow a_1, \ldots, a_n \in M$ *and* $T \vdash \theta(a_i)$, *then* $T \vdash \theta(r) : s.K$,
5. *subject reduction: if* $T \vdash t' : s.K$ *and* $T \vdash t = t'$ *then* $T \vdash t : s.K$,
6. *reflexivity:* $T \vdash t = t$,
7. *symmetry: if* $T \vdash t = t'$ *then* $T \vdash t' = t$,
8. *transitivity: if* $T \vdash t = t'$ *and* $T \vdash t' = t''$, *then* $T \vdash t = t''$,
9. *congruence: if* $T \vdash t_i = t'_i$, *then* $T \vdash f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$.

Definition C.12 [p. 284] · (Ω Algebra) *An* Ω *Algebra* A *for a signature* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$, *is a triple* $(\mathcal{K}^A, \mathcal{S}^A, \mathcal{F}^A)$ *of sets, where for each* $K \in \mathcal{K}$ *there is a set* $K^A \in \mathcal{K}^A$; *for each* $s.K \in \mathcal{S}$ *there is a set* $(s.K)^A \in \mathcal{S}^A$ *such that* $(s.K)^A \subseteq K^A$; *and for each* $f : K_1\, K_2\, \ldots\, K_n \to K \in \mathcal{F}$ *there is a function* $f^A : K_1^A \times \ldots \times K_n^A \to K^A \in \mathcal{F}^A$.

Definition C.13 [p. 284] · (Valuation) *Let* X *be a set of variables and* A *an* Ω *algebra for signature* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$. *A valuation is a function* $v : X \to \bigcup_{K \in \mathcal{K}} K^A$, *such that* $v(x.K) \in K^A$ *for all* $x \in X$.

Definition C.14 [p. 285] · (Interpretation) *Let* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ *be a signature,* $x \in X$ *be a variable,* $c :\to K \in \mathcal{F}$ *a nullary function name, and* $f : K_1 \ldots K_n \to K' \in \mathcal{F}$ *a* n*-ary function name. The interpretation of a term* $t \in Term_\Omega(X)$ *in an* Ω *algebra* A *is denoted by* $[\![t]\!]_v^A$, *where* v *is a valuation. It is defined by*

1. $[\![x]\!]_v^A = v(x)$,
2. $[\![c]\!]_v^A = c^A$,
3. $[\![f(t_1, \ldots, t_n)]\!]_v^A = f^A([\![t_1]\!]_v^A, \ldots, [\![t_n]\!]_v^A)$.

Definition C.15 [p. 285] · (Membership equational model) *Let* $T = (\Omega, E, M)$ *be an equational specification with* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$. *An* Ω *algebra* A *is a* model *of* T *if it satisfies the following conditions:*

1. *if* $t_1 = t_2 \in E$, *then* $[\![t_1]\!]_v = [\![t_2]\!]_v$ *should hold for all valuations* v,
2. *if* $t : s.K \in M$, *then* $[\![t]\!]_v \in (s.K)^A$ *should hold for all valuations* v,
3. *if* $t_1 = t_2 \Leftarrow a_1, \ldots a_n \in E$ *and* $[\![a_1]\!]_v$, ..., $[\![a_n]\!]_v$ *hold, then* $[\![t_1]\!]_v = [\![t_2]\!]_v$ *should hold for all valuations* v,
4. *if* $t : s.K \Leftarrow a_1, \ldots a_n \in M$ *and* $[\![a_1]\!]_v$, ..., $[\![a_n]\!]_v$ *hold, then* $[\![t]\!]_v \in (s.K)^A$ *should hold for all valuations* v.

Definition C.16 [p. 286] · (Homomorphism) *Let* A *and* A' *be* Ω *algebras, where* $\Omega = (\mathcal{K}, \mathcal{S}, \mathcal{F})$ *and let* $c_1, \ldots, c_n \in \bigcup_{K \in \mathcal{K}} K^A$. *A function* $h : \bigcup_{K \in \mathcal{K}} K^A \to \bigcup_{K \in \mathcal{K}} K^{A'}$ *is a homomorphism from* A *to* A' *if it satisfies*

$$h(f^A(c_1, \ldots, c_n)) = f^{A'}(h(c_1), \ldots, f(c_n)).$$

Definition C.17 [p. 286] · (Initiality) *An* Ω *algebra* A *is* initial *if for any* Ω *algebra* A' *there is a unique homomorphism* $h : A \to A'$.

List of deduction rules · E

[p. 69] $$\frac{}{\langle\, \varepsilon, \sigma \,\rangle\!\downarrow}\ 1$$

[p. 69] $$\frac{\sigma(e) = 0}{\langle\, \Delta e, \sigma \,\rangle\!\downarrow}\ 2$$

[p. 69] $$\frac{}{\langle\, \mathsf{skip}, \sigma \,\rangle \xrightarrow{\tau} \langle\, \varepsilon, \sigma \,\rangle}\ 3$$

[p. 69] $$\frac{\sigma(e) = c}{\langle\, x := e, \sigma \,\rangle \xrightarrow{aa(x,c)} \langle\, \varepsilon, \sigma[c/x] \,\rangle}\ 4$$

[p. 69] $$\frac{\sigma(e) = c}{\langle\, m\,!\,e, \sigma \,\rangle \xrightarrow{sa(m,c)} \langle\, \varepsilon, \sigma[c/m] \,\rangle}\ 5$$

[p. 69] $$\frac{\sigma(m) = c}{\langle\, m\,?\,x, \sigma \,\rangle \xrightarrow{ra(m,x)} \langle\, \varepsilon, \sigma[c/x] \,\rangle}\ 6$$

[p. 69] $$\frac{}{\langle\, m\,!\,e, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, m\,!\,e, \sigma \,\rangle}\ 7$$

[p. 69] $$\frac{}{\langle\, m\,?\,x, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, m\,?\,x, \sigma \,\rangle}\ 8$$

[p. 69] $$\frac{d \leq \sigma(e)}{\langle\, \Delta e, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \Delta e - d, \sigma \,\rangle}\ 9$$

[p. 70] $$\frac{\sigma(e) = \mathit{true},\ \langle\, p, \sigma \,\rangle\!\downarrow}{\langle\, e :\!\rightarrow p, \sigma \,\rangle\!\downarrow}\ 10$$

[p. 70] $$\frac{\sigma(e) = \mathit{true},\ \langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, e :\!\rightarrow p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}\ 11$$

[p. 70] $$\frac{\sigma(e) = \mathit{true},\ \langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}{\langle\, e :\!\rightarrow p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}\ 12$$

[p. 73] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow}{\langle\, p \parallel q, \sigma \,\rangle\!\downarrow,\ \langle\, q \parallel p, \sigma \,\rangle\!\downarrow}\ 13$$

[p. 73] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, p \parallel q, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle,\ \langle\, q \parallel p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}\ 14$$

[p. 73] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, q, \sigma \,\rangle \not\mapsto}{\langle\, p \parallel q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, q \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}\ 15$$

[p. 73] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle}{\langle\, p \parallel q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p' \parallel q', \sigma' \,\rangle}\ 16$$

[p. 77] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow,\ \langle\, q, \sigma \,\rangle\!\downarrow}{\langle\, p\ ;\ q, \sigma \,\rangle\!\downarrow}\ 17$$

[p. 77] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, p \;;\; q, \sigma \,\rangle \xrightarrow{a} \langle\, p' \;;\; q, \sigma' \,\rangle}\ 18$$

[p. 77] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow,\ \langle\, q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle}{\langle\, p \;;\; q, \sigma \,\rangle \xrightarrow{a} \langle\, q', \sigma' \,\rangle}\ 19$$

[p. 77] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, p, \sigma \,\rangle\not\!\downarrow}{\langle\, p \;;\; q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p' \;;\; q, \sigma' \,\rangle}\ 20$$

[p. 77] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, q, \sigma \,\rangle\not\mapsto}{\langle\, p \;;\; q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p' \;;\; q, \sigma' \,\rangle}\ 21$$

[p. 77] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow,\ \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle,\ \langle\, p, \sigma \,\rangle\not\mapsto}{\langle\, p \;;\; q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle}\ 22$$

[p. 77] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, p, \sigma \,\rangle\!\downarrow,\ \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle}{\langle\, p \;;\; q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, (p' \;;\; q) \,[\!]\, q', \sigma' \,\rangle}\ 23$$

[p. 89] $$\frac{}{\langle\, p^*, \sigma \,\rangle\!\downarrow}\ 24$$

[p. 89] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, p^*, \sigma \,\rangle \xrightarrow{a} \langle\, p' \;;\; p^*, \sigma' \,\rangle}\ 25$$

[p. 89] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}{\langle\, p^*, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p' \;;\; p^*, \sigma' \,\rangle}\ 26$$

[p. 92] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow,\ \langle\, q, \sigma \,\rangle\!\downarrow}{\langle\, p \parallel q, \sigma \,\rangle\!\downarrow}\ 27$$

[p. 92] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, p \parallel q, \sigma \,\rangle \xrightarrow{a} \langle\, p' \parallel q, \sigma' \,\rangle,\ \langle\, q \parallel p, \sigma \,\rangle \xrightarrow{a} \langle\, q \parallel p', \sigma' \,\rangle}\ 28$$

[p. 92] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{sa(m,c)} \langle\, p', \sigma' \,\rangle,\ \langle\, q, \sigma' \,\rangle \xrightarrow{ra(m,x)} \langle\, q', \sigma'' \,\rangle}{\langle\, p \parallel q, \sigma \,\rangle \xrightarrow{ca(m,x,c)} \langle\, p' \parallel q', \sigma'' \,\rangle,\ \langle\, q \parallel p, \sigma \,\rangle \xrightarrow{ca(m,x,c)} \langle\, q' \parallel p', \sigma'' \,\rangle}\ 29$$

[p. 92] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, q, \sigma \,\rangle\!\downarrow,\ \langle\, q, \sigma \,\rangle \not\overset{d}{\longmapsto}}{\langle\, p \parallel q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, q \parallel p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}\ 30$$

[p. 92] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, q', \sigma' \,\rangle}{\langle\, p \parallel q, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p' \parallel q', \sigma' \,\rangle}\ 31$$

[p. 96] $$\frac{\langle\, p, s :: \sigma \,\rangle\!\downarrow}{\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle\!\downarrow}\ 32$$

[p. 96] $$\frac{\langle\, p, s :: \sigma \,\rangle \xrightarrow{a} \langle\, p', s' :: \sigma' \,\rangle}{\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle \xrightarrow{a} \langle\, [\![\, s' \mid p' \,]\!], \sigma' \,\rangle}\ 33$$

[p. 96] $$\frac{\langle\, p, s :: \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', s' :: \sigma' \,\rangle}{\langle\, [\![\, s \mid p \,]\!], \sigma \,\rangle \overset{d}{\longmapsto} \langle\, [\![\, s' \mid p' \,]\!], \sigma' \,\rangle}\ 34$$

[p. 103] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow}{\langle\, \partial_A(p), \sigma \,\rangle\!\downarrow}\ 35$$

[p. 103] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle,\ a \notin A}{\langle\, \partial_A(p), \sigma \,\rangle \xrightarrow{a} \langle\, \partial_A(p'), \sigma' \,\rangle}\ 36$$

[p. 103] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}{\langle\, \partial_A(p), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \partial_A(p'), \sigma' \,\rangle} \quad 37$$

[p. 105] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow}{\langle\, \pi(p), \sigma \,\rangle\!\downarrow} \quad 38$$

[p. 105] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle}{\langle\, \pi(p), \sigma \,\rangle \xrightarrow{a} \langle\, \pi(p'), \sigma' \,\rangle} \quad 39$$

[p. 105] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle,\ \langle\, p, \sigma \,\rangle \not\mapsto}{\langle\, \pi(p), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \pi(p'), \sigma' \,\rangle} \quad 40$$

[p. 107] $$\frac{\langle\, p, \sigma \,\rangle\!\downarrow}{\langle\, \tau_A(p), \sigma \,\rangle\!\downarrow} \quad 41$$

[p. 107] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle,\ a \notin A}{\langle\, \tau_A(p), \sigma \,\rangle \xrightarrow{a} \langle\, \tau_A(p'), \sigma' \,\rangle} \quad 42$$

[p. 107] $$\frac{\langle\, p, \sigma \,\rangle \xrightarrow{a} \langle\, p', \sigma' \,\rangle,\ a \in A}{\langle\, \tau_A(p), \sigma \,\rangle \xrightarrow{\tau} \langle\, \tau_A(p'), \sigma' \,\rangle} \quad 43$$

[p. 107] $$\frac{\langle\, p, \sigma \,\rangle \overset{d}{\longmapsto} \langle\, p', \sigma' \,\rangle}{\langle\, \tau_A(p), \sigma \,\rangle \overset{d}{\longmapsto} \langle\, \tau_A(p'), \sigma' \,\rangle} \quad 44$$

Curricula vitarum

V. Bos

Victor Bos was born on the 28th of November, 1970 in Apeldoorn, The Netherlands. In 1990, he finished the VWO at the Winschoter Scholengemeenschap in Winschoten. He studied Computer Science from 1990 until 1995 at the University of Groningen. From 1996–1998, he was a student of the postgraduate programme on Software Technology at the Stan Ackermans Institute at the Eindhoven University of Technology. In 1998, he started his Ph.D. project at the Formal Methods Group of the Faculty of Mathematics and Computer Science at the Eindhoven University of Technology. This is a combined project with the Systems Engineering Group of the Faculty of Mechanical Engineering at the same university.

J.J.T. Kleijn

Jeroen Kleijn was born on the 17th of March, 1973 in Goirle, The Netherlands. In 1991, he finished the *atheneum* at the Kruisheren College in Uden. He studied Mechanical Engineering at the Eindhoven University of Technology from 1991 until 1997 and graduated within the Systems Engineering group. After graduation, he started his Ph.D. project on formal specification and analysis of industrial systems at the same group. This is a combined project with the Formal Methods group of the Faculty of Mathematics and Computing Science at the same university.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

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, Univ. Leiden. 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. *Schedulere 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 Progam 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