Process Algebra and Structured Operational Semantics

Process Algebra and Structured Operational Semantics

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. P.W.M. de Meijer
in het openbaar the verdedigen in de Aula der Universiteit
(Oude Lutherse kerk, ingang Singel 411, hoek Spui),
op vrijdag 1 november 1991 te 15.00 uur

door
Jan Friso Groote
geboren te Doetinchem

Centrum voor Wiskunde en Informatica
1991

Promotor:	prof. dr. J.A. Bergstra
Co-promotor:	dr. J.C.M. Baeten
Faculteit:	Wiskunde en Informatica

The work in this thesis has been carried out at the CWI, Amsterdam in the context of ESPRIT project no. 432, an Integrated Formal Approach to Industrial Software Development (METEOR), RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS) and ESPRIT Basic Research Action no. 3006 (CONCUR).

Acknowledgements

It is hard to give proper credits to all who have contributed to this thesis. Below I have certainly not mentioned all to whom I am indebted.

I want to thank Jan Bergstra and Jos Baeten for being my promotor and co-promotor. Together with Jaco de Bakker they have created a very pleasant working atmosphere at the CWI where I had the feeling that I could exactly carry out the research that I like to do.

I also want to thank all my colleagues at the CWI, the University of Amsterdam and elsewhere. Especially, I am a lot indebted to Frits Vaandrager. By writing an article together, he led me from a state of inertia to a state in which I was able to write this thesis. More recently we wrote a second article which forms chapter 3 of this thesis. I am also very grateful to Roland Bol with whom I wrote the chapter 5 in this thesis, and Alban Ponse with whom the last chapter has been written. That chapter has been heavily influenced by Jan Bergstra. During several debates he managed to have us develop a concise language. I also want to thank Rob van Glabbeek who was always willing to point out flaws in my ideas. This often helped me to find the right track.

Many thanks also go to Krzysztof Apt, Frank de Boer, Jacob Brunekreef, Arie van Deursen, Nicolien Drost, Willem Jan Fokkink, Jan Heering, Paul Hendriks, Eiichi Horita, Jean Marie Jacquet, Jan Willem Klop, Steven Klusener, Henri Korver, Karst Koymans, Sjouke Mauw, Emma van der Meulen, Aart Middeldorp, Hans Mulder, Catuscia Palamidessi, Jan Rekers, Jan Rutten, Scott Smolka, Gert Veltink, Chris Verhoef, Emile Verschure, Jos Vrancken, Fer-Jan de Vries, Jeroen Warmerdam and Peter Weijland for interesting discussions, pointing out articles, spotting errors in my papers, helping to use software, stimulating to write my ideas down and for giving good advice.

Of course I should not forget the members of the SPECS consortium. I learned a lot from SPECS. I want to thank SPECS for the inspiration (chapter 6 and 7 are inspired by SPECS) and for showing me the management techniques necessary for such a project. My thanks especially go to Wiet Bouma, Jeroen Bruijning, Michel Dauphin, Jens Godskesen, Tanja de Groot, Bertrand Gruson, Jan Gustafsson, Günter Karjoth, Georg Karner, Martin Kooij, Max Michel, Cees Middelburg, Marc Phallipou, Simon Pickin, Sylvie Simon and Han Zuidweg.

Last, but not least, I thank my promotor and co-promotor, and Jaco de Bakker, Peter van Emde Boas, Matthew Hennessy, Paul Klint and Joachim Parrow for being member of the 'promotiecommissie'.

June 9, 2009

Contents

Contents

1

Introduction

This thesis proposes a number of solutions to various problems in process algebra and operational semantics. Below these problems and their historical contexts are sketched.

1.1 Process algebra

Process algebra studies concurrent communicating processes in an algebraic framework. Processes are modelled by structures, for instance projective limits [6], graphs or (especially in this thesis) transition systems. These structures are equipped with several operators, and so one gets an algebra in the sense of model theory.

1.1.1 Historical overview

Process algebra as we know it today has a considerable history. It originated from the wish to compose behaviour of processes into more complex behaviour through the use of operators. Many models in theoretical computer science, e.g. Turing machines, Petri nets or automata, are essentially not compositional in this sense. In 1969 Burstall and Landin [23] gave a plea to apply algebraic techniques in the study of programming languages. They provided the earliest examples of process algebras although these in no way resemble the algebras that are generally considered today.

The roots of process algebra as we know it now can be traced back to the work of Scott and Strachey [82, 86] (1971). They studied sequential programming languages and provided them with a mathematical semantics in terms of continuous functions. These functions describe how a program transforms initial input into final output. Continuous functions are not satisfactory to give a semantics to many features whose analogues are often found in concrete systems. In 1973 Milner signaled this problem and gave the following classes of programs that could not be dealt with [59, 60]:

1. non terminating programs with side effects,
2. non deterministic programs and
3. parallel programs.

As a basis to study these programs Milner proposed an algebra of processes with as operations a set of atomic input/output transformations, an alternative composition operator ($?$), a sequential composition operator ($*$) and a parallel composition operator ($\|$). A process is a function from input to output and a continuation of the process. This continuation is also a process in the sense that it can read more input and provide new output. This continuous reading and writing of processes describe its side effects. Non-deterministic behaviour, caused by the alternative and parallel composition operators is described using an oracle, which arbitrarily picks a possible future behaviour of the process. These processes strongly resemble Mealy machines. Somewhat earlier, in 1971, Bekič, who also wanted to model the behaviour of parallel programs, developed a process algebra too [10]. Bekič based his work on elementary state transformations. A process consists of a set of interleaved sequences of these.

Bekič extended his work slightly (for instance to CSP [42], see [11] opus 60 and 62). Milner developed his theory considerably (see e.g. [58, 61, 62]). In 1980 he published his famous book 'A Calculus of Communicating Systems' [63]. Generally this book is seen as the starting point for process algebra in its current form. In this book the main process constructors are an alternative composition operator ($+$), action prefix operators ($a : .$) for a number of atomic actions a, and a parallel operator ($|$). Processes are defined as transition systems or labeled automata modulo either strong congruence or observation congruence. Strong congruence says that two processes are the same if they can mutually simulate each other at any time. Observation congruence is defined equally, except that it takes internal or hidden actions into account. Park [72] adapted these definitions slightly to obtain the now more popular strong bisimulation and observation (or weak) bisimulation congruence. The resulting algebra is called CCS in conformity with the title of Milner's book.

CCS has had an enormous influence on the development of process based languages. For instance in 1978, before CCS existed, CSP (Communicating Sequential Processes) was presented as 'an ambitious attempt to find a single simple solution' for communication and synchronisation in parallel programming (see [42], quote from the introduction). The synchronous communication of CSP was meant to serve as a single and simple way to describe parallel programs, as an alternative to constructs such as semaphores [28], conditional critical regions [41], monitors and queues [21]. In 1985, influenced by Milner, Hoare viewed CSP as the 'simplest possible mathematical theory' that can be used as a specification, a programming and a verification language (see [43], quote can be found in chapter 7).

In 1982 Bergstra and Klop studied the question whether general equations over a process language, with a set of atomic actions, alternative composition,

sequential composition and a parallel operator, have solutions in a completed metrical space [12]. In order to solve this question, they introduce the so called axiomatic approach: a process algebra is any algebra satisfying a number of elementary equations. In 1984, continuing along this line, they proposed to determine a general algebraic concurrency theory with some fixed basic operators, analogous (say) to rings or vector spaces [13, 15]. They defined a hierarchy of process algebras, starting with BPA (Basic Process Algebra) based on the alternative composition operator ($+$) and the sequential composition operator ($\cdot$). These operators are supposed to satisfy axioms A1–A5 in table 5.2. PA (Process Algebra) is BPA extended with the merge ($\|$) and the left merge ($\lfloor\!\lfloor$). The left merge has been introduced for a finite axiomatisation of PA [69]. In PA processes can be described that behave in an interleaved fashion. ACP is PA extended with the communication merge ($|$) to describe communication. PA and ACP also have their characteristic sets of axioms. See [6] for a full treatment of BPA, PA and ACP. Bergstra and Klop applied the axiomatic approach in an early stage to prove some processes specifications correct (see e.g. [14]).

The different algebraic theories triggered a substantial amount of research. For an overview of the developments in process algebra we refer to the textbooks [6, 37, 43, 65]. In the sequel we only mention the developments in process algebra relevant to this thesis.

1.1.2 ω-Completeness

Many authors have observed that bisimulations are not the only interesting process equivalences. For instance in [25] De Nicola and Hennessy argue that two processes can be considered equivalent if they cannot be distinguished by some relevant notion of testing. Due to several different notions of testing this led to a hierarchy of process equivalences. In van Glabbeek [34] a part of this hierarchy, namely for finite, concrete[1], sequential processes, can be found. In [34] each defined equivalence has also been provided with a complete axiomatisation for closed process expressions.

In [68] Moller studied axiomatisations for strong bisimulation for recursion-free CCS expressions without concurrency and for recursion-free CCS expressions with the left merge. He addressed the question whether these axiomatisations are ω-complete, i.e. complete for process expressions containing variables, and answered it affirmatively. This immediately raised the question whether the axiomatisations given in [34] are also ω-complete. In chapter 2 of this thesis this question is answered for almost all axiom sets in [34]. The result is somewhat surprising. Some sets are ω-complete, some other sets are ω-complete if there are at least two actions and others are only ω-complete if there are an infinite number of actions. In order to prove the ω-completeness of these axiom sets in a concise way, a proof technique is developed which is then fruitfully applied.

[1] Concrete means in [34] without internal activity.

1.1.3 An algorithm for branching bisimulation

Another question in the area of process equivalences is to establish whether two processes are equivalent in some sense, and especially to do this automatically. In [48] (see also [49]) an algorithm is presented to decide weak and strong bisimulation on finite state processes in an efficient manner. It is also shown that deciding bisimulation can be done considerably more efficiently than deciding many other equivalences, which are mainly based on traces.

Recently, VAN GLABBEEK and WEIJLAND [35] have proposed an alternative for weak bisimulation, called branching bisimulation, because they felt that weak bisimulation did not properly preserve the branching structure of processes. This led to the question whether the algorithms to decide strong and weak bisimulation could be adapted to branching bisimulation. This question is answered positively in chapter 3 of this thesis. The algorithm for branching bisimulation even outperforms the algorithm for weak bisimulation equivalence (see table 3.1 and also [29]).

1.1.4 Process algebra with (non deterministic) time steps

An old and thoroughly studied question is how to describe real time systems and prove properties about it. In the setting of process algebra (mainly CSP) some early work has been done (see [32], [50] and [78]). The unpleasant complexity of these gave rise to the question whether it was possible to describe and verify real time systems in a simpler way. Based on a simple idea, which turned out to be exactly the idea underlying [79], a simple real time process algebra has been set up, which is reported on in chapter 6. This exercise shows that many real time features can indeed be described and verified in such a simple setting. Recently, real time process algebra has become popular and many comprehensive alternatives have been proposed ([5, 40, 70, 71]).

1.1.5 The definition of μCRL

The work on CCS, CSP and ACP has initiated work on specification languages, i.e. LOTOS [47], PSF [56] and CRL [83], and the programming language OCCAM [46]. These languages are constructed by taking some basic process algebra operators together with some practical extra operators and adding the possibility to describe data. In these languages the relation between process behaviour and data manipulation is important. Therefore, there is a need to get a deeper understanding of it. Due to their intended character LOTOS, CRL, PSF and OCCAM are not perfect in this respect. There have been some attempts to provide a better suited language [20, 66]. However, these are not very close to the specification and programming languages mentioned above. In the last chapter of this thesis a language, called μCRL (micro CRL), is presented which is based on process algebra and abstract data types. This language is designed such that it only contains the most essential operations of both disciplines. It is hoped that μCRL will provide useful theory to understand the interaction between processes

and data and moreover, that this will lead to techniques for constructing concrete and reliable systems in an efficient way.

1.2 Structured operational semantics

The field of semantics is the branch of computer science that studies how programming and specification languages can be provided with a meaning. Languages often contain a wealth of syntactical constructs of which the precise meaning is not always – a priori – obvious. Semantics provides programs written in a language with a meaning by mapping them to a well understood domain.

1.2.1 Historical overview

Semantics came into existence when the first higher level languages were developed (around 1960). Many approaches were pursued at the same time (see e.g. Steel [84] or de Bakker [7]). Operational semantics, as it is called nowadays, was proposed by McCarthey [57] in 1962. The term 'operational semantics' arose somewhere in that decade. Written accounts of it can be found in [81, 54, 55]. Operational semantics views a program as a combination of atomic instructions that generally operate on a data-state. A step function *step* tells, given a program and a data-state, which atomic action is executed and how it transforms the data-state. It yields the transformed data-state and the remainder of the program. An execution of a program in an initial data-state is a sequence

$$(p_0, s_0), (p_1, s_1), ..., (p_n, s_n), ...$$

where p_i are programs, s_i are data-states and $(p_{i+1}, s_{i+1}) = step(p_i, s_i)$. The languages PL/I [73] and Algol 60 [53] are early examples of larger languages that have been provided with such a semantics.

Operational semantics only provides the translation of a program into its meaning indirectly. For instance the meaning of a program may be a mapping from the initial data-state to its execution, or it may be the mapping from the initial data-state to the final data-state if it exists. For larger languages such a two step semantics may become complex and unstructured. It is often especially hard or even impossible to extract the effect of language constructs as functions in the semantic domain. In this case the meaning of programs may be clear, but the meaning of language constructs may be opaque. In order to guarantee that the effect of the language constructs is also clear, another type of semantics, now often called denotational semantics, has been developed. In this semantics the constructs in a language are directly mapped to functions in the semantic domain. One of the advantages of this style of semantics is compositionality which means that the meaning of the whole can be obtained using the meaning of its parts. Denotational semantics can be traced back to Landin (1965, see [51]) and Steel (1969, see [85]). It became especially popular due to the work of Scott and Strachey [82] (see also [86]).

It is often argued that languages should be provided with more than one type of semantics [8, 44]. The semantics are then related which implies that the language has the properties of all of them. An operational semantics is defined as an assistant for the implementor of the language, safe-guarding executability. A denotational semantics is defined to guarantee that a language is compositional. A lot of research has been carried out in this vein (see e.g. [1, 2, 8, 74]).

Structure in operational semantics has also been brought in along another route. Around 1973 some examples emerged where the operational steps of a term were defined using the steps of its subterms [9, 44]. This inspired Plotkin to a more structured approach. In 1977 he introduced a notation that has now become quite popular to give operational semantics to languages [74]. He defined the operational semantics by rules of the form:

$$\frac{(p_1, s_1) \longrightarrow (p_1', s_1') \quad ... \quad (p_n, s_n) \longrightarrow (p_n', s_n')}{(p, s) \longrightarrow (p', s')}$$

where p_i $(1 \leq i \leq n)$ are subterms of p. The meaning of such a rule is that the step (now denoted by an arrow) in the conclusion may be done if all steps in the premises can take place. Plotkin and Hennessy have applied operational semantics of this kind to a simple programming language [38] and CSP [76]. In 1981 Plotkin wrote an overview of how structural operational semantics could be applied in general on programming languages [75]. The term structural refers to the use of smaller processes in the premises than in the conclusion (see also [24]).

In order to give an operational semantics to CCS Hennessy and Plotkin [39] used a slightly modified step relation. They wrote

$$p \stackrel{a}{\longrightarrow} p'$$

meaning that process p transforms itself into process p' by executing an atomic action a. The a above the arrow represents the side effect that p causes by executing a. Non-determinism is modelled by giving p the possibility to do several steps, e.g. $p \stackrel{a}{\longrightarrow} p'$ and $p \stackrel{b}{\longrightarrow} p''$ which means that p can either do an a or a b-step. An explicit data-state turned out not to be necessary and it is therefore omitted. The operational rules now have the form:

$$\frac{p_1 \stackrel{b_1}{\longrightarrow} p_1' \quad ... \quad p_n \stackrel{b_n}{\longrightarrow} p_n'}{p \stackrel{a}{\longrightarrow} p'}$$

where $b_1, ..., b_n$ and a are atomic actions.

Since then structural operational semantics has become a subject of study in itself. In [26] and [27] De Simone studied a format of rules, now called the De Simone-format [36] and he proved a correspondence result with so-called architectural expressions in Meije-SCCS [4, 64].

The De Simone-format has an important property; if processes are constructed by operators defined in this format then each subprocess may be replaced by a

failure equivalent [22] process without changing the completed traces of the process. Two processes are failure equivalent, if they have the same failure traces, i.e. traces of the process marked at the end with a set of actions it cannot perform after executing the trace. In fact failure equivalence is the coarsest such equivalence: it is the completed trace congruence induced by the De Simone-format. From this we may conclude that, assuming that one can only observe the completed traces of a process, failure equivalence is a natural candidate to act as a semantics for processes whenever operators can be defined in the De Simone-format. As languages such as CCS, CSP and ACP can essentially be defined in this way, failure equivalence is one of the more popular process equivalences. Sometimes one can only directly observe the traces of a process. In this case trace equivalence is a natural semantics because it is the trace congruence induced by the De Simone format.

The study of operational semantics has been continued in GROOTE and VAANDRAGER [36]. There the term *structured* operational semantics was used, to stress the uniformity of the notation to define an operational semantics. In [36, 39] it is not required that in the operational rules the behaviour of a term depends on its subterms. This means that structural induction is not adequate to prove properties of operational steps and therefore the term structural operational semantics is not appropriate any more. In structured operational semantics, the induction on terms has been replaced by induction on the depth of proof trees.

In [36] a format, called the *tyft/tyxt*-format has been defined. It has the property that strong bisimulation is a congruence with respect to all operators definable in this format and hence, an operational semantics in *tyft/tyxt*-format induces a compositional semantical mapping from process expressions to the domain of transition systems modulo strong bisimulation equivalence. In [80] this has been made explicit in the setting of metric semantics. Furthermore, it is shown in [36] that the completed trace congruence belonging to the *tyft/tyxt*-format is 2-nested simulation equivalence and the trace congruence is 1-nested simulation. In contrast with failure and trace equivalence, there is less reason to consider nested simulations as natural process semantics as the *tyft/tyxt*-format allows one to define rather unusual operators for forming contexts.

1.2.2 Negative premises

The strong point of structured operational semantics is that it is close to intuition. But it turned out to be convenient and sometimes necessary to use negative information in the premises. This led to operational semantics that were intuitively clear, but that did not fit the formats of rules that were studied in [27, 36, 75]. With negative premises rules have the general format

$$\frac{u_1 \xrightarrow{b_1} u_1' \;\; \ldots \;\; u_n \xrightarrow{b_n} u_n' \quad v_1 \xrightarrow{c_1}\!\!\!\!\!\!/ \;\; \ldots \;\; v_m \xrightarrow{c_m}\!\!\!\!\!\!/}{t \xrightarrow{a} t'}.$$

The idea is that if each term u_i can do a b_i-step to u_i' $(1 \leq i \leq n)$ and if v_j cannot do a c_j-step $(1 \leq j \leq m)$ then t can do an a-step to t'. By default a term cannot do a step unless the existence of that step has been shown. This kind of reasoning must be carried out with extreme care as the assumption that a step cannot take place can lead to the conclusion that it can take place. In the literature there are a number of operational semantics in which carelessness in this sense led to structured operational semantics of which the precise meaning is not completely obvious [19, 45, 67]. This posed the problem of finding techniques to define meaningful structured operational semantics with negative premises.

A first and rather general solution for the use of negative premises in structured operational semantics was provided by Bloom Istrail and Meyer [17, 18]. They defined a format for operational rules, often called the GSOS-format, in which guarded recursion and a form of negative premises are allowed. Their approach is based on a number of observations which imply that negative premises in the GSOS-format always make sense. The trace and completed trace congruence for the GSOS-format is 2/3-bisimulation [52], which is also called ready-simulation. The main features of the GSOS format are the negative premises and copying of processes. Assuming that these are legitimate operations, ready simulation is a natural process equivalence [16].

There are several examples of structured operational semantics that do not fit the GSOS-format. In particular priorities and renaming in combination with unguarded recursion is problematic. In chapter 4 of this thesis a general method, based on stratifications in logic programming [3, 77] is adapted to deal with these cases. The method says that if one can define a so-called stratification for a set of rules, then these rules define a step function in a reasonable way.

Also in this chapter the *ntyft/ntyxt*-format is defined as an extension of the *tyft/tyxt*-format and it is shown that strong bisimulation is the trace and completed trace congruence with respect to this format. Due to the capabilities of the *ntyft/ntyxt*-format for defining unnatural operators, we do not see this as a decisive argument in favour of strong bisimulation.

The stratification technique turned out not to be sufficiently strong to deal with the combination of the rules for internal actions [33], unguarded recursion and priorities. This question is dealt with in chapter 5. Again inspired by logic programming [30, 31], a unique stable transition relation is defined as the step relation belonging to a structured operational semantics with negative premises. This notion generalizes all definitions of step-functions associated to a set of rules given up till now. Moreover, it seems that if a set of rules has no unique stable model, it is hard, if not impossible, to associate a reasonable step function to such a set.

It is shown that under certain pathological circumstances the operational rules for priorities, internal actions and unguarded recursion do not have any stable model. Hence, it is reasonable to conclude that under these circumstances these rules do not make sense.

As the definition of a unique stable model is not constructive, it is often very difficult to prove that a unique stable model exists. The stratification technique

from chapter 4 can be used, but, as mentioned earlier, this technique is not adequate under all circumstances. Therefore a reduction technique, which is also taken from logic programming [30], is adapted and applied. This technique is not very easy to use but, as shown in chapter 5, with the stratification technique from chapter 4 it turns out to be capable of showing the existence of a unique stable transition relation for priorities, internal actions and unguarded recursion under some reasonable conditions.

1.3 The origins of the chapters

The chapters 2 to 7 of this thesis are slightly modified versions of articles that have been published earlier. The chapters are all fully self contained, although they sometimes refer to each other for comparison or because techniques of other chapters have been applied. Below a list is given indicating where these articles appeared. Note that chapter 2 and chapter 6 have been published under different titles.

Chapter 2 J.F. Groote. A new strategy for proving ω–completeness with applications in process algebra. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 314–331. Springer-Verlag, 1990.

Chapter 3 J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M.S. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer-Verlag, 1990.

Chapter 4 J.F. Groote. Transition system specifications with negative premises. Report CS-R8950, CWI, Amsterdam, 1989. An extended abstract appeared in J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90*, Amsterdam, LNCS 458, pages 332–341. Springer-Verlag, 1990.

Chapter 5 R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. Report CS-R9054, CWI, Amsterdam, 1990. An extended abstract appeared in J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Proceedings* 18^{th} *ICALP,* Madrid, pages 481–494, 1991.

Chapter 6 J.F. Groote. Specification and verification of real time systems in ACP. Report CS-R9015, CWI, Amsterdam, 1990. An extended abstract appeared in L. Logrippo, R.L. Probert and H. Ural, editors, *Proceedings* 10^{th} *International Symposium on Protocol Specification, Testing and Verification*, Ottawa, pages 261–274, 1990.

Chapter 7 J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, 1990.

References

[1] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. A denotational semantics of a parallel object-oriented language. Report CS-R8626, CWI, Amsterdam, 1986. To appear in *Information and Computation.*

[2] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Operational semantics of a parallel object-oriented language. In *Proceedings* 13^{th} *ACM Symposium on Principles of Programming Languages,* St. Petersburg, Florida, pages 194–208, 1986.

[3] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, chapter 10, pages 495–574. North-Holland, 1990.

[4] D. Austry and G. Boudol. Algèbre de processus et synchronisations. *Theoretical Computer Science*, 30(1):91–131, 1984.

[5] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[6] J.C.M. Baeten and W.P. Weijland. *Process algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[7] J.W. Bakker. Semantics of programming languages. *Advances in Information Systems Science*, 2:173–227, 1969.

[8] J.W. de Bakker. *Mathematical theory of program correctness.* Prentice Hall International, 1980.

[9] J.W. de Bakker and W.P. de Roever. A calculus for recursive program schemes. In *Proceedings* 1^{st} *ICALP*, pages 167–196. North-Holland, 1972.

[10] H. Bekič. Towards a mathematical theory of processes. Technical Report TR25.125, IBM Laboratory, Vienna, 1971. This document also appeared in [11].

[11] H. Bekič. *Programming Languages and Their Definition* (C.B. Jones, editor), volume 177 of *Lecture Notes in Computer Science.* Springer-Verlag, 1984.

[12] J.A. Bergstra and J.W. Klop. Fixed point semantics in process algebras. Report IW 206, Mathematisch Centrum, Amsterdam, 1982.

[13] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[14] J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. In W. Bibel and K.P. Jantke, editors, *Math. Methods of Spec. and Synthesis of Software Systems '85, Math. Research 31*, pages 9–23, Berlin, 1986. Akademie-Verlag. First appeared as: Report CS-R8404, CWI, Amsterdam, 1984.

[15] J.A. Bergstra and J.W. Klop. Algebra of communicating processes. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Mathematics and Computer Science*, CWI Monograph 1, pages 89–138. North-Holland, Amsterdam, 1986.

[16] B. Bloom. *Ready simulation, bisimulation, and the semantics of CCS-like languages.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 1989.

[17] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: preliminary report. In *Proceedings* 15^{th} *ACM Symposium on Principles of Programming Languages,* San Diego, California, pages 229–239, 1988.

[18] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced. Technical Report 90-1150, Department of Computer Science, Cornell University, Ithaca, New York, August 1990.

[19] T. Bolognesi, F. Lucidi, and S. Trigila. From timed Petri nets to timed LOTOS. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Proceedings* 10^{th} *IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification,* Ottawa, pages 395–408, 1990.

[20] G. Boudol. Towards a lambda-calculus for concurrent and communicating systems. In *TAPSOFT 1989,* Lecture Notes in Computer Science 351, pages 149–161. Springer-Verlag, 1989.

[21] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, 1975.

[22] 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, 1984.

[23] R.M. Burstall and P. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer and D. Michie, editors, *Machine intelligence*, volume 4, pages 17–43. Edinburgh University Press, 1969.

[24] R.M.B. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[25] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[26] R. De Simone. *Calculabilité et expressivité dans l'algebra de processus parallèles* MEIJE. Thèse de 3^e cycle, Univ. Paris 7, 1984.

[27] R. De Simone. Higher-level synchronising devices in MEIJE–SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[28] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112, Academic Press, New York, 1968.

[29] L. Fredlund, P. Ernberg and B. Jonsson. Specification and validation of a simple overtaking protocol using LOTOS. Technical report T90006, SICS, Stockholm, 1990.

[30] A. van Gelder, K. Ross, and J.S. Schlipf. Unfounded sets and well–founded semantics for general logic programs. In *Proceedings of the* 7^{th} *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–230, 1988.

[31] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings* 5^{th} *International Conference on Logic Programming*, pages 1070–1080. MIT press, 1988.

[32] R.T. Gerth and A. Boucher. A timed failures model for extended communicating processes. In Th. Ottmann, editor, *Proceedings* 14^{th} *ICALP,* Karlsruhe, volume 267 of *Lecture Notes in Computer Science*, pages 95–114. Springer-Verlag, 1987.

[33] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 336–347. Springer-Verlag, 1987.

[34] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[35] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989.

[36] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings* 16^{th} *ICALP,* Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation.*

[37] M. Hennessy. *Algebraic theory of processes.* MIT Press, Cambridge, Massachusetts, 1988.

[38] M. Hennessy and G.D. Plotkin. Full abstraction for a simple programming language. In J. Bečvář, editor, 8^{th} *Symposium on Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer-Verlag, 1979.

[39] M. Hennessy and G.D. Plotkin. A term model for CCS. In P. Dembiński, editor, 9^{th} *Symposium on Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*, pages 261–274. Springer-Verlag, 1980.

[40] M. Hennessy and T. Regan. A temporal process algebra. Report 2/90, Computer Science Department, University of Sussex, 1990.

[41] C.A.R. Hoare. Towards a theory of parallel programming. In C.A.R. Hoare and R.H. Perrott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[42] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[43] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[44] C.A.R. Hoare and P.E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.

[45] H. Ichikawa, K. Yamanaka, and J. Kato. Incremental specifications in LOTOS. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Proceedings* 10^{th} *IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification,* Ottawa, pages 183–196, 1990.

[46] INMOS, Ltd. *The occam programming manual.* Prentice Hall International, 1984.

[47] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[48] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the* 2^{nd} *Annual ACM Symposium on Principles of Distributed Computing,* Montreal, Quebec, Canada, 1983.

[49] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[50] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79:210–256, 1988.

[51] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part i. *Communications of the ACM*, 8(2):89–101, 1965.

[52] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Proceedings* 16^{th} *ACM Symposium on Principles of Programming Languages,* Austin, Texas, pages 344–352, 1989.

[53] L.P. Lauer. Formal definition of Algol 60. Technical Report TR.25.088, IBM Lab. Vienna, 1968.

[54] P. Lucas. Formal definition of programming languages and systems. In *Proceedings of the IFIP Congress 1971*, pages 291–297. North Holland, 1972.

[55] P. Lucas. On program correctness and the stepwise development of implementations. In *Proceedings of the Convegno di Informatica Teorica, University of Pisa, March 1973*, pages 219–251, 1973.

[56] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[57] J. McCarthey. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Information Processing 1962*, pages 21–28, 1963.

[58] G. Milne and R. Milner. Concurrent processes and their syntax. *Journal of the ACM*, 26(2):302–321, 1979.

[59] R. Milner. An approach to the semantics of parallel programs. In *Proceedings Convegno di Informatica Teorica, Pisa*, pages 283–302, 1973.

[60] R. Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings Logic Colloquium 1973*, pages 158–173. North-Holland, 1973.

[61] R. Milner. Synthesis of communicating behaviour. In J. Winkowski, editor, *Proceedings of MFCS,* LNCS 64, pages 71–83, 1978.

[62] R. Milner. Flowgraphs and flow algebras. *Journal of the ACM*, 26(4):794–818, 1979.

[63] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[64] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[65] R. Milner. *Communication and concurrency.* Prentice Hall International, 1989.

[66] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I + II. Technical Report ECS-LFCS-89-85 + ECS-LFCS-89-86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989. To appear in *Information and Computation.*

[67] M.W. Mislove and F.J. Oles. A simple language supporting angelic nondeterminism and parallel composition, In *Proceedings* 7^{th} *Conference on the Mathematical Foundations of Programming Semantics*, Pittsburgh, 1991.

[68] F. Moller. *Axioms for concurrency.* PhD thesis, Report CST-59-89, Department of Computer Science, University of Edinburgh, 1989.

[69] F. Moller. The importance of the left merge operator in process algebras. In M.S. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 752–764. Springer-Verlag, 1990.

[70] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 1990.

[71] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: An algebra for timed processes. In M. Broy and C.B. Jones, editors, *Proceedings IFIP Working Conference on Programming Concepts and Methods*, Sea of Gallilea, Israel. North-Holland, 1990.

[72] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[73] PL/I definition Group. Formal definition of PL/I version 1. Report TR25.071, American Nat. Standards Institute, 1986.

[74] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[75] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[76] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II,* Garmisch, pages 199–225, Amsterdam, 1983. North-Holland.

[77] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[78] R. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[79] J.L. Richier, J. Sifakis, and J. Voiron. Une algèbre des processus temporisés, 1987. In A. Arnold, editor, *Actes du deuxième colloque* C^3, Angouléme, 1987.

[80] J.J.M.M. Rutten. Deriving denotational models for bisimulation from structured operational semantics. In M. Broy and C.B. Jones, editors, *Proceedings IFIP Working Conference on Programming Concepts and Methods,* Sea of Gallilea, Israel. North-Holland, 1990.

[81] D. Scott. Outline of a mathematical theory of computation. In *Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.

[82] D.S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*, 1971.

[83] SPECS-semantics. *Definition of MR and CRL Version 2.1*, 1990.

[84] T.B. Steel, editor. *Formal Language Description Languages for Computer Programming. Proceedings of the IFIP Working Conference on Formal Language Description Languages*. North-Holland, 1966.

[85] T.B. Steel. A formalization of semantics for programming language description. In T.B. Steel, editor, *Formal Language Description Languages for Computer Programming. Proceedings of the IFIP Working Conference on Formal Language Description Languages*, pages 25–36, 1969.

[86] J. Stoy. *Denotational Semantics: the Scott-Strachey approach to Programming Language Theory*. MIT press, 1977.

2

Proving ω-Completeness using Inverted Substitutions – with Applications in Process Algebra

(Jan Friso Groote)

A technique for proving ω-completeness based on variable for term substitutions is presented. First we apply this technique to axiom systems for finite, concrete and sequential processes. It turns out that the number of actions is important for these sets to be ω-complete. For the axiom systems for bisimulation and completed trace semantics one action suffices and for the trace axioms 2 actions are enough. The ready, failure, ready trace and failure trace axiom sets are only ω-complete if an infinite number of actions is available. Secondly we consider process algebra with parallelism and show several axiom sets (containing the axioms of standard concurrency) ω-complete.

2.1 Introduction

An equational theory E over a signature Σ is called ω-*complete* iff for all open terms t_1, t_2:

$$E \vdash t_1 = t_2 \quad \Leftrightarrow \quad E \vdash \sigma(t_1) = \sigma(t_2) \text{ for all closed substitutions } \sigma.$$

Not all equational theories are ω-complete: a well known example is the commutativity of the $+$ in Peano arithmetic: all closed instances are derivable from any standard axiomatisation, but the law itself is not. Another example is the three-element groupoid of Murskiĭ [13], who showed that for an ω-complete specification of the groupoid an infinite number of equations is necessary[1].

Several process algebra theories are ω-incomplete, too, and up till now this was more or less ignored (exceptions are Milner [11] and Moller [12]). But there

[1] Bergstra and Heering have shown that using hidden sorts and functions every recursively enumerable equational theory has an ω-complete initial algebra specification [1].

are several reasons why ω-completeness should not be neglected. In the first place equations between open terms play an important role in process algebra. For instance, processes are often described with sets of (open) equations. A complete set of axioms (not necessarily ω-complete) gives no guarantee that such sets of equations can be dealt with in a satisfactory manner. An example of this situation are the so-called 'axioms of standard concurrency' [2] in ACP, which had to be introduced in addition to the 'complete' set of axioms in order to prove the expansion theorem [3]. The status of these axioms became clear only after MOLLER [12] showed that in CCS with interleaving, but without communication, some of the axioms of standard concurrency are required for ω-completeness.

Furthermore, ω-completeness is also useful for theorem provers [8, 9, 15]. In [14] the so-called method 'proof by consistency' is introduced which can be applied to show inductive theorems equationally provable if ω-completeness of the axioms has been shown. In HEERING [6] it is argued that ω-completeness is desirable for the partial evaluation of programs. If $P(x, y)$ is a program with parameters x and y, and x has fixed value c, then the program $P_c(y)$ $(=P(c, y))$ should be evaluated as far as possible. In general this can only be achieved if the evaluation rules are ω-complete.

A more or less standard technique for proving ω-completeness is the following: given a set of axioms E over a signature Σ, find 'normal forms' and show that every open term is provably equal to a normal form. Then prove that for all pairs of different normal forms, closed instantiations can be found that differ in a model $\mathcal{M}$ for E. E does not necessarily have to be complete with respect to $\mathcal{M}$. This last step shows that the equivalence of these instantiations cannot be derived from E. From this ω-completeness of E follows directly. We prove the ω-completeness of the trace and completed trace axioms in this way. This technique has some disadvantages. The proofs are in general quite long and it is often difficult to find a suitable normal form.

In this paper we present an alternative technique that employs inverted substitutions. Basically, it consists of finding a substitution ρ and an *inverted substitution* R. If ρ and R satisfy certain properties then ω-completeness follows directly. The substitution ρ is used to transform a valid (open) equation e into a closed one which is derivable via some proof. Then R transforms this proof into a proof of e by replacing variables for subterms in it. It is completely explained in section 2.3. This inverted substitution technique cannot always be applied, as shown by an example, but if applicable, proofs of ω-completeness turn out to be shorter and for the major part straightforward. Moreover, no reference to a model is necessary. We apply our method to five sets of axioms, which are taken from [4], for finite, concrete (i.e. without internal moves), sequential processes. Only for the set of bisimulation axioms ω-completeness had been shown before by MOLLER [12] using a longer proof. It turns out that the number of actions is important for the axiom sets to be ω-complete. We need an infinite number of actions for the ready trace, failure trace, ready and failure axioms. For the bisimulation and the completed trace axioms at least one action is required whereas for the trace axioms two actions are necessary. Then we study axiom sets for finite,

$\frac{}{x = x}$ (reflexivity) $\quad \frac{x = y}{y = x}$ (symmetry) $\quad \frac{x = y \;\; y = z}{x = z}$ (transitivity)	
$\frac{x_i = y_i \;\; 1 \leq i \leq rank(f)}{f(x_1, ..., x_{rank(f)}) = f(y_1, ..., y_{rank(f)})}$ for all $f \in F$ (congruence)	

Table 2.1: The inference rules of equational logic

concrete process algebra with interleaving without communication (also done in [12]) and interleaving with communication. We give straightforward proofs of the ω-completeness of these sets.

2.2 Preliminaries

Throughout this text we assume the existence of a countably infinite set V of variables with typical elements x, y, z. A (one sorted) *signature* $\Sigma = (F, rank)$ consists of a set of *function names* F, disjoint with V, and a rank function $rank : F \to \mathbb{N}$, denoting the arity of each function name in F. $T(\Sigma)$ is the set of *closed* or *ground terms* over signature Σ and $\mathbb{T}(\Sigma)$ is the set of *open terms* over Σ and V. We use the symbol $\equiv$ for syntactic equality between terms. Furthermore, we have *substitutions* $\sigma, \rho : V \to \mathbb{T}(\Sigma)$ mapping variables to terms. Substitutions are in the standard way extended to functions from terms to terms. An expression of the form $t = u$ $(t, u \in \mathbb{T}(\Sigma))$ is called an *equation* over Σ. The letter e is used to range over equations. An expression of the form

$$\frac{e_1, ..., e_n}{e}$$

is called an *inference rule*. We call $e_1, ..., e_n$ the *premises* and e the *conclusion* of the inference rule. Substitutions are extended to equations and inference rules as expected.

An *equational theory* over a signature Σ is a set E of equations over Σ. These equations are called *axioms*. An equation e can be *proved* from a theory E, notation $E \vdash e$, if e is an instantiation of an axiom in E, i.e. $e \equiv \sigma(e')$ for some axiom $e' \in E$ and substitution σ, or if e is the conclusion of an instantiation of an inference rule in table 2.1 of which all (instantiated) premises can be proved. If it is clear from the context what E is, we sometimes write only e instead of $E \vdash e$. We write $E_1 \vdash E_2$ if $E_1 \vdash e$ for all $e \in E_2$. Note that if $E \vdash t = u$ for $t, u \in T(\Sigma)$, then $t = u$ can be proved using ground axioms and inference rules only.

An equational theory E is *ω-complete* if for all equations e: $E \vdash e$ iff $E \vdash \sigma(e)$ for all substitutions $\sigma : V \to T(\Sigma)$. Note that the implication from left to right

is trivial. So, in general we only prove the implication from the right-hand side to the left-hand side.

2.3 The general proof strategy

Let $\Sigma = (F, rank)$ be a signature and let E be an equational theory over Σ. We present a technique to show that E is ω-complete. Assume $t = t'$ is an equation between open terms that can be proved for all its closed instantiations by the axioms of E. We transform $t = t'$ to a closed equation by a substitution $\rho : V \to T(\Sigma)$ that maps each variable in t and t' to a unique closed (sub)term representing this variable. By assumption $E \vdash \rho(t) = \rho(t')$. We transform the proof of this fact to a proof for $E \vdash t = t'$ by an inverted substitution R which replaces each subterm representing a variable by the variable itself. This transformation yields the desired proof if requirements (2.1), (2.2) and (2.3) below are satisfied. (2.1) says that the translation of $\rho(t) = \rho(t')$ must yield $t = t'$ (or something provably equivalent). In general this only works properly if each subterm representing a variable is unique for that variable and cannot be confused with other subterms. Requirements (2.2) and (2.3) guarantee that the transformed proof is indeed a proof. This is most clearly stated in (2.5), which is a consequence of (2.2) and (2.3).

- For $u \equiv t$ and $u \equiv t'$:
$$E \vdash R(\rho(u)) = u. \tag{2.1}$$
- For each $f \in F$ with $rank(f) > 0$ and $u_1, ..., u_{rank(f)}, u'_1, ..., u'_{rank(f)} \in T(\Sigma)$:
$$E \cup \{u_i = u'_i,\ R(u_i) = R(u'_i) \mid 1 \leq i \leq rank(f)\} \vdash \tag{2.2}$$
$$R(f(u_1, ..., u_{rank(f)})) = R(f(u'_1, ..., u'_{rank(f)})).$$
- For each axiom $e \in E$ and closed substitution $\sigma : V \to T(\Sigma)$:
$$E \vdash R(\sigma(e)). \tag{2.3}$$

Theorem 2.3.1. *Let E be an equational theory over signature Σ. If for each pair of terms $t, t' \in \mathbb{T}(\Sigma)$ that are provably equal for all closed instantiations, there exist a substitution $\rho : V \to T(\Sigma)$ and a mapping $R : T(\Sigma) \to \mathbb{T}(\Sigma)$ satisfying (2.1),(2.2) and (2.3), then E is ω-complete.*

Proof. Let $t, t' \in \mathbb{T}(\Sigma)$ such that for each substitution $\sigma : V \to T(\Sigma)$:
$$E \vdash \sigma(t) = \sigma(t'). \tag{2.4}$$
We must prove that $E \vdash t = t'$. This is an immediate corollary of the following statement:
$$E \vdash u = u' \text{ for } u, u' \in T(\Sigma) \quad \Rightarrow \quad E \vdash R(u) = R(u'). \tag{2.5}$$

It follows from (2.4) that $E \vdash \rho(t) = \rho(t')$. Using (2.5) this implies $E \vdash R(\rho(t)) = R(\rho(t'))$. By (2.1) it follows that $E \vdash t = t'$.

Statement (2.5) is shown by induction on the proof of $E \vdash u = u'$. As u and u' are closed terms, we may assume that the whole proof of $E \vdash u = u'$ consists of closed terms. First we consider the inference rules without premises. There are two possibilities. In the first case $u = u'$ has been shown by the inference rule $x = x$, i.e. $u \equiv \sigma(x) \equiv u'$ for some substitution $\sigma : V \to T(\Sigma)$. Clearly, $E \vdash R(u) = R(u')$ using the same inference rule and a substitution $\sigma' : V \to \mathbb{T}(\Sigma)$ defined by $\sigma'(x) = R(\sigma(x))$. Otherwise, $u = u'$ is an instantiation $\sigma(e)$ of an axiom $e \in E$. Using (2.3) it follows immediately that $E \vdash R(\sigma(e))$.

We check here the inference rules with premises. First we deal with the rule for transitivity. So assume $E \vdash u = u'$ has been proved using $E \vdash u = u''$ and $E \vdash u'' = u'$. By induction we know that there are proofs for $E \vdash R(u) = R(u'')$ and $E \vdash R(u'') = R(u')$. Applying the inference rule for transitivity again we have that $E \vdash R(u) = R(u')$. The rule for symmetry can be dealt with in the same way. Now suppose that $E \vdash f(u_1, ..., u_{rank(f)}) = f(u'_1, ..., u'_{rank(f)})$ has been proved using $E \vdash u_i = u'_i$ $(1 \leq i \leq rank(f))$. By induction we know that $E \vdash R(u_i) = R(u'_i)$. Using (2.2), it follows immediately that $E \vdash R(f(u_1, ..., u_{rank(f)})) = R(f(u'_1, ..., u'_{rank(f)}))$. □

This proof strategy cannot always be applied. This is illustrated by the following example.

Example 2.3.2. Suppose we have an axiomatisation for the natural numbers with a function max giving the maximum of any pair of numbers. In the signature we have a 0, a successor function S and max. The following set E_{max} of axioms is easily seen to be complete with respect to the standard interpretation.

$$\begin{aligned} &max(x, 0) = x, \\ &max(0, x) = x, \\ &max(S(x), S(y)) = S(max(x, y)). \end{aligned}$$

An ω-complete characterisation of the natural numbers with maximum is

$$\begin{aligned} &max(x, 0) = x, \\ &max(S(x), S(y)) = S(max(x, y)), \\ &max(S(x), x) = S(x), \\ &max(x, x) = x, \\ &max(x, y) = max(y, x), \\ &max(x, max(y, z)) = max(max(x, y), z). \end{aligned}$$

However, it is impossible to use our technique to prove this or any other extension of E_{max} ω-complete. This can be seen by considering the following two terms:

$$\begin{aligned} &t_1 = max(S(0), x) \text{ and} \\ &t_2 = x. \end{aligned}$$

We can see that these terms are not provably equal because with $x = 0$, the first term is equal to $S(0)$ and the second is equal to 0. Note that this is the only way to see the difference. If any term that is not equal to 0 is substituted for x then both terms are equivalent.

Suppose we would like to apply our technique in this case. If we take ρ such that $\rho(x) = 0$ then we must define the translation R such that $R(\rho(x)) = R(0) = x$. But then $R(\rho(0)) = x$ which is not (provably) equivalent to 0, violating (2.1). Suppose ρ is chosen such that $\rho(x) \neq 0$ and R could be defined such that (2.1) holds, i.e. $E_{max} \vdash R(\rho(t_i)) = t_i$ $(i = 1, 2)$. This implies that (2.5), which follows from (2.2) and (2.3), cannot hold because it implies that $E_{max} \vdash t_1 = t_2$.

So, this example shows that the inverted substitution technique is not generally applicable, but as will be shown in the next sections, there are enough cases where application of this technique leads to attractive proofs.

2.4 Applications in finite, concrete, sequential process algebra

In the remainder of this paper we apply our technique to prove completeness of several axiom systems. In this section sets given for BCCSP in [4] are studied. BCCSP is a basic CCS and CSP-like language for finite, concrete, sequential processes. It is parameterised by a set *Act* of actions representing the elementary activities that can be performed by processes. We write $|Act|$ for the number of elements in *Act* ($|Act| = \infty$ if *Act* has an infinite number of elements). The language BCCSP contains a constant δ, which is comparable to **0** or NIL in CCS and to STOP in CSP. We call δ *inaction* or sometimes *deadlock*. There is an *alternative composition* operator $+$ with its usual meaning and, furthermore, there is an *action prefix* operator $a :$ for each action a in *Act*.

In the sequel we will often use sums of arbitrary finite size. It is convenient to have a notation for these. Therefore we introduce the abbreviation:

$$\sum_{i \in I} t_i = t_{i_1} + ... + t_{i_n}$$

where $I = \{i_1, ..., i_n\}$ is a finite index set and $t_i \in \mathbb{T}(\text{BCCSP})$ $(i \in I)$. We take $\sum_{i \in \emptyset} t_i = \delta$. Note that this notation is only justified if $+$ is commutative and associative. We only use this notation when this is the case.

The depth $|t|$ of a term $t \in \mathbb{T}(\text{BCCSP})$ is inductively defined as follows:

$$|\delta| = 0, \qquad |x| = 0 \text{ for all } x \in V,$$
$$|a : t| = 1 + |t| \text{ for all } a \in Act, \qquad |t_1 + t_2| = \max(|t_1|, |t_2|).$$

In table 2.2 we present several axiom systems, taken from [4], corresponding to several semantics in process algebra. We investigate the ω-completeness of these sets. On the top line of this table we find their abbreviations: B stands

	B	RT	FT	R	F	CT	T
$x+y=y+x$	+	+	+	+	+	+	+
$(x+y)+z=x+(y+z)$	+	+	+	+	+	+	+
$x+x=x$	+	+	+	+	+	+	+
$x+\delta=x$	+	+	+	+	+	+	+
(see (2.6) in text)		+	+	v	v	v	v
$a:x+a:y=a:x+a:y+a:(x+y)$			+		v	v	v
$a(b:x+u)+a:(b:y+v)=$							
$\quad a:(b:x+b:y+u)+a:(b:x+b:y+v)$				+	+	v	v
$a:x+a:(y+z)=a:x+a:(x+y)+a:(y+z)$					+	ω	v
$a:(b:x+u)+a:(c:y+v)=a:(b:x+c:y+u+v)$						+	v
$a:x+a:y=a:(x+y)$							+

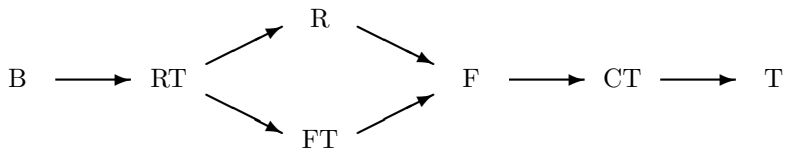
Table 2.2: Axioms for several process algebra semantics

for *Bisimulation*, RT for *Ready Trace*, FT for *Failure Trace*, R for *Ready*, F for *Failure*, CT for *Completed Trace* and finally T for *Trace* semantics. The axioms that are necessary for ready trace semantics (besides the axioms for bisimulation) are given by the following scheme:

$$a:(\sum_{i\in I} a_i:x_i+y)+a:(\sum_{i\in J} a_i:x_i+y)=a:(\sum_{i\in I\cup J} a_i:x_i+y) \qquad (2.6)$$

where $\{a_i \mid i\in I\}=\{a_i \mid i\in J\}$, and $x_i,y\in V$ $(i\in I\cup J)$. This scheme differs from the axiomatisation given in [4], where an additional function name I and a conditional axiom were used to axiomatise ready trace semantics. We do not want to introduce these concepts here, although we show in lemma 2.4.2 that for terms over the signature BCCSP, the ω-completeness of the axiomatisation of [4] follows directly from the ω-completeness of RT.

Let X stand for any of the semantics B,RT,... The symbol 'v' in a column of semantics X indicates that an axiom is derivable from the other axioms valid for X. The symbol '+' means that the axiom is required for a complete axiomatisation of the models given in [4] and 'ω' means that the axiom is only necessary for an ω-complete axiomatisation. It follows immediately that:



where the semantics to the left are finer than the semantics to the right. The semantics FT and R are incomparable. The abbreviation for a semantics is also used to denote the set of axioms necessary for its ω-complete axiomatisation.

Lemma 2.4.1. *Let $t, u \in \mathbb{T}$(BCCSP). If* $\mathrm{T} \vdash t = u$, *then* $|t| = |u|$.

Proof. Direct using induction on the proof of $t = u$. □

As $\mathrm{T} \vdash \mathrm{B}$, $\mathrm{T} \vdash \mathrm{RT}$ etc. it immediately follows from the last lemma that '$\mathrm{X} \vdash t = u \Rightarrow |t| = |u|$', where X is any of the sets B, RT, etc.

2.4.1 The semantics B

We start by considering the axioms for bisimulation semantics. If *Act* contains at least one element, then B is ω-complete. This fact has already been shown in [12]. Note that it makes no sense to investigate the situation where $Act = \emptyset$, because in that case all closed terms have the form δ, $\delta + \delta$, $\delta + \delta + ...$ and therefore they are equal and we only require the axiom $x = y$ for an ω-complete axiomatisation.

Theorem 2.4.1.1. *If* $|Act| \geq 1$, *then the axiom system* B *is ω-complete.*

Proof. As $|Act| \geq 1$, *Act* contains at least one action a. This action plays an important role in this proof. We follow the lines set out in theorem 2.3.1. So, assume we have two terms $t, t' \in \mathbb{T}$(BCCSP). Select a natural number $m > \max(|t|, |t'|)$ and define $\rho : V \to T$(BCCSP) by:

$$\rho(x) = a^{n(x)\cdot m} : \delta$$

where $a^k : \delta$ is an abbreviation of k applications of $a :$ to δ and $n : V \to \mathbf{N} - \{0\}$ is a function assigning a unique natural number to each variable in V. Define the inverted substitution $R : T$(BCCSP) $\to \mathbb{T}$(BCCSP) as follows:

$R(\delta) = \delta$,
$R(t + u) = R(t) + R(u)$,
$R(b : t) = b : R(t)$ if $b \neq a$ or $|b : t| \neq m \cdot n(x)$ for all $x \in V$,
$R(a : t) = x$ if $|a : t| = m \cdot n(x)$ for some $x \in V$.

We now check conditions (2.1), (2.2) and (2.3) of theorem 2.3.1. We prove (2.1) with induction on a term $u \in \mathbb{T}$(BCCSP) provided $|u| < m$. Note that this is sufficient as $|t| < m$ and $|t'| < m$.

$R(\rho(\delta)) = \delta$,
$R(\rho(x)) = R(a^{n(x)\cdot m} : \delta) = x$,
$R(\rho(u_1 + u_2)) = R(\rho(u_1)) + R(\rho(u_2)) = u_1 + u_2$,
$R(\rho(b : u)) = b : R(\rho(u)) = b : u$ if $b \neq a$,
$R(\rho(a : u)) = R(a : \rho(u)) =^* a : R(\rho(u)) = a : u$.

$=^*$ follows directly from the observation that $|a : \rho(u)| \neq m \cdot n(x)$ for all $x \in V$. In order to see this, first note that $1 \leq |a : u| < m$. If u does not contain variables, it is clear that $1 \leq |a : \rho(u)| < m$ and hence, $|a : \rho(u)| \neq m \cdot n(x)$. So, suppose u contains variables. By applying ρ to u each variable x is replaced by $a^{n(x)\cdot m} : \delta$.

So $|a : \rho(u)| = p + n(x) \cdot m$ where x is a variable in u such that there is no other variable y in u with $n(y) > n(x)$ and p $(1 \leq p < m)$ is the 'depth' of the deepest occurrence of x in u. As $1 \leq p < m$, $|a : \rho(u)| \neq n(x) \cdot m$ for each $x \in V$.

Now we check (2.2). Assume $\mathrm{B} \vdash u_i = u_i'$ and $\mathrm{B} \vdash R(u_i) = R(u_i')$ for $u_i, u_i' \in T(\mathrm{BCCSP})$ and $i = 1, 2$. We find that:

$\mathrm{B} \vdash R(u_1 + u_2) = R(u_1) + R(u_2) = R(u_1') + R(u_2') = R(u_1' + u_2')$.
$\mathrm{B} \vdash R(b : u_1) = b : R(u_1) = b : R(u_1') = R(b : u_1')$ if $b \neq a$.
$\mathrm{B} \vdash R(a : u_1) =^* a : R(u_1) = a : R(u_1') =^+ R(a : u_1')$
if $|a : u_1| \neq m \cdot n(x)$ for all $x \in V$.

$=^*$ follows directly from the condition. As $\mathrm{B} \vdash u_1 = u_1'$ it follows that $|a : u_1| = |a : u_1'|$ (cf. lemma 2.4.1) and hence, $|a : u_1'| \neq m \cdot n(x)$ for all $x \in V$. This justifies $=^+$.

$$\mathrm{B} \vdash R(a : u_1) = x =^* R(a : u_1') \text{ if } |a : u_1| = m \cdot n(x) \text{ for some } x \in V.$$

It follows that $|a : u_1'| = m \cdot n(x)$ explaining $=^*$.

Finally, we must check (2.3). This is trivial as the axioms do not contain actions. We only check the axiom $x + y = y + x$. The other axioms can be dealt with in the same way. Let $\sigma : V \to T(\mathrm{BCCSP})$ be a substitution, then:

$$\begin{aligned}\mathrm{B} \vdash R(\sigma(x + y)) &= R(\sigma(x)) + R(\sigma(y)) \\ &= R(\sigma(y)) + R(\sigma(x)) = R(\sigma(y + x)).\end{aligned}$$

□

2.4.2 The semantics RT,FT,R and F

We show that the sets of axioms RT,FT,R and F are all ω-complete in case *Act* is infinite. If *Act* is finite, we have the following identity:

$$a : \sum_{i \in J} a_i : \delta + a : (x + \sum_{i \in J} a_i : \delta) = a : (x + \sum_{i \in J} a_i : \delta) \tag{2.7}$$

where $\{a_i \mid i \in J\} = Act$. Each closed instance of this identity is derivable from the axioms of RT,FT,R or F. However, (2.7) is not derivable in its general form: if (2.7) were derivable, then it also holds if *Act* is extended by a 'fresh' action $b \notin \{a_i \mid i \in J\}$. Define a substitution σ satisfying $\sigma(x) = b : \delta$. Applying σ to (2.7) yields:

$$a : \sum_{i \in J} a_i : \delta + a : (b : \delta + \sum_{i \in J} a_i : \delta) = a : (b : \delta + \sum_{i \in J} a_i : \delta).$$

but this equation does not hold in the failure model [4]. Hence, it is not derivable from F and therefore it can certainly not be derived from RT,FT or R.

So, in order to prove RT,FT,R and F ω-complete, *Act* must at least be countably infinite. The following theorem shows that this condition is also sufficient.

Theorem 2.4.2.1. *If $|Act|$ is infinite, then the axiom sets* RT,FT,R *and* F *are ω-complete.*

Proof. Take two terms t, t'. Define a substitution $\rho : V \to T(\mathrm{BCCSP})$ by:

$$\rho(x) = a_x : \delta$$

where a_x is a unique action for each $x \in V$ and a_x must not occur in either t or t'. Note that these actions can always be found as $|Act| = \infty$. Define $R : T(\mathrm{BCCSP}) \to \mathbb{T}(\mathrm{BCCSP})$ as follows:

$R(\delta) = \delta$,
$R(a : u) = a : R(u)$ if $a \neq a_x$ for each $x \in V$,
$R(a_x : u) = x$,
$R(u_1 + u_2) = R(u_1) + R(u_2)$.

Condition (2.1) of theorem 2.3.1 can be checked by induction on the structure of open terms not containing action prefix operators $a_x :$.

$R(\rho(\delta)) = \delta$,
$R(\rho(x)) = R(a_x : \delta) = x$,
$R(\rho(a : u)) = R(a : \rho(u)) = a : R(\rho(u)) = a : u$ as $a \neq a_x$ for each $x \in V$,
$R(\rho(u_1 + u_2)) = R(\rho(u_1)) + R(\rho(u_2)) = u_1 + u_2$.

Condition (2.2) can be checked in the same straightforward manner. Suppose $\mathrm{X} \vdash R(u_i) = R(u_i')$ for $u_i, u_i' \in T(\mathrm{BCCSP})$ and $i = 1, 2$ where X may be replaced by either RT,FT,R or F. Then:

$\mathrm{X} \vdash R(a : u_1) = a : R(u_1) = a : R(u_1') = R(a : u_1')$ if $a \neq a_x$ for all $x \in V$.
$\mathrm{X} \vdash R(a_x : u_1) = x = R(a_x : u_1')$.
$\mathrm{X} \vdash R(u_1 + u_2) = R(u_1) + R(u_2) = R(u_1') + R(u_2') = R(u_1' + u_2')$.

Finally, we check (2.3). We restrict ourselves to the ready trace axiom scheme (2.6). All other axioms can be dealt with in the same way. First we assume that $a = a_x$ for some $x \in V$. Let $\sigma : V \to T(\mathrm{BCCSP})$ be a substitution. Then the following holds in RT:

$$R(a_x : (\sum_{i \in I} a_i : \sigma(x_i) + \sigma(y)) + a_x : (\sum_{i \in J} a_i : \sigma(x_i) + \sigma(y))) =$$
$$x + x = x =$$
$$R(a_x : (\sum_{i \in I \cup J} a_i : \sigma(x_i) + \sigma(y))).$$

In case $a \neq a_x$ for each $x \in V$, we have that RT proves:

$$
\begin{aligned}
&R(a:(\sum_{i\in I} a_i:\sigma(x_i)+\sigma(y))+a:(\sum_{i\in J} a_i:\sigma(x_i)+\sigma(y)))=\\
&a:(\sum_{i\in I} R(a_i:\sigma(x_i))+R(\sigma(y)))+a:(\sum_{i\in J} R(a_i:\sigma(x_i))+R(\sigma(y)))=\\
&a:(\sum_{i\in I-\{i\in I|a_i=a_x\}} a_i:R(\sigma(x_i))+\sum_{x\in\{x|a_x=a_i\wedge i\in I\}} x+R(\sigma(y)))+\\
&a:(\sum_{i\in J-\{i\in J|a_i=a_x\}} a_i:R(\sigma(x_i))+\sum_{x\in\{x|a_x=a_i\wedge i\in J\}} x+R(\sigma(y)))=^*\\
&a:(\sum_{i\in (I\cup J)-\{i\in I\cup J|a_i=a_x\}} a_i:R(\sigma(x_i))+\sum_{x\in\{x|a_x=a_i\wedge i\in J\}} x+R(\sigma(y)))=\\
&R(a:(\sum_{i\in I\cup J} a_i:\sigma(x_i)+\sigma(y))).
\end{aligned}
$$

$=^*$ follows from the observations that

$$\{a_i \mid i\in I,\ a_i\neq a_x \text{ for some } x\in V\}=\{a_i \mid i\in J,\ a_i\neq a_x \text{ for some } x\in V\}$$

and

$$\{x \mid a_x=a_i\wedge i\in I\}=\{x \mid a_x=a_i\wedge i\in J\}.$$

This last line follows directly from the fact that $\{a_i \mid i\in I\}=\{a_i \mid i\in J\}$. □

In [4] the semantics for RT is characterised by the following conditional axiom:

$$I(x)=I(y)\ \Rightarrow\ a:x+a:y=a:(x+y). \tag{2.8}$$

It may be used in equational proofs in the same way as the inference rules in table 2.1, where $I(x)=I(y)$ is the premise and $a:x+a:y=a:(x+y)$ is the conclusion. The auxiliary function name I gives the initial actions of a term. It is subject to the following axioms:

$I(\delta)=\delta$,
$I(a:x)=a:\delta$,
$I(x+y)=I(x)+I(y)$.

These axioms, together with the inference rule mentioned above and the axioms B are called RT$'$. For terms not including I, RT$'$ can prove every equation that can be proved with RT, which we show in the next lemma. As a result, we find that RT$'$ is ω-complete for BCCSP-expressions.

Lemma 2.4.2.1. *If $t,t'\in\mathbb{T}$(BCCSP), then:*

$$\text{RT}\vdash t=t' \quad\Rightarrow\quad \text{RT}'\vdash t=t'.$$

Proof. The proof is straightforward if one notes that an application of (2.6) can easily be replaced by an application of the inference rule in RT$'$ using the axioms for I. □

2.4.3 The completed trace axioms

We now show the ω-completeness for the axiom set CT. However, it is not possible to use the inverted substitution technique. This will be shown in example 2.4.3.3. Therefore, we use the more traditional technique. Hence, it is necessary to explicitly define the completed trace semantics for BCCSP. In CT the meaning of a process is its set of traces that end in inaction.

Definition 2.4.3.1. The interpretation $[\![.]\!]_{\mathrm{CT}} : T(\mathrm{BCCSP}) \to 2^{Act^\star}$ (the set of subsets of strings over *Act*) is defined as follows:

$$[\![\delta]\!]_{\mathrm{CT}} = \emptyset,$$
$$[\![a : t]\!]_{\mathrm{CT}} = \{a \star s \mid s \in [\![t]\!]_{\mathrm{CT}}\} \cup \{a \mid [\![t]\!]_{\mathrm{CT}} = \emptyset\},$$
$$[\![t_1 + t_2]\!]_{\mathrm{CT}} = [\![t_1]\!]_{\mathrm{CT}} \cup [\![t_2]\!]_{\mathrm{CT}}.$$

We say that $t_1, t_2 \in T(\mathrm{BCCSP})$ are *completed trace equivalent*, notation $t_1 =_{\mathrm{CT}} t_2$, iff $[\![t_1]\!]_{\mathrm{CT}} = [\![t_2]\!]_{\mathrm{CT}}$.

Lemma 2.4.3.2 *(Soundness).* *Let* $t_1, t_2 \in T(\mathrm{BCCSP})$*:*

$$\mathrm{CT} \vdash t_1 = t_2 \quad \Rightarrow \quad t_1 =_{\mathrm{CT}} t_2.$$

Proof. Straightforward using the definitions. □

The following lemma gives some equations that are derivable from CT.

Lemma 2.4.3.2. $\mathrm{CT} \vdash$

(a) $a : x + a : (x + y) = a : x + a : y + a : (x + y)$,

(b) $a : (x + y) + a : x + a : z = a : (x + y + z) + a : x + a : z$,

(c) $a : (b : x + y) + a : z = a : (b : x + y + z) + a : z$.

Moreover, B+(b)+(c) $\vdash$ CT. Hence, B+(b)+(c) is an alternative ω-complete axiomatisation for completed trace semantics.

For completed trace semantics theorem 2.4.3.4 states the completeness of the axioms with respect to the given model. As t_1 and t_2 may be open terms, ω-completeness is implied also. The technique as set out in theorem 2.3.1 does not work. This is shown in the following example.

Example 2.4.3.3. Consider the following two BCCSP-terms.

$$t_1 = a : x + a : (a : \delta + x),$$
$$t_2 = a : (a : \delta + x).$$

These two terms are clearly different in CT as for a substitution σ with $\sigma(x) = \delta$, $\sigma(t_1)$ has a completed trace a which is not available in $\sigma(t_2)$. For every substitution σ' with $\sigma'(x) \neq \delta$, $\sigma'(t_1) =_{CT} \sigma'(t_2)$. Hence, using the same arguments as in example 2.3.2, we cannot apply our technique.

The next theorem states that CT is ω-complete.

Theorem 2.4.3.4. *If $|Act| \geq 1$, then for all $t_1, t_2 \in \mathbb{T}(\mathrm{BCCSP})$, we have that:*

$$\forall \sigma : V \to T(\mathrm{BCCSP})\ \sigma(t_1) =_{\mathrm{CT}} \sigma(t_2) \quad \Rightarrow \quad \mathrm{CT} \vdash t_1 = t_2.$$

Proof. We write $\vec{x}$ for $\sum_{x \in W} x$ where $W \subseteq V$ is a finite set of variables. $\vec{x}$ is called a *sequence of variables*. We call a term t a *CT-normal form* iff

$$t \equiv \sum_{a \in A} (a : (t_a + \vec{y}_a) + \sum_{j \in J_a} a : \vec{x}_j)$$

satisfying for each $a \in A$:

(a) $A \subseteq Act$ is a finite set of actions,

(b) t_a is a CT-normal form,

(c) for each $j \in J_a$, it holds that all variables in $\vec{x}_j$ also appear in $\vec{y}_a$,

(d) if t_a has no initial actions (see below), then for each $j \in J_a$, there is a variable x_j in $\vec{y}_a$ that is not present in $\vec{x}_j$,

(e) for $j_1, j_2 \in J_a$ and $j_1 \neq j_2$, $\vec{x}_{j_1}$ contains a variable not present in $\vec{x}_{j_2}$.

If for a CT-normal form t: $|A| > 0$, then we say that t has *initial actions*. Note that if t has no initial actions, then t is equal to δ.

Fact 1. For each term $t \in \mathbb{T}(\mathrm{BCCSP})$, there is a term u in CT-normal form such that $\mathrm{CT} \vdash t = u + \vec{x}$ for some sequence of variables $\vec{x}$.

Proof of fact 1. This proof is based on induction on the structure of t. Let $t \equiv \delta$. If we take $A = \emptyset$ and $\vec{x} \equiv \delta$, we obtain the CT-normal form $\delta + \delta$, which is clearly provably equal to δ. Let $t \equiv x$. Then we can take the CT-normal form $\delta + x$.

If $t \equiv a : t'$, then, by induction, there is a CT-normal form u', such that $u' + \vec{x}'$, for some sequence of variables $\vec{x}'$, is provably equal to t'. Hence:

$$\mathrm{CT} \vdash t = a : t' = a : (u' + \vec{x}') + \delta$$

and $a : (u' + \vec{x}') + \delta$ is a CT-normal form (conditions (a),(b),(c),(d) and (e) can easily be checked).

Suppose $t \equiv t_1 + t_2$. Then, by induction, there are CT-normal forms u_1 and u_2 such that for sequences $\vec{z}_1$ and $\vec{z}_2$:

$$\mathrm{CT} \vdash t_1 = u_1 + \vec{z}_1 \text{ and } \mathrm{CT} \vdash t_2 = u_2 + \vec{z}_2.$$

The term u_l $(l = 1, 2)$ can be written as:

$$u_l \equiv \sum_{a \in A^l} (a : (t_a^l + \vec{y}_a^l) + \sum_{j \in J_a^l} a : \vec{x}_j).$$

We assume that $J_a^1 \cap J_a^2 = \emptyset$ for $a \in A^1 \cap A^2$. The term $t_1 + t_2$ is provably equal to $u_1 + u_2 + \vec{z}_1 + \vec{z}_2$, which is, using CT, equal to:

$$\sum_{a \in A^1 \cap A^2} (a : (t_a + \vec{y}_a^1) + a : (t_a^2 + \vec{y}_a^2) + \sum_{j \in J_a^1 \cup J_a^2} a : \vec{x}_j) + \tag{2.9}$$
$$\sum_{a \in A^1 - A^2} (a : (t_a^1 + \vec{y}_a^1) + \sum_{j \in J_a^1} a : \vec{x}_j) +$$
$$\sum_{a \in A^2 - A^1} (a : (t_a^2 + \vec{y}_a^2) + \sum_{j \in J_a^2} a : \vec{x}_j) + \vec{z}_1 + \vec{z}_2.$$

Now note that the summands in (2.9) are in CT-normal form for $a \in A^1 - A^2$ and $a \in A^2 - A^1$. In order to prove fact 1, it is sufficient to only transform the first summand into CT-normal form. Therefore we consider for each $a \in A_1 \cap A_2$:

$$a : (t_a^1 + \vec{y}_a^1) + a : (t_a^2 + \vec{y}_a^2) + \sum_{j \in J_a^1 \cup J_a^2} a : \vec{x}_j. \tag{2.10}$$

First we deal with the possibility that both t_a^1 and t_a^2 in (2.10) have initial actions. In this case we can apply the axiom

$$a : (b : x + u) + a : (c : y + v) = a : (b : x + c : y + u + v)$$

to rewrite (2.10) to:

$$a : (t_a^1 + t_a^2 + \vec{y}_a^1 + \vec{y}_a^2) + \sum_{j \in J_a^1 \cup J_a^2} a : \vec{x}_j. \tag{2.11}$$

This term is a CT normal form, satisfying conditions (a) and (b). Later on we show how conditions (c), (d) and (e) are satisfied.

Now, suppose that t_a^1 has no initial actions (the case where t_a^2 has no initial actions is symmetric and therefore skipped). Then t_a^1 is equal to δ. Hence (2.10) has the form:

$$a : (t_a^2 + \vec{y}_a^2) + (a : \vec{y}_a^1 + \sum_{j \in J_a^1 \cup J_a^2} a : \vec{x}_j) \tag{2.12}$$

and this term satisfies conditions (a) and (b).

From (2.11) and (2.12) we may assume that (2.10) is provably equal to a term of the form:

$$a : (t + \vec{y}) + \sum_{j \in J} a : \vec{x}_j$$

which is a CT normal form satisfying conditions (a) and (b). Now, assume condition (c) does not hold. This means that there is a $k \in J$ and a variable x in $\vec{x}_k$ such that x is not in $\vec{y}$. We can prove from CT using the typical RT axiom that:

$$a : (t + \vec{x}) + a : \vec{x}_k + \sum_{j \in J - \{k\}} a : \vec{x}_j =$$
$$a : (t + \vec{x} + \vec{x}_k) + a : (t + \vec{x}) + a : \vec{x}_k + \sum_{j \in J - \{k\}} a : \vec{x}_j.$$

If t has initial actions then this is equal to:

$$a : (t + \vec{x} + \vec{x}_k) + (a : \vec{x}_k + \sum_{j \in J - \{k\}} a : \vec{x}_j),$$

otherwise, it is equal to:

$$a : (\vec{x} + \vec{x}_k) + a : \vec{x} + a : \vec{x}_k + \sum_{j \in J - \{k\}} a : \vec{x}_j.$$

As J is finite and each $\vec{x}_j$ $(j \in J)$ contains a finite number of variables, we can repeat this step a finite number of times and satisfy condition (c).

Hence, we may assume that we have a term $a : (t + \vec{y}) + \sum_{j \in J} a : \vec{x}_j$ satisfying conditions (a), (b) and (c). Suppose t has no initial actions and for some $j \in J$, $\vec{x}_j$ and $\vec{y}$ contain the same variables. Apply axiom $x = x + x$ to fulfill condition (d). Note that the conditions (a), (b) and (c) are not invalidated by this operation. Now we consider a term:

$$a : (t + \vec{y}) + \sum_{j \in J} a : \vec{x}_j$$

which satisfies condition (a), (b), (c) and (d), but for which (e) does not hold. This means that there are sequences of variables $\vec{x}_{j_1}$ and $\vec{x}_{j_2}$ $(j_1, j_2 \in J$ and $j_1 \neq j_2)$ such that all variables in $\vec{x}_{j_1}$ are also present in $\vec{x}_{j_2}$. Hence, there is a sequence of variables $\vec{x}$ such that $\vec{x}_{j_1} + \vec{x} = \vec{x}_{j_2}$ such that $\vec{x}$ and $\vec{x}_{j_1}$ do not have variables in common. Now we apply lemma 2.4.3(a) to show:

$$\begin{aligned}
&a : (t + \vec{y}) + a : (\vec{x}_{j_1} + \vec{x}) + a : \vec{x}_{j_1} + \sum_{j \in J - \{j_1, j_2\}} a : \vec{x}_j = \\
&a : (t + \vec{y}) + a : (\vec{x}_{j_1} + \vec{x}) + a : \vec{x}_{j_1} + a : \vec{x} + \sum_{j \in J - \{j_1, j_2\}} a : \vec{x}_j =^* \\
&a : (t + \vec{y}) + \sum_{j \in J - \{j_2\}} a : \vec{x}_j + a : \vec{x}.
\end{aligned}$$

For $=^*$ we use condition (c) and the typical FT axiom. It can be seen that this step can also only be applied a finite number of times. So after some time we achieve a term satisfying all conditions for a CT-normal form. □

Let in the sequel for $l = 1, 2$:

$$t_l \equiv \sum_{a \in A^l} (a : (t_a^l + \vec{y}_a^l) + \sum_{j \in J_a^l} a : \vec{x}_j).$$

We say that t_1 and t_2 are *different* if one of the following holds:

(1) $A^1 \neq A^2$,

(2) $A^1 = A^2$ and for some $a \in A^1$, t_a^1 and t_a^2 are different,

(3) $A^1 = A^2$ and for some $a \in A^1$, $\vec{y}_a^1$ and $\vec{y}_a^2$ do not contain the same variables.

(4) $A^1 = A^2$ and for some $a \in A^1$, there is a $j_1 \in J_a^1$ such that for each $j_2 \in J_a^2$, $\vec{x}_{j_1}$ and $\vec{x}_{j_2}$ do not contain the same variables or, symmetrically, there is a $j_2 \in J_a^2$ such that for each $j_1 \in J_a^1$, $\vec{x}_{j_2}$ and $\vec{x}_{j_1}$ do not contain the same variables.

Fact 2. If two CT-normal forms t_1 and t_2 are not different, then $\mathrm{B} \vdash t_1 = t_2$ (and thus $\mathrm{CT} \vdash t_1 = t_2$).

Proof of fact 2. Straightforward. □

Fact 3. Let t be a CT-normal form. Let $m \in \mathbf{N}$ be selected such that $m > |t|$. Let $\sigma : V \to T(\mathrm{BCCSP})$ be a substitution such that $\sigma(x) = \delta$ or $\sigma(x) = b^m : \delta$ where $b \in Act$. For each $s \in [\![\sigma(t)]\!]_{\mathrm{CT}}$: $1 \leq |s| \leq |t|$ or $|s| > m$.

Proof of fact 3. By definition $|s| \geq 1$. The remainder of this fact follows directly by induction on the structure of t. □

An important corollary of this fact is that $|s| \neq m$.

Fact 4. Let t_1 and t_2 be two different CT-normal forms. Let $m \in \mathbf{N}$ be selected such that $m > \max(|t_1|, |t_2|)$ and let $b \in Act$. There is a substitution $\sigma : V \to T(\mathrm{BCCSP})$ with for each $x \in V$: $\sigma(x) = \delta$ or $\sigma(x) = b^m : \delta$ such that $\sigma(t_1) \neq_{\mathrm{CT}} \sigma(t_2)$.

Proof of fact 4. This proof is given by induction on $|t_1| + |t_2|$. As t_1 and t_2 are different, one of the following must be the case:

(1) $A^1 \neq A^2$. Then there is an $a \in Act$ such that $a \in A^1 - A^2$ or $a \in A^2 - A^1$. We only consider the first case. Define $\sigma(x) = \delta$ for all $x \in V$. If $a \in A^1 - A^2$, then there is a completed trace $a \star s \in [\![\sigma(t_1)]\!]_{\mathrm{CT}}$ for some $s \in Act^\star$ and for any $s' \in Act^\star$, $a \star s' \notin [\![\sigma(t_2)]\!]_{\mathrm{CT}}$. Hence, $\sigma(t_1) \neq_{\mathrm{CT}} \sigma(t_2)$.

(2) If $A^1 = A^2$ then it can be that for some $a \in A^1$, t_a^1 and t_a^2 are different. By induction there is a substitution σ such that for all $x \in V$, $\sigma(x) = \delta$ or $\sigma(x) = b^m : \delta$ and $\sigma(t_a^1) \neq_{\mathrm{CT}} \sigma(t_a^2)$. Hence, there is some $s \in [\![\sigma(t_a^1)]\!]_{\mathrm{CT}} - [\![\sigma(t_a^2)]\!]_{\mathrm{CT}}$ or $s \in [\![\sigma(t_a^2)]\!]_{\mathrm{CT}} - [\![\sigma(t_a^1)]\!]_{\mathrm{CT}}$. Again we only consider the first case. It is obvious that $a \star s \in [\![\sigma(t_1)]\!]_{\mathrm{CT}}$. We now show that $a \star s \notin [\![\sigma(t_2)]\!]_{\mathrm{CT}}$. Note that this immediately implies $\sigma(t_1) \neq_{\mathrm{CT}} \sigma(t_2)$. By fact 3 we know that $|s| \neq m$. As for all x in $\vec{y}_a^2$ or in $\vec{x}_j$ ($j \in J_a^2$), $\sigma(x) = \delta$ or $\sigma(x) = a^m : \delta$, $s \notin [\![\sigma(\vec{y}_a^2)]\!]_{\mathrm{CT}}$ and $s \notin [\![\sigma(\vec{x}_j)]\!]_{\mathrm{CT}}$. As already stated, $s \notin [\![\sigma(t_a^2)]\!]_{\mathrm{CT}}$. Hence, $a \star s \notin [\![\sigma(t_2)]\!]_{\mathrm{CT}}$.

(3) If $A^1 = A^2$, then it can be that for some $a \in A^1$, $\vec{y}_a^1$ and $\vec{y}_a^2$ do not contain the same variables. This means that there is a variable $x \in \vec{y}_a^1$ such that $x \notin \vec{y}_a^2$ or vice versa. It is sufficient to consider only the first case. Define a substitution σ by:

$$\sigma(y) = \begin{cases} b^m : \delta & \text{if } x = y, \\ \delta & \text{otherwise.} \end{cases}$$

Clearly, $a \star b^m \in [\![\sigma(t_1)]\!]_{\mathrm{CT}}$. We show that $a \star b^m \notin [\![\sigma(t_2)]\!]_{\mathrm{CT}}$. By definition of σ, $b^m \notin [\![\sigma(\vec{y}_a^2)]\!]_{\mathrm{CT}}$. Also, $b^m \notin [\![\sigma(t_a^2)]\!]_{\mathrm{CT}}$ because, by fact 3, for no $s \in [\![\sigma(t_a^2)]\!]_{\mathrm{CT}}$: $|s| = m$. By condition (c) of a CT normal form $a \star b^m \notin [\![\sigma(\sum_{j \in J_a^2} a : \vec{x}_j)]\!]_{\mathrm{CT}}$. So we may conclude $a \star b^m \notin [\![\sigma(t_2)]\!]$.

(4) Assume none of the three cases above apply. Then it must be the case that for some $a \in A^1 (= A^2)$ there is a $j_1 \in J_a^1$ such that for each $k_2 \in J_a^2$, $\vec{x}_{j_1}$ and $\vec{x}_{k_2}$ do not contain the same variables, or there is a $j_2 \in J_a^2$ such that for each $k_1 \in J_a^1$, $\vec{x}_{j_2}$ and $\vec{x}_{k_1}$ do not contain the same variables. Consider all $\vec{x}_{j_1}$ and $\vec{x}_{j_2}$ with the above mentioned property and select one, say $\vec{x}_j$ such that $|\vec{x}_j|$ is minimal. For symmetry we may assume that $j \in J_a^1$. Define σ by:

$$\sigma(x) = \begin{cases} \delta & \text{if } x \text{ in } \vec{x}_j, \\ b^m : \delta & \text{otherwise.} \end{cases}$$

Note that $a \in [\![\sigma(t_1)]\!]_{\mathrm{CT}}$. We show that $a \notin [\![\sigma(t_2)]\!]_{\mathrm{CT}}$. From the definition of completed traces it follows that if $a \in [\![\sigma(t_2)]\!]_{\mathrm{CT}}$ then

> for each $k \in J_a^2$, $[\![\sigma(\vec{x}_k)]\!]_{\mathrm{CT}} = \emptyset$ or
> t_a^2 has no initial actions and $[\![\sigma(\vec{y}_a^2)]\!]_{\mathrm{CT}} = \emptyset$.

But neither of these holds. Consider some $\vec{x}_k$ ($k \in J_a^2$). As $\vec{x}_j$ is minimal, it follows by condition (e) that there is some x in $\vec{x}_k$, but x is not in $\vec{x}_j$. Hence $\sigma(x) = b^m : \delta$ and thus $[\![\sigma(\vec{x}_k)]\!]_{\mathrm{CT}} \neq \emptyset$.

By condition (d), either t_a^1 has initial variables or y_a^1 contains a variable which is not present in $\vec{x}_j$. Because none of the cases (1), (2), (3) above applied, t_a^2 and t_a^1 have the same initial actions and y_a^1 and y_a^2 contain the same variables. Hence, $[\![\sigma(t_a^2 + \vec{y}_a^2)]\!]_{\mathrm{CT}} \neq \emptyset$. So we may conclude that $a \notin [\![\sigma(t_2)]\!]_{\mathrm{CT}}$.

□

With these facts, it is straightforward to finish the proof. Suppose $t_1, t_2 \in \mathbb{T}(\mathrm{BCCSP})$ and $\sigma(t_1) =_{\mathrm{CT}} \sigma(t_2)$ for each closed substitution σ. By fact 1 there are CT-normal forms u_1, u_2 and sequences of variables $\vec{y}_1$, $\vec{y}_2$ such that ($l = 1, 2$):

$$\mathrm{CT} \vdash t_l = u_l + \vec{y}_l.$$

There are three possible cases that must be considered.

1. u_1 and u_2 are different. By fact 4 there is a trace $s \in Act^\star$ with length $|s| \neq m$ ($m > \max(|t_1|, |t_2|)$) such that $s \in [\![\sigma(u_1)]\!]_{\mathrm{CT}}$ and $s \notin [\![\sigma(u_2)]\!]_{\mathrm{CT}}$ or vice versa for a closed substitution σ. For symmetry we only consider the first case. Hence $s \in [\![\sigma(u_1 + \vec{y}_1)]\!]_{\mathrm{CT}}$. By fact 4 we also know that for each $x \in V$, $\sigma(x) = \delta$ or $\sigma(x) = b^m : \delta$. Hence, $s \notin [\![\sigma(u_2 + \vec{y}_2)]\!]_{\mathrm{CT}}$ and therefore $\sigma(t_1) \neq_{\mathrm{CT}} \sigma(t_2)$ which contradicts the assumption.

2. There is a variable x in $\vec{y}_1$ that is not available in $\vec{y}_2$ (or vice versa). Define a substitution σ by:

$$\sigma(y) = \begin{cases} b^m : \delta & \text{if } y = x, \\ \delta & \text{otherwise.} \end{cases}$$

   Hence, $b^m \in [\![\sigma(u_1 + \vec{y}_1)]\!]_{\mathrm{CT}}$. As for each $s \in [\![\sigma(u_2)]\!]_{\mathrm{CT}}$: $|s| \neq m$ and x does not appear in $\vec{y}_2$, $s \notin [\![\sigma(u_2 + \vec{y}_2)]\!]_{\mathrm{CT}}$. Hence, $\sigma(t_1) \neq_{\mathrm{CT}} \sigma(t_2)$. Contradiction.

3. Suppose the cases above do not apply. Then by fact 2:

$$\mathrm{B} \vdash u_1 + \vec{y}_1 = u_2 + \vec{y}_2.$$

   Hence, $\mathrm{CT} \vdash t_1 = t_2$.

□

2.4.4 The trace axioms

Again we do not use the inverted substitution technique, although we do not know whether it can not be applied. In this case the 'standard' technique seems to be more convenient to use. Therefore, we have to give the trace semantics explicitly. In trace semantics each process is characterised by its set of prefix closed traces:

Definition 2.4.4.1. The interpretation $[\![.]\!]_{\mathrm{T}} : T(\mathrm{BCCSP}) \to 2^{Act^\star}$ is defined as follows:

$$[\![\delta]\!]_{\mathrm{T}} = \emptyset,$$
$$[\![a : t]\!]_{\mathrm{T}} = \{a \star \sigma \mid \sigma \in [\![t]\!]_{\mathrm{T}}\} \cup \{a\},$$
$$[\![t_1 + t_2]\!]_{\mathrm{T}} = [\![t_1]\!]_{\mathrm{T}} \cup [\![t_2]\!]_{\mathrm{T}}.$$

We say that $t_1, t_2 \in T(\mathrm{BCCSP})$ are *trace equivalent*, notation $t_1 =_{\mathrm{T}} t_2$, iff $[\![t_1]\!]_{\mathrm{T}} = [\![t_2]\!]_{\mathrm{T}}$.

Lemma 2.4.4.2 *(Soundness).* *Let $t_1, t_2 \in T(\mathrm{BCCSP})$:*

$$\mathrm{T} \vdash t_1 = t_2 \quad \Rightarrow \quad t_1 =_{\mathrm{T}} t_2.$$

Proof. Straightforward using the definitions. □

For trace semantics we need two actions in order to prove T ω-complete. If $|Act| = 1$ then the following axiom is valid:

$$x + a : x = a : x.$$

This can easily be seen by proving $\mathrm{T} \vdash t + a : t = a : t$ for all $t \in T(\mathrm{BCCSP})$ with induction on t if $|Act| = 1$. The axiom $x + a : x = a : x$ is in general not derivable from T, because instantiating x with $b : \delta$ yields $b : \delta + a : b : \delta \neq_{\mathrm{T}} a : b : \delta$ where $a, b \in Act$ are two different actions. In the next theorem we show that if $|Act| \geq 2$ then the axiom set T is ω-complete. First we define the notion of a syntactic summand. This notion is only used in this section.

Definition 2.4.4.3. Let $t, u \in \mathbb{T}(\mathrm{BCCSP})$. t is a *syntactic summand* of u, notation $t \sqsubseteq u$ iff:

- $t \equiv a : t'$ and $u \equiv a : t'$ for some $t' \in \mathbb{T}(\mathrm{BCCSP})$ or,
- $u \equiv u_1 + u_2$ and $t \sqsubseteq u_1$ or $t \sqsubseteq u_2$.

Lemma 2.4.4.3. *Let $t_1, t_2 \in \mathbb{T}(\mathrm{BCCSP})$. If for each syntactic summand $u \in \mathbb{T}(\mathrm{BCCSP})$,*

$$u \sqsubseteq t_1 \quad \Leftrightarrow \quad u \sqsubseteq t_2$$

then $B \vdash t_1 = t_2$.

Proof. Straightforward. □

The next theorem says that T is ω-complete if two actions are available.

Theorem 2.4.4.4. *If $|Act| \geq 2$, then for each $t_1, t_2 \in \mathbb{T}(\mathrm{BCCSP})$, we have that:*

$$\forall \sigma : V \to T(\mathrm{BCCSP}) : \ \sigma(t_1) =_{\mathrm{T}} \sigma(t_2) \quad \Rightarrow \quad \mathrm{T} \vdash t_1 = t_2.$$

Proof. We use the abbreviation $a_1 \star ... \star a_n : t$ with $a_1 \star ... \star a_n \in Act^\star$ for $a_1 : ... : a_n : t$. For $s \in Act^\star$, we define $|s|$ to be $|s : \delta|$, i.e. the length of trace s. For traces $s_1, s_2 \in Act^\star$ we write $s_1 \leq s_2$ if for some $r \in Act^\star$, $s_1 \star r = s_2$ or $s_1 = s_2$. In this case s_1 is a *prefix* of s_2.

First we define a T-*normal form*, which plays a crucial role in this proof. A term $t \in \mathbb{T}(\mathrm{BCCSP})$ is a T-normal form if

$$t \equiv \sum_{i \in I} s_i : \delta + \sum_{i \in J} s_i : x_i$$

with $s_i \in Act^\star$ $(i \in I \cup J)$, satisfying:

(1) for each s_j $(j \in I \cup J)$ with $|s_j| > 1$, there is a $i \in I$ such that $s_i \star a = s_j$ for some $a \in Act$.

(2) for each s_j $(j \in J)$ with $|s_j| > 0$, there is a $i \in I$ such that $s_j = s_i$.

Fact 1. Let $t \in \mathbb{T}(\mathrm{BCCSP})$. Then there is a T-normal form t' such that:

$$\mathrm{T} \vdash t = t'.$$

Proof of fact 1. Straightforward with induction on t. □

Fact 2. Let t and t' be two T-normal forms such that for some u, $u \sqsubseteq t$, $u \not\sqsubseteq t'$ or vice versa. Then there is a substitution $\sigma : V \to T(\mathrm{BCCSP})$ such that:

$$\sigma(t) \neq_{\mathrm{T}} \sigma(t').$$

Proof of fact 2. For symmetry it is sufficient to consider only the case where $u \sqsubseteq t$ and $u \not\sqsubseteq t'$. We can distinguish between:

(1) $u \equiv s : \delta$ with $s \in Act^{\star}$. Define $\sigma(x) = \delta$ for all $x \in V$. Note that $s \in [\![\sigma(t)]\!]_{\mathrm{T}}$. Moreover, it holds that $s \in [\![\sigma(t')]\!]_{\mathrm{T}}$ iff $s : \delta \sqsubseteq t'$; conditions (1) and (2) are required to prove this. Hence, as $s : \delta \not\sqsubseteq t'$, $s \notin [\![\sigma(t')]\!]_{\mathrm{T}}$.

(2) $u \equiv s : x$ for some $x \in V$ and $s \in Act^{\star}$. Let m be a natural number such that $m > \max(|t|, |t'|)$. Define $\sigma(x) = a^m : b : \delta$ where $a, b \in Act$ are two different actions and $\sigma(y) = \delta$ if $y \not\equiv x$. Clearly, $s \star a^m \star b \in [\![\sigma(t)]\!]_{\mathrm{T}}$. We show that $s \star a^m \star b \notin [\![\sigma(t')]\!]_{\mathrm{T}}$. Therefore we write $t' \equiv \sum_{i \in I} s_i : \delta + \sum_{i \in J} s_i : y_i$ in the following way:

$$\sum_{i \in I} s_i : \delta + \sum_{i \in K_1} s_i : y_i + \sum_{i \in K_2} s_i : x + \sum_{i \in K_3} s_i : x + \sum_{i \in K_4} s_i : x$$

where

$$\begin{aligned} K_1 &= \{i \mid i \in J \text{ and } y_i \not\equiv x\}, \\ K_2 &= \{i \mid i \in J,\ y_i \equiv x \text{ and } |s_i| < |s|\}, \\ K_3 &= \{i \mid i \in J,\ y_i \equiv x \text{ and } |s_i| = |s|\}, \\ K_4 &= \{i \mid i \in J,\ y_i \equiv x \text{ and } |s_i| > |s|\}. \end{aligned}$$

Note that $J = K_1 \cup K_2 \cup K_3 \cup K_4$. We show that $s \star a^m \star b$ cannot originate from any of these components. We deal with all five cases separately:

(a) For any $r \in [\![\sum_{i \in I} s_i : \delta]\!]_{\mathrm{T}}$, $|r| < m$ and therefore $r \neq s \star a^m \star b$.

(b) For any $r \in [\![\sum_{i \in K_1} s_i : \sigma(y_i)]\!]_{\mathrm{T}}$, $|r| < m$ because $\sigma(y_i) = \delta$. Hence, $r \neq s \star a^m \star b$.

(c) For any $r \in [\![\sum_{i \in K_2} s_i : \sigma(x)]\!]_{\mathrm{T}}$, $|r| \leq |s_i| + m + 1 < |s| + m + 1 = |s \star a^m \star b|$. Hence, $r \neq s \star a^m \star b$.

(d) For any $r \in [\![\sum_{i \in K_3} s_i : \sigma(x)]\!]_{\mathrm{T}}$, $r \leq s_i \star a^m \star b$ for some $i \in K_3$. If $|r| < |s| + m + 1$, clearly, $r \neq s \star a^m \star b$. If $|r| = |s| + m + 1$, then $r = s_i \star a^m \star b$. As $s : x \not\sqsubseteq t'$, $s_i \neq s$. Therefore $r \neq s \star a^m \star b$.

(e) Let for some $r \in Act^\star$, $r[i]$ be the i^{th} symbol in r. For any $r \in [\![\sum_{i \in K_4} s_i : \sigma(x)]\!]_{\mathrm{T}}$, $r \leq s_i \star a^m \star b$ for some $i \in K_4$. If $|r| \leq |s| + m$, then clearly $r \neq s \star a^m \star b$. If $|r| > |s| + m$, consider $r[|s| + m + 1]$. As $|s_i \star a^m \star b| > |s \star a^m \star b| > |s_i|$, $r[|s| + m + 1] = a$. But, $s \star a^m \star b[|s| + m + 1] = b$. Hence, if $|r| > |s| + m$, it also holds that $r \neq s \star a^m \star b$.

This finishes the proof of the second fact. □

Using both facts it follows almost immediately that T is ω-complete with respect to $=_{\mathrm{T}}$. Suppose $t, t' \in \mathbb{T}(\mathrm{BCCSP})$ such that for each substitution $\sigma : V \to T(\mathrm{BCCSP})$, it holds that $\sigma(t) =_{\mathrm{T}} \sigma(t')$. Both t and t' are provably equal to T-normal forms u and u' (fact 1). If u and u' have different syntactic summands, then by the second fact $\rho(u) \neq_{\mathrm{T}} \rho(u')$ for some substitution $\rho : V \to T(\mathrm{BCCSP})$. This is a contradiction. Hence, by lemma 2.4.4, $B \vdash u = u'$ and therefore:

$$\mathrm{T} \vdash t = u = u' = t'.$$

□

2.5 Extensions with the parallel operator

We extend the signature BCCSP with operators for parallelism.

2.5.1 Interleaving without communication

First, we study BCCSP with the merge and the leftmerge, but without communication. The resulting signature is called $\mathrm{BCCSP}_{\lfloor\!\lfloor}$. We study $\mathrm{BCCSP}_{\lfloor\!\lfloor}$ in the setting of bisimulation where $|Act| = \infty$. The axioms in the two top boxes of table 2.3 are complete. This follows immediately from the completeness of the axiom set B for BCCSP because any closed term over the signature $\mathrm{BCCSP}_{\lfloor\!\lfloor}$ can be rewritten to a term over the signature BCCSP.

In order to have an ω-complete set of axioms, we add two new axioms (see the lower squares of table 2.3). These axioms are derivable for all closed instances. Therefore they are valid in bisimulation semantics. The complete set of axioms in table 2.3 is called $\mathrm{B}_{\lfloor\!\lfloor}$. The following theorem states the ω-completeness of $\mathrm{B}_{\lfloor\!\lfloor}$.

$x + y = y + x$	$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x$
$(x + y) + z = x + (y + z)$	$\delta \mathbin{\lfloor\!\lfloor} x = \delta$
$x + x = x$	$a : x \mathbin{\lfloor\!\lfloor} y = a : (x \parallel y)$
$x + \delta = x$	$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$
$x \mathbin{\lfloor\!\lfloor} \delta = x$	$x \mathbin{\lfloor\!\lfloor} (y \parallel z) = (x \mathbin{\lfloor\!\lfloor} y) \mathbin{\lfloor\!\lfloor} z$

Table 2.3: The axioms for BCCSP with the leftmerge

Theorem 2.5.1.1. *The set of axioms in table* 2.3 *is ω-complete if Act contains an infinite number of actions.*

Proof. We use the technique presented in section 2.3. Suppose two terms $t, t' \in \mathbb{T}(\mathrm{BCCSP}_{\between})$ are given. Define $\rho : V \to T(\mathrm{BCCSP}_{\between})$ by $\rho(x) = a_x : \delta$ where a_x is a unique action for each $x \in V$ and a_x does neither occur in t nor in t'. Define $R : T(\mathrm{BCCSP}_{\between}) \to \mathbb{T}(\mathrm{BCCSP}_{\between})$ as follows:

$R(\delta) = \delta$,
$R(a : t) = a : R(t)$ where $a \neq a_x$ for all $x \in V$,
$R(a_x : t) = x \between R(t)$,
$R(t + u) = R(t) + R(u)$,
$R(t \parallel u) = R(t) \parallel R(u)$,
$R(t \between u) = R(t) \between R(u)$.

In order to show the axioms in table 2.3 ω-complete we must check properties (2.1), (2.2) and (2.3) of theorem 2.3.1.

(2.1) We show that $\mathrm{B}_{\between} \vdash R(\rho(u)) = u$ with induction on $u \in \mathbb{T}(\mathrm{BCCSP}_{\between})$, provided u does not contain actions of the form a_x.
$R(\rho(x)) = x \between \delta = x$,
$R(\rho(\delta)) = \delta$,
$R(\rho(t + u)) = R(\rho(t)) + R(\rho(u)) = t + u$,
$R(\rho(a : t)) = R(a : \rho(t)) =^* a : R(\rho(t)) = a : t$.
$=^*$ follows from the fact that $a \neq a_x$ for all $x \in V$.

(2.2) For the $+$-operator the proof is straightforward: $\mathrm{B}_{\between} \cup \{R(t_i) = R(u_i) \mid i = 1, 2\} \vdash R(t_1 + t_2) = R(t_1) + R(t_2) = R(u_1) + R(u_2) = R(u_1 + u_2)$. The function names $\between$ and $\parallel$ can be dealt with in the same way. The action prefix case is slightly more complicated. $R(t_1) = R(u_1) \vdash R(a : t_1) = a : R(t_1) = a : R(u_1) = R(a : u_1)$ if $a \neq a_x$ for all $x \in V$. If $a = a_x$ for some $x \in V$, then $R(t_1) = R(u_1) \vdash R(a_x : t_1) = x \between R(t_1) = x \between R(u_2) = R(a_x : u_1)$.

(2.3) It is straightforward to check the axioms that do not explicitly refer to actions. So we only check the axiom $a : x \between y = a : (x \parallel y)$. Let $\sigma : V \to T(\Sigma)$ be defined such that $\sigma(x) = t$ and $\sigma(y) = u$. $\mathrm{B}_{\between} \vdash R(a : t \between u) = a : R(t) \between R(u) = a : (R(t) \parallel R(u)) = R(a : (t \parallel u))$ if $a \neq a_x$ for all $x \in V$. In the other case $\mathrm{B}_{\between} \vdash R(a_x : t \between u) = (x \between R(t)) \between R(u) = x \between (R(t) \parallel R(u)) = R(a_x : (t \parallel u))$.

□

In many cases it is easy to show the ω-completeness of the axioms of new features introduced in $\mathrm{BCCSP}_{\between}$. As examples we introduce the silent step τ into $\mathrm{BCCSP}_{\between}$ and we consider $\mathrm{BCCSP}_{\between}$ in trace semantics.

Example 2.5.1.2. We add a constant τ (*the silent step* or *internal move*) to $\mathrm{BCCSP}_{\parallel}$. The new signature is called $\mathrm{BCCSP}^{\tau}_{\parallel}$. The internal step has been axiomatised in different ways. In [10] τ is characterised by three τ-laws. This characterisation is often called *weak bisimulation*.

$$\begin{aligned} a:\tau:x &= a:x, \\ \tau:x+x &= \tau:x, \\ a:(\tau:x+y) &= a:(\tau:x+y)+a:x. \end{aligned}$$

If one adds these laws to $\mathrm{B}_{\parallel}$, obtaining $\mathrm{B}^{\tau}_{\parallel}$, we have to add the following two axioms in order to make $\mathrm{B}^{\tau}_{\parallel}$ ω-complete. Axioms of this form already appeared in [7].

$$\begin{aligned} z \between \tau:x &= z \between x, \\ z \between (\tau:x+y) &= z \between (\tau:x+y)+z \between x. \end{aligned}$$

Both new axioms are derivable for all closed instances, and therefore valid in any model for $\mathrm{B}^{\tau}_{\parallel}$.

In [5] τ is characterised by the single equation:

$$a:(\tau:(x+y)+x) = a:(x+y).$$

This variant is called *branching bisimulation*. The set $\mathrm{B}_{\parallel}$, together with this axiom is called $\mathrm{B}^{b}_{\parallel}$. The single axiom:

$$z \between (\tau:(x+y)+x) = z \between (x+y)$$

suffices to make $\mathrm{B}^{b}_{\parallel}$ ω-complete. This axiom is derivable for all closed instances, and therefore it holds in any model for $\mathrm{B}^{b}_{\parallel}$.

We do not give the ω-completeness proofs as they can easily be given along the lines of the proof of theorem 2.5.1.1. In fact it suffices to only check condition (3) for the new axioms, because conditions (1) and (2) are provable in exactly the same way.

Example 2.5.1.3. Here we study the ω-completeness of $\mathrm{BCCSP}_{\parallel}$ in trace semantics. As any term over the signature $\mathrm{BCCSP}_{\parallel}$ can be rewritten to a term over the signature BCCSP by the axioms in $\mathrm{B}_{\parallel}$, and T is complete for the signature BCCSP in trace semantics, $\mathrm{B}_{\parallel} \cup \mathrm{T}$ is complete for $\mathrm{BCCSP}_{\parallel}$ in trace semantics. For ω-completeness we must add the equation:

$$x \between y + x \between z = x \between (y+z),$$

which is derivable from $\mathrm{B}_{\parallel} \cup \mathrm{T}$ for all its closed instances. The proof of this fact follows the lines of the proof of theorem 2.5.1.1.

2.5.2 Interleaving with communication

In this section the signature BCCSP is extended with the merge, the leftmerge and the *communication merge* (|). The signature obtained in this way is called $\mathrm{BCCSP}_{|}$. Its properties are described by the axioms in table 2.4 which are taken from [2]. In order to represent communication, we have an operator | on actions. The actions that we consider are those that are freely generated using *Act*, | and the commutativity and associativity of |. The axioms in the upper two squares

$x + y = y + x$	
$(x + y) + z = x + (y + z)$	
$x + x = x$	
$x + \delta = x$	
$x \parallel y = x \between y + y \between x + x \mid y$	$x \mid y = y \mid x$
$a : x \between y = a : (x \parallel y)$	$a : x \mid b : y = (a \mid b) : (x \parallel y)$
$\delta \between x = \delta$	$\delta \mid x = \delta$
$(x + y) \between z = x \between z + y \between z$	$(x + y) \mid z = x \mid z + y \mid z$
$(x \between y) \between z = x \between (y \parallel z)$	$(x \mid y) \mid z = x \mid (y \mid z)$
$x \between \delta = x$	$x \mid (y \between z) = (x \mid y) \between z$

Table 2.4: The axioms for $\mathrm{BCCSP}_{|}$

of table 2.4 combined with the condition that | on actions is commutative and associative, are already complete for $\mathrm{BCCSP}_{|}$-terms in the bisimulation model. This can again easily be seen by the fact that any term over the signature $\mathrm{BCCSP}_{|}$ can be rewritten to a term over BCCSP. For BCCSP the four axioms in the left upper corner of table 2.4 are complete in the bisimulation model. The axioms in the lower squares are necessary for an ω-complete axiomatisation. We call the axiom system in table 2.4 $\mathrm{B}_{|}$.

Example 2.5.2.1. The following facts are derivable from $\mathrm{B}_{|}$. We leave the proofs to the reader.

$x \parallel y = y \parallel x$,
$(x \parallel y) \parallel z = x \parallel (y \parallel z)$,
$(a_1 \mid ... \mid (a_i \mid a_{i+1}) \mid ... \mid a_n) : x = (a_1 \mid ... \mid (a_{i+1} \mid a_i) \mid ... \mid a_n) : x$,
$(a_1 \mid ... \mid (a_i(a_{i+1} \mid a_{i+2})) \mid ... \mid a_n) : x =$
$\quad (a_1 \mid ... \mid ((a_i \mid a_{i+1}) \mid a_{i+2}) \mid ... \mid a_n) : x$.

The last two identities show that it is not necessary to include axioms for the commutativity and the associativity of | on actions in $\mathrm{B}_{|}$.

Theorem 2.5.2.2. $\mathrm{B}_{|}$ *is ω-complete if Act contains an infinite number of actions.*

Proof. This proof has the same structure as the proof of theorem 2.5.1.1. We only give the non-trivial steps. Suppose two terms $t, t' \in \mathbb{T}(\mathrm{BCCSP}_{|})$ are given.

Define $\rho : V \to T(\mathrm{BCCSP}_|)$ as follows:

$$\rho(x) = a_x : \delta$$

where a_x is unique for each $x \in V$ and does not occur in t or t'. We define $R : T(\mathrm{BCCSP}_|) \to \mathbb{T}(\mathrm{BCCSP}_|)$ by:

$R(\delta) = \delta$,
$R((a_1 \mid ... \mid a_n) : t) = (a_1 \mid ... \mid a_n) : R(t)$
  if $a_i \neq a_x$ for $1 \leq i \leq n$ and $x \in V$,
$R(a_x : t) = x \between R(t)$,
$R((a_x \mid a_1 \mid ... \mid a_n) : t) = x \mid R((a_1 \mid ... \mid a_n) : t)$ for $n \geq 1$,
$R((a_1 \mid a_2 \mid ... \mid a_n) : t) = R(a_2 \mid ... \mid a_n \mid a_1) : t)$
  for $n \geq 2$ provided $a_1 \neq a_x$ for all $x \in V$,
$R(t + u) = R(t) + R(u)$,
$R(t \parallel u) = R(t) \parallel R(u)$,
$R(t \between u) = R(t) \between R(u)$,
$R(t \mid u) = R(t) \mid R(u)$.

For ρ and R we now check properties (2.1), (2.2) and (2.3) of theorem 2.3.1.

(2.1) Straightforward. In this step the axiom $x \between \delta = x$ plays an essential role.

(2.2) Straightforward for almost all cases, the only exception being the action prefix operator $(a_1 \mid ... \mid a_n) : x$ where for some a_i $(1 \leq i \leq n)$, $a_i = a_x$ with $x \in V$. Assuming that $\mathrm{B}_| \vdash R(t) = R(u)$ for $t, u \in T(\mathrm{BCCSP}_|)$, we show that $\mathrm{B}_| \vdash R((a_1 \mid ... \mid a_n) : t) = R((a_1 \mid ... \mid a_n) : u)$.

$R((a_1 \mid ... \mid a_n) : t) =$

(a) $x_j \mid (... \mid (x_{j'} \mid ((a_k \mid ... \mid a_{k'}) : R(t)))...) =$
$x_j \mid (... \mid (x_{j'} \mid ((a_k \mid ... \mid a_{k'}) : R(u)))...) =$
$R((a_1 \mid ... \mid a_n) : u)$ if there is a $1 \leq i \leq n$ such that $a_i \neq a_x$ for all $x \in V$.

(b) $x_1 \mid (... \mid (x_{n-1} \mid (x_n \between R(t)))...) =$
$x_1 \mid (... \mid (x_{n-1} \mid (x_n \between R(u)))...) = R((a_1 \mid ... \mid a_n) : u)$, otherwise.

(2.3) Only the axioms containing occurrences of the action prefix operator are non trivial to check. So we consider the axioms $a : (x \between y) = a : (x \parallel y)$ and $a : x \mid b : y = (a \mid b) : (x \parallel y)$. We start off with the first one. Let $a = (a_1 \mid ... \mid a_n)$ and let σ be a closed substitution such that $\sigma(x) = t$ and $\sigma(y) = u$. Three cases must be considered.

(a) $a_i \neq a_x$ for all $1 \leq i \leq n$ and $x \in V$.
$\mathrm{B}_| \vdash R(a : t \between u) = a : R(t) \between R(u) =$
$a : (R(t) \parallel R(u)) = R(a : (t \parallel u))$.

(b) $a_i = a_{x_i}$ for each $1 \leq i \leq n$ and $x_i \in V$.
$R((a_1 \mid ... \mid a_n) : t \between u) =$
$(x_1 \mid (... \mid (x_{n-1} \mid (x_n \between R(t)))...)) \between R(u) =$
$(((x_1 \mid ... \mid x_{n-1}) \mid x_n) \between R(t)) \between R(u) =$
$((x_1 \mid ... \mid x_{n-1}) \mid x_n) \between (R(t) \parallel R(u)) =$
$(x_1 \mid (... \mid (x_{n-1}) \mid (x_n \between (R(t) \parallel R(u))))...)) =$
$R((a_1 \mid ... \mid a_n) : (t \parallel u))$.

(c) For some $1 \leq i \leq n$, $a_i \neq a_x$ for all $x \in V$ and for some $1 \leq i \leq n$, $a_i = a_x$.
$R((a_1 \mid ... \mid a_n) : t \between u) =$
$(x_j \mid (... \mid (x_{j'} \mid ((a_{k_1} \mid ... \mid a_{k'}) : R(t)))...)) \between R(u) =$
$(x_j \mid ... \mid x_{j'}) \mid ((a_{k_1} \mid ... \mid a_{k'}) : R(t) \between R(u)) =$
$(x_j \mid ... \mid x_{j'}) \mid ((a_k \mid ... \mid a_{k'}) : (R(t) \parallel R(u))) =$
$x_j \mid (... \mid (x_{j'} \mid ((a_k \mid ... \mid a_{k'}) : (R(t) \parallel R(u))))...) =$
$R((a_1 \mid ... \mid a_n) : (t \parallel u))$.

We now check the axiom $a : x \mid b : y = (a \mid b) : (x \parallel y)$. We can distinguish 9 cases (cf. checking the axiom $a : x \between y = a : (x \parallel y)$). We do not discuss all of these, but restrict ourselves to the case where some of the actions, but not all, in a and b have the form a_x.

$R(a : t \mid b : u) =$
$(x_{j_1} \mid (... \mid (x_{j'_1} \mid (a_{k_1} \mid ... \mid a_{k'_1}) : R(t))...)) \mid$
$\quad (y_{j_2} \mid (... \mid (y_{j'_2} \mid (b_{k_2} \mid ... \mid b_{k'_2}) : R(u))...)) =$
$(x_{j_1} \mid ... \mid x_{j'_1} \mid y_{j_2} \mid ... \mid y_{j'_2}) \mid$
$\quad ((a_{k_1} \mid ... \mid a_{k'_1}) : R(t) \mid (b_{k_2} \mid ... \mid b_{k'_2}) : R(u)) =$
$(x_{j_1} \mid (... \mid (x_{j'_1} \mid (y_{j_2} \mid (... \mid (y_{j'_2} \mid ((a_{k_1} \mid ... \mid a_{k'_1}) \mid$
$\quad (b_{k_2} \mid ... \mid b_{k'_2})) : (R(t) \parallel R(u)))...)))...)) =$
$R((a_1 \mid ... \mid a_n \mid b_1 \mid ... \mid b_n) : (t \parallel u))$.

In the last step we used example 2.5.2.1 to rearrange the actions.

□

References

[1] J.A. Bergstra and J. Heering. Which data types have ω-complete initial algebra specifications? Report CS-R8958, CWI, Amsterdam, December 1989.

[2] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[3] J.A. Bergstra and J.V. Tucker. Top down design and the algebra of communicating processes. *Science of Computer Programming*, 5(2):171–199, 1984.

[4] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[5] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989.

[6] J. Heering. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science*, 43:149–167, 1986.

[7] M. Hennessy. Axiomatising finite concurrent processes. *SIAM Journal on Computing*, 17(5):997–1017, 1988.

[8] D. Kapur and D.R. Musser. Proof by consistency. *Artificial Intelligence*, 31:125–157, 1987.

[9] A. Lazrek, P. Lescanne, and J.-J. Thiel. Tools for proving inductive equalities, relative completeness, and ω-completeness. *Information and Computation*, 84:47–70, 1990.

[10] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[11] R. Milner. A complete axiomatisation for observational congruence of finite-state behaviours. Technical Report ECS-LFCS-86-8, Department of Computer Science, University of Edinburgh, 1986.

[12] F. Moller. *Axioms for concurrency.* PhD thesis, Report CST-59-89, Department of Computer Science, University of Edinburgh, 1989.

[13] V.L. Murskiĭ. The existence in three-valued logic of a closed class with finite basis, not having a finite complete system of identities. *Doklady Akademii Nauk SSSR*, 163:815–818, 1965. English translation in: *Soviet Mathematics Doklady*, 6:1020-1024, 1965.

[14] D.L. Musser. On proving inductive properties of abstract data types. In *Proceedings* 7^{th} *ACM Symposium on Principles of Programming Languages,* New York, pages 154–162. ACM, 1980.

[15] E. Paul. Proof by induction in equational theories with relations between constructors. In B. Courcelle, editor, 9^{th} *Coll. on Trees in Algebra and Programming,* Bordeaux, France, pages 211–225, London, 1984. Cambridge University Press.

3

An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence

(Jan Friso Groote & Frits Vaandrager)

This paper presents an efficient algorithm for the Relational Coarsest Partition with Stuttering problem (RCPS). The RCPS problem is closely related to the problem of deciding stuttering equivalence on finite state Kripke structures (see Browne, Clarke and Grumberg [3]), and to the problem of deciding branching bisimulation equivalence on finite state labelled transition systems (see van Glabbeek and Weijland [12]). If n is the number of states and m the number of transitions, then our algorithm has time complexity $O(n \cdot (n+m))$ and space complexity $O(n+m)$. The algorithm induces algorithms for branching bisimulation and stuttering equivalence which have the same complexity. Since for Kripke structures $m \leq n^2$, this confirms a conjecture of Browne, Clarke and Grumberg [3], that their $O(n^5)$-time algorithm for stuttering equivalence is not optimal.

3.1 Introduction

In this paper we present an efficient algorithm for the *Relational Coarsest Partition with Stuttering problem (RCPS)*. This problem is interesting because it is closely related to (1) deciding stuttering equivalence on finite Kripke structures, and (2) deciding branching bisimulation equivalence on finite labelled transition systems. Below we comment on these two problems separately.

Temporal logic model checking procedures have been successful in finding errors in relatively small network protocols and sequential circuits (for an overview see [5]). However, a serious problem for the model checking approach is the *state explosion problem*: in general, the number of states in the global state graph may grow exponentially with the number of components. In order to deal with this problem, it seems natural to hide details that do not need to be visible externally and merge those states that become indistinguishable. In the

setting of temporal logic, hiding of 'details' is achieved by making it illegal to refer to these details in temporal logic formulas. If a program is constructed in a hierarchical fashion, then state explosion may be avoided by simplifying the components before computing the global state graph [6]. Our paper deals with the question how the idea of merging indistinguishable states can be implemented in the setting of the logic CTL*.

The computation tree logic CTL* [10] is a very powerful temporal logic that combines both branching time and linear time operators. CTL [4] is a restricted subset of CTL* that permits only branching time operators. One of the operators in CTL/CTL*, the nexttime operator X, has been subject to some criticism. Lamport [15] argues that in reasoning about concurrent systems, the nexttime operator may be dangerous since it refers to the *global* next state instead of the *local* next state. If, for this reason, one decides not to use the nexttime operator, then it becomes interesting to study the equivalences on states induced by the sets of formulas CTL$-X$ and CTL$^* - X$. The idea is that two states that satisfy the same CTL$-X$/CTL$^* - X$ formulas have the same relevant properties (and maybe some irrelevant ones as well) and are therefore identified. Browne, Clarke and Grumberg [3] introduce the notion of *stuttering equivalence* on Kripke structures and prove that this equivalence characterises both the equivalence induced by CTL$-X$ and the equivalence induced by CTL$^* - X$. They show that stuttering equivalence can be decided in polynomial time but give a rather high upper bound of $O(n^5)$ for the time complexity, where n is the number of states of the Kripke structure. They conjecture the existence of a faster algorithm. The present paper confirms this conjecture: our algorithm for the RCPS problem solves stuttering equivalence in $O(m \cdot n)$ time, where m is the number of transitions in the Kripke structure. Here, like in the rest of this paper, we assume $m \geq n$. This assumption may be dropped if $m+n$ is read wherever we write m. In Kripke structures always $m \leq n^2$.

van Glabbeek and Weijland [12] introduce the notion of *branching bisimulation equivalence* on labelled transition systems. This equivalence resembles, but is finer than the *observation equivalence* of Milner [17]. They argue that, unlike observation equivalence, branching bisimulation preserves the branching structure of processes, in the sense that it preserves computations together with the potentials in all intermediate states that are passed through, even if silent moves are involved.

We show how our algorithm for RCPS can be easily transformed to an $O(m \cdot n)$ algorithm for deciding branching bisimulation equivalence ($O(m \log m + m \cdot n)$ if the set of labels is infinite or not fixed).

The structure of this paper is as follows. In section 3.2 we present the RCPS problem and in section 3.3 our algorithm to solve it. Section 3.4 describes how the algorithm can be used to decide stuttering equivalence and in section 3.5 we show how a variant of the algorithm solves the problem of deciding branching bisimulation. Section 3.6 contains some concluding remarks. We expect that, at the price of a more complicated algorithm, the efficiency of our algorithm for RCPS can be slightly improved upon by incorporating ideas from the $O(m \log n)$

algorithm of PAIGE and TARJAN [18] for the *Relational Coarsest Partition problem (RCP)*. Also in section 3.6, we compare the complexity of deciding branching bisimulation equivalence with the complexity of deciding observation equivalence.

3.2 The RCPS problem

Let S be a set. A collection $\{B_j \mid j \in J\}$ of nonempty subsets of S is called a *partition* of S if $\bigcup_{j\in J} B_j = S$ and for $i \neq j$: $B_i \cap B_j = \emptyset$. The elements of a partition are called *blocks*. If P and P' are partitions of S then P' *refines* P, and P is *coarser* than P', if any block of P' is included in a block of P. The equivalence $\sim_P$ on S induced by a partition P is defined by: $r \sim_P s$ iff $\exists B \in P:\ r \in B \wedge s \in B$.

The *Relational Coarsest Partition with Stuttering problem (RCPS)* can now be specified as follows:

Given: a nonempty, finite set S of *states*, a relation $\longrightarrow \subseteq S \times S$ of *transitions* and an *initial partition* P_0 of S.

Find: the coarsest partition P_f satisfying:

(i) P_f refines P_0;

(ii) if $r \sim_{P_f} s$ and $r \longrightarrow r'$, then there is an $n \geq 0$ and there are $s_0, ..., s_n \in S$ such that:

- $s_0 = s$;
- for all $0 \leq i < n$: $r \sim_{P_f} s_i$ and $s_i \longrightarrow s_{i+1}$;
- $r' \sim_{P_f} s_n$.

Below we show that a coarsest partition satisfying (i) and (ii) always exists; if it exists, then clearly it is unique.

3.3 The Algorithm

Next we describe our algorithm for the RCPS problem. We fix a nonempty, finite set S of states, a transition relation $\longrightarrow$ and an initial partition P_0. Let $|S| = n$ and $|\longrightarrow| = m$. For $B, B' \subseteq S$ we define the set $pos(B, B')$ as the set of states in B from which, after some initial stuttering, a state in B' can be reached:

$$pos(B,B') \;=\; \{s \in B \mid \exists n \geq 0\ \exists s_0, ..., s_n :\ s_0 = s,\ [\forall i < n:\ s_i \in B \ \wedge\ s_i \longrightarrow s_{i+1}] \text{ and } s_n \in B'\}.$$

Call B' a *splitter* of B and (B, B') a *splitting pair* iff $\emptyset \neq pos(B, B') \neq B$. Since $pos(B, B) = B$, a block can never be a splitter of itself. If P is a partition of S and B' a splitter of B, define $Ref_P(B, B')$ as the partition obtained from P

by replacing B by $pos(B, B')$ and $B - pos(B, B')$. P is *stable* with respect to a block B' if for no block B, B' is a splitter of B. P is *stable* if it is stable with respect to all its blocks. Thus the RCPS problem consists of finding the coarsest stable partition that refines P_0. Our algorithm maintains a partition P that is initially P_0. The following refinement step is repeated as long as P is not stable:

find $B, B' \in P$ such that B' is a splitter of B;
$P := Ref_P(B, B')$

Theorem 3.3.1. *The above algorithm terminates after at most $n - |P_0|$ refinement steps. The resulting partition P_f is the coarsest stable partition refining P_0.*

Proof. In order to see that the algorithm terminates, observe that after each iteration of the refinement step the number of blocks of P has increased by one. Since a partition of S can have at most n blocks, termination will occur after at most $n - |P_0|$ iterations.

Next we show that the algorithm solves the RCPS problem. By induction on the number of refinement steps we prove that any stable refinement of P_0 is also a refinement of the current partition. Clearly the statement holds initially. Suppose it is true before a refinement step that refines a partition P to a partition Q, using a splitting pair (B, B'). Let R be any stable refinement of P_0 and let C be a block of R. It is enough to show that C is included in a block from Q. By the induction hypothesis, we can assume that C is included in a block D of P. If $D \neq B$, then D is a block of Q and we are done. So suppose $D = B$. We have to show that either $C \subseteq pos(B, B')$ or $C \subseteq B - pos(B, B')$. Suppose that there are $r, s \in C$ with $r \in pos(B, B')$ and $s \notin pos(B, B')$. We derive a contradiction. There are $r_0, ..., r_n$ such that $r = r_0$, for all $i < n$: $r_i \in B \wedge r_i \longrightarrow r_{i+1}$ and $r_n \in B'$. Let $C_0, ..., C_n$ be the blocks of R such that $r_i \in C_i$. Then $C_0 = C$ and, by induction, for all $i < n$: $C_i \subseteq B$ and $C_n \subseteq B'$. Now use the fact that R is stable to construct a sequence $s_0, ..., s_m$ with $s_0 = s$, for $i < m$: $s_i \in B \wedge s_i \longrightarrow s_{i+1}$ and $s_m \in B'$. This contradicts $s \notin pos(B, B')$. Thus we have proved the induction step. □

Below we describe an implementation of our algorithm. We show how to compute in $O(m)$ time whether or not a partition is stable. The computation is organised in such a way that if the partition is not stable, a counterexample, i.e. a splitting pair (B, B'), is produced. Next we show how to compute $Ref_P(B, B')$ in $O(m)$ time. Since the number of iterations of the main loop is $O(n)$, this establishes a complexity of $O(m \cdot n)$ for the RCPS problem.

Efficient implementation of the algorithm requires some preprocessing. Let P be a partition. We call a transition $s \longrightarrow s'$ *inert* with respect to P, or *P-inert*, iff $s \sim_P s'$. If in the initial partition a set of states is strongly connected via inert transitions, then these states will be in the same block of the final partition: by definition of inert they are in the same block of the initial partition, and no refinement step will place two states from the set in a different block.

As a preprocessing step in our algorithm we look for strongly connected components with respect to inert transitions in the initial partition and 'collapse' these components to one state. Here we can use the well-known $O(m)$ algorithm for finding strongly connected components in a directed graph (see for instance AHO, HOPCROFT and ULLMAN [1]). Thus it is sufficient to solve the RCPS problem in the case where P_0 contains no cycles of inert transitions.

For $B \subseteq S$, define the set $\mathit{bottom}(B)$ of *bottom* states of B by:

$$\mathit{bottom}(B) \;=\; \{r \in B \mid \forall s:\; r \longrightarrow s \;\Rightarrow\; s \notin B\}.$$

If P is a partition, then the set $\mathit{bottom}(P)$ of *bottom* states of P is given by:

$$\mathit{bottom}(P) \;=\; \bigcup_{B \in P} \mathit{bottom}(B).$$

The following two observations play a crucial role in the implementation of our algorithm:

Lemma 3.3.2. *Let P be a refinement of P_0 and let $B, B' \in P$. Then B' is a splitter of B iff*

(1) $B \neq B'$,

(2) for some $r \in B$ and $r' \in B'$: $r \longrightarrow r'$, and

(3) there is an $s \in \mathit{bottom}(B)$ such that for no $s' \in B'$: $s \longrightarrow s'$.

Proof. "⇒" Suppose B' is a splitter of B. Then $B \neq B'$ because a block can never split itself. By definition of a splitter: $\emptyset \neq \mathit{pos}(B, B')$. Thus $r \longrightarrow r'$ for some $r \in B$ and $r' \in B'$. Suppose that for every bottom state s of B there is an $s' \in B'$ with $s \longrightarrow s'$. We derive a contradiction. Pick an element $t \in B$. Since P_0 contains no cycle of P_0-inert transitions, P does not contain a cycle of P-inert transitions. Thus there must be a path of inert transitions from t to a bottom state t' of B. Since for some $t'' \in B'$ we have $t' \longrightarrow t''$, t is in $\mathit{pos}(B, B')$. But since t was chosen arbitrarily, this means that $\mathit{pos}(B, B') = B$. This contradicts the fact that B' is a splitter of B.
"⇐" Suppose that B and B' satisfy condition (1), (2) and (3). Then B' is a splitter of B: $\mathit{pos}(B, B') \neq \emptyset$ because of (2), and $\mathit{pos}(B, B') \neq B$ because of (1) and (3). □

Lemma 3.3.3. *Let P, R be partitions such that R refines P, and P and R have the same bottom states. Let B be a block of both P and R such that P is stable with respect to B. Then R is stable with respect to B.*

Proof. Let P, R and B be as above. Pick a block B' of R. Suppose that B is a splitter for B'. We will derive a contradiction. Application of lemma 3.3.2 gives: (1) $B' \neq B$, (2) for some $r \in B'$ and $r' \in B$: $r \longrightarrow r'$, and (3) there is an

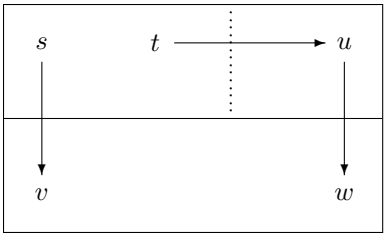


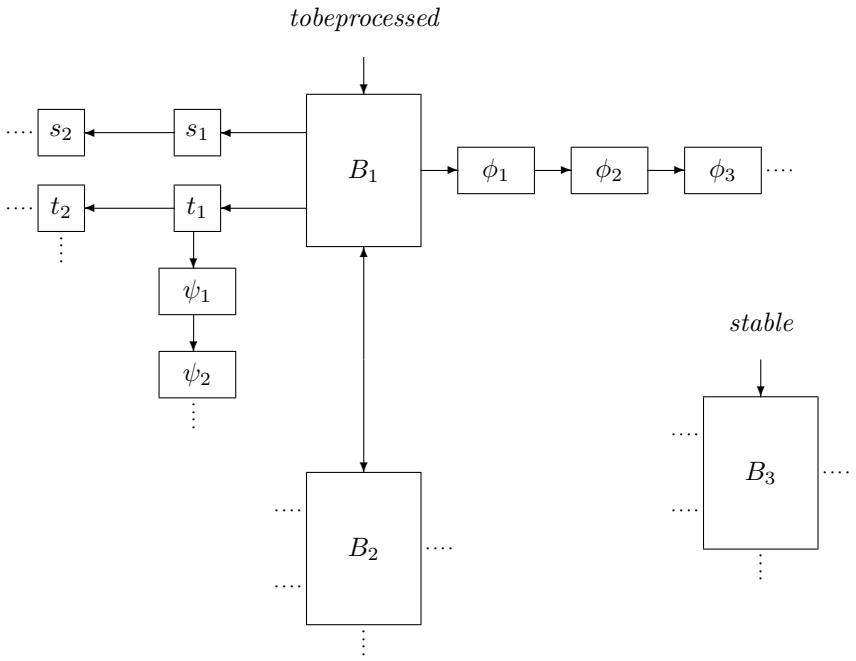
Figure 3.1: Stability is not inherited under refinement

$s \in \mathit{bottom}(B')$ such that for no $s' \in B$: $s \longrightarrow s'$. Now use that B' is included in some block C of P. Clearly $C \neq B$. Moreover we can find $r \in C$ and $r' \in B$ with $r \longrightarrow r'$. Since $\mathit{bottom}(P) = \mathit{bottom}(R)$ we have $\mathit{bottom}(B') \subset \mathit{bottom}(C)$. Thus we can find an $s \in \mathit{bottom}(C)$ such that for no $s' \in B$: $s \longrightarrow s'$. Now apply lemma 3.3.2 to conclude that B is a splitter for C, which is a contradiction. □

Remark 3.3.4. In the setting of the Relational Coarsest Partition problem, stability is inherited under refinement in general; that is, if R is a refinement of P and P is stable with respect to B, then so is R. The $O(m \log n)$ algorithm of Paige and Tarjan [18] depends crucially on this property.

In the case of the RCPS problem stability is in general not inherited under refinement; the condition in lemma 3.3.3 that P and R have the same bottom states cannot be dropped. An example is presented in figure 3.1. If $P = \{\{s,t,u\},\{v,w\}\}$ and $R = \{\{s,t\},\{u\},\{v,w\}\}$, then P is stable w.r.t. $\{v,w\}$ but R is not. As a consequence the idea behind the Paige and Tarjan [18] algorithm cannot be applied to the RCPS problem in the same way.

In the implementation of the algorithm, there is for each block, state and transition a corresponding record of type *block*, *state* resp. *transition* (see figure 3.2). We identify a block, state and transition with the record representing it. There are two doubly linked lists, *tobeprocessed* and *stable*, of blocks. A block B is in *stable* when the current partition is stable with respect to B. Otherwise B is in the list *tobeprocessed*. Initially, all blocks of P_0 are in the list *tobeprocessed* and the list *stable* is empty. Each block B contains a list of bottom states in B and a list of non-bottom states in B. We assume that whenever $s \longrightarrow s'$ for $s, s' \in B - \mathit{bottom}(B)$, s is after s' in the list of non-bottom states. Initially, the division of states in bottom states and non-bottom states, and also the ordering on the non-bottom states, can be accomplished by a standard depth first search algorithm using $O(m)$ time and space (see for instance Aho, Hopcroft and Ullman [1]). Each state contains a pointer to the block of which it is an element. Each transition contains two pointers to resp. its starting state and its target state. Each non-bottom state contains a list of the inert transitions starting in this state. Each block points to a list of the non-inert transitions that end in this block. Each state and each block has an auxiliary field *flag* of



– $B_1, B_2, B_3, ...$ are the blocks of the current partition.

– $s_1, s_2, ...$ are the bottom states in block B_1.

– $t_1, t_2, ...$ are the non-bottom states in block B_1.

– $\phi_1, \phi_2, \phi_3, ...$ are the non-inert transitions that end in block B_1.

– $\psi_1, \psi_2, ...$ are the inert transitions that start in state t_1.

Figure 3.2: The data structure of the implementation

type boolean, which is 0 initially. Moreover we use an auxiliary list BL of blocks, which is empty initially.

Next we describe how to find out in $O(m)$ time whether a partition is stable. Let B' be a block in *tobeprocessed*. Scan the list of non-inert transitions which end in B'. When some transition is visited, the flag of the starting state is raised (i.e. the value 1 is assigned to the field *flag* of the starting state). If the flag of the block to which the starting state belongs has not yet been raised, then we do so and add a copy of this block to the list BL. After having scanned all non-inert transitions ending in B', we consider the list BL. This list contains all blocks, different from B', which contain a state from which a state from B' can be reached. There is at most one reference to each block in the list. Remove the first block B from the list BL. By lemma 3.3.2, B' is a splitter of B iff there is a bottom state s of B such that for no $s' \in B'$: $s \longrightarrow s'$. So in order to find out whether B' is a splitter of B we only have to check whether the flag of all bottom states in B is raised.

Suppose that B' is a splitter of B. In this case we remove B from the linked list of blocks in which it occurs (in general this can be either the list *tobeprocessed* or the list *stable*), and insert two new blocks B_1 and B_2 in the list *tobeprocessed*. The *flag* fields of the new blocks are set to 0. All bottom states of B with a raised flag become bottom states of B_1, the other bottom states of B become bottom states of B_2. Next we scan the non-bottom states of B. If for some non-bottom state the flag is not raised and if none of the outgoing P-inert transitions leads to a state in B_1, then this state becomes a non-bottom state of B_2 (here we use the ordering on the list of non-inert states in the old partition). In this case the outgoing inert transitions of this state remain the same. Otherwise, the state is placed in B_1. It may be that certain transitions which were inert in the old partition lead to a state in B_2. In that case they have to be moved to the list of non-inert transitions which end in B_2. It may be that a state which is not a bottom state in the old partition becomes a bottom state in the new partition, just because no inert transitions are left. If, in a refinement, a non-bottom state becomes a bottom state, then (cf. lemma 3.3.3) we append the list *stable* to the list *tobeprocessed* and make *stable* empty. The non-inert transitions ending in B are distributed in the obvious way over B_1 and B_2.

If B' is not a splitter of B, or if it is a splitter and we have carried out the splitting as described above, then we consider the next first block of the list BL and check whether B' is splitter for that block, etc. When we have dealt with all blocks of BL then we know that for no block in the current partition B' is a splitter. We move B' from the list *tobeprocessed* to the list *stable*, reinitialise all flags by an additional scan of the non-inert transitions with an end state in B', and we apply the same procedure for a next block in *tobeprocessed*, etc. If *tobeprocessed* is empty then we know that the current partition is stable.

One can easily check that in $O(m)$ time we either have found a splitter and refined the partition, or we have established that the current partition is stable. Moreover the space complexity is $O(m)$. Thus we have the following theorem:

Theorem 3.3.5. *The RCPS problem can be decided in $O(m \cdot n)$ time, using $O(m)$ space.*

Remark 3.3.6. The implementation may be simplified slightly by eliminating the *stable* list. However, since the *stable* list provides a very simple way to avoid a lot of work (in our trial implementation the time performance increased with more than a factor 2), we decided to include it in the above description.

3.4 Stuttering equivalence

In this section we show how a solution of the the RCPS problem can be used to decide stuttering equivalence on finite Kripke structures. Let **AP** be a set of *atomic proposition names.*

Definition 3.4.1. A *Kripke structure* is a triple $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ where S is a set of *states*, $\longrightarrow \subseteq S \times S$ is the *transition relation* and $\mathcal{L} : S \to 2^{AP}$ is the *proposition labelling.* A Kripke structure is *finite* if the set of states is finite and for each state the set of associated proposition names is finite.

Definition 3.4.2. Let $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ be a Kripke structure. A relation $R \subseteq S \times S$ is called a *divergence blind stuttering bisimulation* if it is symmetric and whenever $r\,R\,s$ then:

(i) $\mathcal{L}(r) = \mathcal{L}(s)$ and

(ii) if $r \longrightarrow r'$ then there exist $s_0, ..., s_n \in S$ $(n \geq 0)$ such that $s = s_0$, for all $0 \leq i < n$: $s_i \longrightarrow s_{i+1} \wedge r\,R\,s_i$, and $r' R\,s_n$.

Two states $r, s \in S$ are *divergence blind stuttering equivalent*, notation $\mathcal{K} : r \leftrightarrows_{dbs} s$ or just $r \leftrightarrows_{dbs} s$, iff there exists a divergence blind stuttering bisimulation relation relating r and s.

One can easily check that divergence blind stuttering equivalence is indeed an equivalence relation.

Let $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ be a finite Kripke structure with $|S| = n$ and $|\longrightarrow| = m$. In order to determine whether two states in S are divergence blind stuttering equivalent, one can use our RCPS algorithm as follows.

The initial partition P_0 is constructed by putting all states with the same labels in the same block. Assuming that the set **AP** is finite and fixed, the initial partition can be computed in $O(n)$ time using a lexicographic sorting method [1]. If lexicographic sorting is not feasible, it can be computed in $O(n \log n)$ time. Next the RCPS algorithm is used to compute the coarsest stable partition P_f that refines P_0. This takes $O(m \cdot n)$ time. The following theorem says that partition P_f solves our problem.

Theorem 3.4.3. *Two states are in the same block of P_f exactly when they are divergence blind stuttering equivalent.*

Proof. Suppose $r, s \in B$ for some block B in P_f. Let R_f be the relation that relates two states iff they are in the same block of P_f. Clearly $r\, R_f\, s$. We show that R_f is a divergence blind stuttering bisimulation. Let $p, q \in S$ with $p\, R_f q$. As P_f refines P_0, $\mathcal{L}(p) = \mathcal{L}(q)$. Moreover, condition (ii) of definition 3.4.2 holds as it exactly coincides with the condition (ii) in the RCPS problem. Hence R_f is a divergence blind stuttering bisimulation and r and s are divergence blind stuttering equivalent. Let P_s be the partition of S induced by $\leftrightarrows_{dbs}$. By definition of divergence blind stuttering P_s refines P_0. Moreover, P_s is stable. As P_f is the coarsest stable partition refining P_0, P_s refines P_f. □

Thus the time complexity of deciding divergence blind stuttering equivalence is at most $O(m \cdot n + n \log n) = O(m \cdot n)$ (remember $m \geq n$).

In De Nicola and Vaandrager [9] it is shown that for finite Kripke structures divergence blind stuttering equivalence coincides with the equivalence induced by CTL$-X$ and CTL$^* - X$ formulas if one quantifies over all paths in the Kripke structure (the finite as well as the infinite ones). If one only quantifies over the infinite paths, then this leads to the following *stuttering equivalence* [3]:

Definition 3.4.4. Let $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ be a Kripke structure. Let s_0 be a state not in S and let p_0 be an atomic proposition such that for all s in S: $p_0 \notin \mathcal{L}(s)$. Define a Kripke structure $\mathcal{K}' = (S', \longrightarrow', \mathcal{L}')$ by:

- $S' = S \cup \{s_0\}$,
- $\longrightarrow' = \longrightarrow \cup \{(s, s_0) \mid s \in S$ has no outgoing transition or occurs on a cycle of states which all have the same label$\}$,
- $\mathcal{L}' = \mathcal{L} \cup \{(s_0, \{p_0\})\}$.

Two states $r, s \in S$ are *stuttering equivalent* if in $\mathcal{K}'$: $r \leftrightarrows_{dbs} s$ (note that this definition does not depend on the particular choice of state s_0 and atomic proposition p_0).

For finite Kripke structures the stuttering equivalence as defined above coincides with the equivalence induced by CTL$-X$ and also CTL$^* - X$ if one quantifies over infinite paths [9]. Since Browne, Clarke and Grumberg [3] proved the same result for their version of stuttering equivalence, both notions agree. For finite Kripke structures the transformation of definition 3.4.4 can be accomplished in $O(m)$ time. Thus our algorithm for RCPS can be used to decide stuttering equivalence in $O(m \cdot n)$ time.

3.5 Branching bisimulation equivalence

The RCPS algorithm cannot be used directly for deciding branching bisimulation equivalence. We have to generalise RCPS to the case where transitions have

labels.

The *Generalised Relational Coarsest Partition with Stuttering problem (GRCPS)* is given by:

Given: A nonempty, finite set S of *states*, a finite set A of *labels* containing the *silent step* τ, a relation $\longrightarrow \subseteq S \times A \times S$ of *transitions* and an *initial partition* P_0 of S.

Find: the coarsest partition P_f satisfying:

(i) P_f refines P_0,

(ii) if $r \sim_{P_f} s$ and $r \stackrel{a}{\longrightarrow} r'$, then either $a = \tau$ and $r' \sim_{P_f} s$, or there is an $n \geq 0$ and there are $s_0, ..., s_n, s'$ such that $s_0 = s$, for all $0 < i \leq n$: $[r \sim_{P_f} s_i \ \wedge \ s_{i-1} \stackrel{\tau}{\longrightarrow} s_i]$, $s_n \stackrel{a}{\longrightarrow} s'$ and $r' \sim_{P_f} s'$.

Our algorithm for the GRCPS-problem is a minor modification of the algorithm for the RCPS-problem. Therefore, we will not describe it in detail but only sketch the differences. We fix S, A, $\longrightarrow$ and P_0 and, as usual, write $|S| = n$ and $|\longrightarrow| = m$. For $B, B' \subseteq S$ and $a \in A$, the set $pos_a(B, B')$ is defined by:

$$pos_a(B, B') = \{s \in B \mid \exists n \geq 0 \ \exists s_0, ..., s_n \in B \ \exists s' \in B' : \\ s_0 = s, \ [\forall 0 < i \leq n : \ s_{i-1} \stackrel{\tau}{\longrightarrow} s_i] \text{ and } s_n \stackrel{a}{\longrightarrow} s'\}.$$

We say that B' is a *splitter of B with respect to a* iff $B \neq B'$ or $a \neq \tau$, and $\emptyset \neq pos_a(B, B') \neq B$. If P is a partition of S and B' is a splitter of B with respect to a, then $Ref_P^a(B, B')$ is the partition P where B is replaced by $pos_a(B, B')$ and $B - pos_a(B, B')$. P is *stable* with respect to a block B' if for no block B and for no action a, B' is a splitter of B w.r.t. a. P is *stable* if it is stable with respect to all its blocks.

The algorithm maintains a partition P that is initially P_0. It repeats the following step, until P is stable:

find blocks $B, B' \in P$ and a label $a \in A$
    such that B' is a splitter of B with respect to a;
$P := Ref_P^a(B, B')$.

Theorem 3.5.1. *The above algorithm for the GRCPS problem terminates after at most $n - |P_0|$ refinement steps. The resulting partition P_f is the coarsest stable partition refining P_0.*

Proof. Similar to the proof of theorem 3.3.1. □

We must now show that one can find a splitter B' with respect to some label a in time $O(m)$ or find in $O(m)$ time that no such splitter exists. Moreover, a

refinement must be carried out in $O(m)$ time. To this purpose we use the data structure of the RCPS algorithm. But now a transition $s \xrightarrow{a} s'$ is called *(P-)inert* if $s \sim_P s'$ and $a = \tau$, and a state $s \in B$ is a *bottom state* of B if $s \in B$ and there is no $s' \in B$ such that $s \xrightarrow{\tau} s'$. The data structure is initialised in the same way as for the RCPS algorithm. However, the non-inert transitions ending in a block B are grouped on label, i.e. all transitions with the same label are in subsequent records in the list. If there are non-inert transitions with a label τ ending in a block B, then they are at the beginning of the list. This facilitates adding inert transitions that become non-inert after a refinement at the beginning of the transition list. Grouping of the transitions has time complexity $O(m \log m)$ (heapsort) or $O(|A| + m)$ (bucket sort).

The following lemmas are the counterparts of theorem 3.3.1 and lemma 3.3.2. As the proofs are similar, they are omitted.

Lemma 3.5.2. *Let P be a refinement of P_0 and let $B, B' \in P$ and $a \in A$. Then B' is a splitter of B with respect to a iff*

1) *$a \neq \tau$ or $B \neq B'$,*

2) *for some $r \in B$ and $r' \in B'$: $r \xrightarrow{a} r'$, and*

3) *there is a bottom state s of B such that for no $s' \in B'$: $s \xrightarrow{a} s'$.*

Lemma 3.5.3. *Let P, R be partitions such that R refines P, and P and R have the same bottom states. Let B be a block of both P and R such that P is stable with respect to B. Then R is stable with respect to B.*

A splitter can be found in the same way as in the RCPS algorithm. Continue the following step until the list *tobeprocessed* is empty or a splitter has been found. Consider a block B from the list *tobeprocessed*. Consider subsequently all groups Φ of non-inert transitions ending in B with the same label a, set the *flag* field of the starting states of transitions in Φ and construct BL. A copy of Φ is maintained for resetting the flags. Then check stability of all blocks B' in BL with respect to B and label a and split B' if necessary. Due to lemma 3.5.2 and lemma 3.5.3 this can be performed in exactly the same way as in the RCPS case. Reset the flags of the states using the copy of Φ. If B splits itself into blocks B_1 and B_2, it is not necessary to check more transitions ending in B, as they must again be checked for B_1 and B_2. If all incoming transitions in block B have been checked, if B is not split and if there is no new bottom state, move B from *tobeprocessed* to *stable*.

Branching bisimulation is mostly defined on labelled transition systems (LTS's). The GRCPS-algorithm can be used to decide branching bisimulation on finite LTS's.

Definition 3.5.4. A *labelled transition system (LTS)* is a triple $\mathcal{L} = (S, A, \longrightarrow)$ with S a set of *states*, A a set of *labels* containing the *silent step* τ, and $\longrightarrow \subseteq S \times A \times S$ a *transition relation*. $\mathcal{L}$ is called *finite* if both S and A are finite.

Definition 3.5.5 *([12]).* Let $\mathcal{L} = (S, A, \longrightarrow)$ be an LTS. Let $\Rightarrow$ be the transitive and reflexive closure of $\xrightarrow{\tau}$. A relation $R \subseteq S \times S$ is a *branching bisimulation* iff it is symmetric and whenever $r\,R\,s$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $r'\,R\,s$, or there exist s_1, s' such that $s \Rightarrow s_1 \xrightarrow{a} s'$ and $r\,R\,s_1$ and $r'\,R\,s'$.
Two states $r, s \in S$ are *branching bisimilar*, notation $r \leftrightarrows_b s$, iff there exists a branching bisimulation relation relating r and s.

We could have strengthened this definition by requiring *all* intermediate states in $s \Rightarrow s_1$ to be related with r. The following lemma implies that this would lead to the same equivalence relation.

Lemma 3.5.6 *(cf. lemma 1.3 of [12]). Let $\mathcal{L} = (S, A, \longrightarrow)$ be an LTS and let for some $n > 0$, $r_0 \xrightarrow{\tau} r_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} r_{n-1} \xrightarrow{\tau} r_n$ be a path in $\mathcal{L}$ with $r_0 \leftrightarrows_b r_n$. Then for all $0 \leq i \leq n$: $r_0 \leftrightarrows_b r_i$.*

Theorem 3.5.7. *Let $\mathcal{L} = (S, A, \longrightarrow)$ be a finite LTS. Let P_f be the final partition obtained after applying the GRCPS algorithm on an initial partition containing only block S. Then $\sim_{P_f} = \leftrightarrows_b$.*

Proof. "$\subseteq$" Using theorem 3.5.1 it follows that $\sim_{P_f}$ is a branching bisimulation relation.
"$\supseteq$" $\leftrightarrows_b$ induces a stable partition on S (use lemma 3.5.6). As P_f is the coarsest stable partition, $\sim_{P_f} \supseteq \leftrightarrows_b$. □

So in order to compute whether two states in a finite LTS are branching bisimilar we can apply the GRCPS algorithm with as initial partition the partition containing the set of states as only block. This takes $O(m \log m + m \cdot n)$ resp. $O(|A| + m \cdot n)$ time, depending on the sorting algorithm that has been used.

3.6 Concluding remarks

Is our $O(m \cdot n)$ algorithm for the RCPS problem optimal? We do not think so. In fact we expect that our algorithm can be slightly improved upon by incorporating ideas behind the $O(m \log n)$ algorithm of Paige and Tarjan [18] for the RCP problem. Let m_i be the number of inert transitions in the initial partition and let n_i be the number of states which have an outgoing inert transition in the initial partition but are bottom states in the final partition. We expect that the following holds:

Conjecture 3.6.1. *The RCPS problem can be decided in $O(m \cdot n_i + m_i \cdot n + m \log n)$ time, using $O(m)$ space.*

As already observed in remark 3.3.4, stability is not inherited under refinement in general: problems arise when, in a refinement, a non-bottom state becomes

a bottom state. This situation can occur n_i times. The summand $m \cdot n_i$ in the expression above corresponds to the additional amount of work that has to be done to deal with these situations. At present we do not see how to avoid scanning all inert transitions when we do a refinement step. This explains the summand $m_i \cdot n$. If there are no inert transitions, then the algorithm which we conjecture is as efficient as the PAIGE and TARJAN [18] algorithm for the RCP problem. However, often m_i will be of the same order as m. In that case the order of complexity equals the one of our $O(m \cdot n)$ algorithm. For this reason, and also because the algorithm which we conjecture is rather complex (it combines the techniques of PAIGE and TARJAN [18] with the techniques of our $O(m \cdot n)$ algorithm), we decided to concentrate first on a clear exposition of the $O(m \cdot n)$ algorithm.

In a sense, branching bisimulation equivalence can be viewed as an alternative to observation equivalence. Thus it is interesting to compare the complexities of deciding these equivalences. First consider the situation where the set A of labels is fixed (so $O(m) \leq O(n^2)$). All known algorithms for deciding observation equivalence (see e.g. [2, 14]) work in two phases. First a transitive closure algorithm is used to compute the so-called double arrow relation. With a simple algorithm (see e.g. [1]) this takes $O(n^3)$ time. The result of the transitive closure is a new LTS with at most $O(n^2)$ more edges than the original LTS. Next a variant of the PAIGE and TARJAN [18] algorithm is used to decide *strong bisimulation equivalence* on the new LTS. This takes $O(n^2 \log n)$ time. The resulting complexity in this case for deciding observation equivalence is $O(n^3)$, which is the same as the complexity of our algorithm. Now there are numerous sub-cubic transitive closure algorithms in the literature (see e.g. [7] for an $O(n^{2.376})$ algorithm). These algorithms tend to be practical only for large values of n. Still we have that if the set of labels is fixed, the number of states is large and the number of transitions is of order $O(n^2)$, observation equivalence can be decided faster than branching bisimulation if one uses these sub-cubic algorithms.

However, things change if one does not fix the set of labels. Since one has to compute the double arrow relation for all labels that occur in the LTS, the complexity of computing the double arrow relation then becomes $O(m \cdot n^{2.376})$ [14] (at least, we do not know any faster solution). In that case our algorithm for branching bisimulation is more efficient.

Clearly, the issue of comparing the complexities of observation equivalence and branching bisimulation is nontrivial and the analysis above does not give very much insight into the performance of our algorithm in practical applications. Therefore, we wrote a trial implementation in Pascal and compared the performance of this implementation with the performance of AUTO [19] and Aldébaran [11], as far as we know the two fastest tools currently available for deciding observation equivalence.

The process we used for our tests was the 'scheduler' as described by MILNER [16]. This scheduler schedules k processes in succession modulo k, i.e. after process k process 1 is reactivated again. However, a process must never be

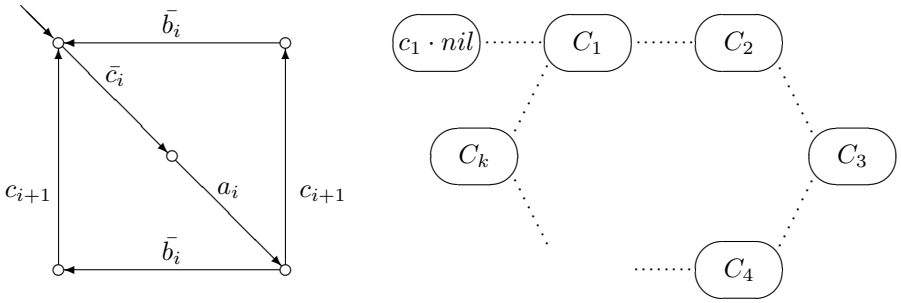


Figure 3.3: A cycler and a scheduler

reactivated before it has terminated. The scheduler is constructed of k cyclers $C_1, ..., C_k$, where cycler C_i takes care of process i. The left part of figure 3.3 shows the transition system for cycler C_i. In the right part the architecture of the scheduler is depicted. The dotted lines indicate where the cyclers synchronise. Cycler C_i first receives a signal $\bar{c}_i$ which indicates that it may start. It then activates process i via an action a_i. Next, it waits for termination of process i, indicated by $\bar{b}_i$, and in parallel, using c_{i+1}, the next cycler is informed to start. Afterwards the cycler is back in its initial state. The cycler C_i is described by:

$$C_i \;=\; \bar{c}_i \cdot a_i \cdot (\bar{b}_i | c_{i+1}) \cdot C_i.$$

The complete scheduler for k processes is described by:

$$Sch_k \;=\; (c_1 \cdot nil | C_1 | ... | C_k) \backslash c_1 ... \backslash c_k.$$

Here $c_1 \cdot nil$ is an auxiliary process that starts the first cycler.

The results of experiments with schedulers of different size are given in table 1. Here k is the number of cyclers per scheduler. The second and third column give the number of states resp. transitions of the corresponding transition system. Then we give the time necessary to calculate the bisimulation equivalence classes of the schedulers for AUTO (AU), Aldébaran (AB) and our trial implementation (BB). We give these figures not only for the case where labels a_i and $\bar{b}_i$ are both visible, but also for the case where actions $\bar{b}_i$ are hidden (i.e. renamed into τ). The figures for AUTO and our trial implementation have been obtained using a SUN 3/60 with 16 MB of memory. The figures for Aldébaran, which are taken from [11], were obtained with a 50 MB SUN 3/60. It is important to note that these figures refer only to the second phase of the algorithm where the strong bisimulation equivalence classes are computed. So the time it takes to carry out the first phase (the transitive closure) is not included. This means that, roughly speaking, the figures for Aldébaran must be multiplied by 2. The figures for AUTO refer to the time needed for both phases of the algorithm. In separate columns the number of resulting equivalence classes of both experiments are given. They are the same for branching bisimulation and observation equivalence. In the table, "-" means that no outcome was obtained due to lack of memory and "*" means that no outcome is reported in [11].

			both a_i and $\bar{b}_i$ visible				only a_i visible			
k	states	trans.	AU	AB	BB	eq.cl.	AU	AB	BB	eq.cl.
4	97	241	0.5s	0.26s	0.07s	64	0.4s	0.15s	0.02s	4
5	241	721	1.9s	0.88s	0.3s	160	1.1s	0.6s	0.07s	5
6	577	2017	8.0s	2.6s	0.9s	384	3.3s	1.9s	0.2s	6
7	1345	5377	38s	7.2s	2.5s	896	12s	6.9s	0.5s	7
8	3073	13825	201s	21s	7.7s	2048	57s	24s	1.2s	8
9	6913	34561	-	56s	23s	4608	-	80s	2.9s	9
10	15361	84481	-	160s	67s	10240	-	-	7.4s	10
11	33793	202753	-	*	214s	22528	-	-	19s	11
12	73729	479233	-	*	1254s	49152	-	-	53s	12

Table 3.1: Some test results

Our implementation improves the performance of Aldébaran and AUTO considerably, especially when a lot of τ's are around. For the space requirements this is directly reflected in the fact that, in the case where only the a_i-actions are visible, we can handle 12 cyclers on a 16 MB machine, whereas Aldébaran, on a 50 MB machine, can handle only 9 cyclers and AUTO, on a 16 MB machine, only 8. The figures about the time performance also show a considerable improvement (up to a factor 47). So in this experiment our algorithm is doing better than the usual algorithms for observation equivalence. We expect that algorithms for branching bisimulation will perform better than algorithms for weak bisimulation. However, we find it hard to draw this as a firm conclusion from the experiments, since these are influenced by many factors that are difficult to control, such as the skill of the programmer and the programming language used.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] T. Bolognesi and S.A. Smolka. Fundamental results for the verification of observational equivalence: a survey. In H. Rudin and C. West, editors, *Proceedings 7^{th} IFIP WG6.1 International Symposium on Protocol Specification, Testing, and Verification,* Zürich, Switserland, May 1987. North-Holland, 1987.

[3] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1,2):115–131, 1988.

[4] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume

131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[5] E.M. Clarke and O. Grümberg. Research on automatic verification of finite state concurrent systems. *Ann. Rev. Comput. Sci.*, 2:269–290, 1987.

[6] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings* 4^{th} *Annual Symposium on Logic in Computer Science,* Asilomar, California, pages 353–362, Washington, 1989. IEEE Computer Society Press.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the* 19^{th} *Annual ACM Symposium on Theory of Computing,* New York City, pages 1–6, 1987.

[8] R. De Nicola, U. Montanari, and F.W. Vaandrager. Back and forth bisimulations. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer-Verlag, 1990.

[9] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science,* La Roche Posay, France, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.

[10] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'Not Never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[11] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1990.

[12] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989.

[13] R.J. van Glabbeek and W.P. Weijland. Refinement in branching time semantics. Report CS-R8922, CWI, Amsterdam, 1989. Also appeared in: Proceedings AMAST Conference, May 1989, Iowa, USA, pp. 197–201.

[14] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[15] L. Lamport. What good is temporal logic? In R.E. Mason, editor, *Information Processing 83*, pages 657–668. North-Holland, 1983.

[16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[17] R. Milner. *Communication and concurrency.* Prentice Hall International, 1989.

[18] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[19] R. de Simone and D. Vergamini. Aboard AUTO. Technical Report 111, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1989.

4

Transition System Specifications with Negative Premises

(Jan Friso Groote)

In this paper the general approach to Plotkin style operational semantics of [16] is extended to Transition System Specifications (TSS's) with rules that may contain negative premises. Two problems arise: first the rules may be inconsistent, and secondly it is not obvious how a TSS determines a transition relation. We present a general method, based on the stratification technique in logic programming, to prove consistency of a set of rules and we show how a specific transition relation can be associated with a TSS in a natural way. Then a special format for the rules, the *ntyft/ntyxt*-format, is defined. It is shown that for this format three important theorems hold. The first theorem says that bisimulation is a congruence if all operators are defined using this format. The second theorem states that under certain restrictions a TSS in *ntyft*-format can be added conservatively to a TSS in pure *ntyft/ntyxt* -format. Finally, it is shown that the trace congruence for image finite processes induced by the pure *ntyft/ntyxt*-format is precisely bisimulation equivalence.

4.1 Introduction

In recent years, many process calculi, programming languages and specification languages are provided with an operational semantics in Plotkin style [29, 30]. We mention CCS [22, 24], SCCS [23], ACP [15], MEIJE [4], Esterel [9], LOTOS [18] and Ada [3].

In [16] an operational semantics in Plotkin style is defined by a TSS (Transition System Specification). Basically, a TSS consists of three components. The first component is a *signature* defining the language elements. All terms over this signature are referred to as *(process) terms* or *processes*. The second component of a TSS is a set of *actions* or *labels* representing the different activities that

process terms may do. The last component is a set of *rules* that define how processes can perform certain activities depending on the *presence* of specific actions in other processes. In [16] the possibility to perform activity based on the *absence* of actions is not considered.

But in many cases it is convenient to have this possibility. For instance, a deadlock detector $D(p)$ of a process p can naturally be specified as follows: if p can do *no* action then $D(p)$ may signal deadlock. We find deadlock detectors described in this way in [20, 28].

Deadlock detection is also used in sequencing processes. If in $p \cdot q$ (process p sequenced with q) p *cannot* do anything, q may start. See for instance [25] or [10], where it is observed that sequencing can only be defined using negative premises.

Negative conditions are also useful to describe priorities. Suppose θ is a unary operator that blocks all actions which do not have the highest priority. An operational description of $\theta(p)$ could be that it can only perform action a if it *cannot* perform any activity with higher priority. Descriptions of priorities with negative premises can be found in [6, 13, 16].

Another area where negative conditions can be fruitfully applied is the area of synchronous parallel operators. Suppose a sender wants to send data to a receiver. If the receiver is willing to accept the data, then data transfer will take place. If the receiver is *not* willing to accept the data then the sender may not be blocked and data may for instance disappear. This can conveniently be described using negative premises. Pnueli [31] defines an operator in this way. Also the *put* and *get* primitives of Bergstra [8] can be defined using negative premises.

Negative premises are also used quite a lot in timed proces algebras [11, 17, 19, 26]. Generally, they are used to avoid a time-stop, saying that if nothing can happen time can proceed, or to force maximal progress, saying that if certain actions can happen, then they must happen.

Often negative premises can be avoided. A typical example of this can be found in [12]. Using additional labels, function names and rules an operational semantics can be given with only positive premises. But then there are many auxiliary transitions that do not correspond to *positive* activity. Moreover, definitions of operational semantics become more complex than necessary. This means that an important property of operational semantics in Plotkin style, namely simplicity, is violated.

For these reasons we believe that it is useful to investigate how one can deal with negative premises in TSS's.

A format of rules that allows negative premises is the GSOS-format of Bloom, Istrail and Meyer [10]. All operators mentioned above can be defined in this format. The GSOS-format, however, is incompatible with the (pure) *tyft/tyxt*-format [16] that allows *lookahead* and no negative premises. Many useful operators definable in the *tyft/tyxt*-format cannot be defined using the GSOS format. The situation is described by the black arrows in figure 4.1. The *positive GSOS*-format is the most general format that is below both the *tyft/tyxt*-format and the

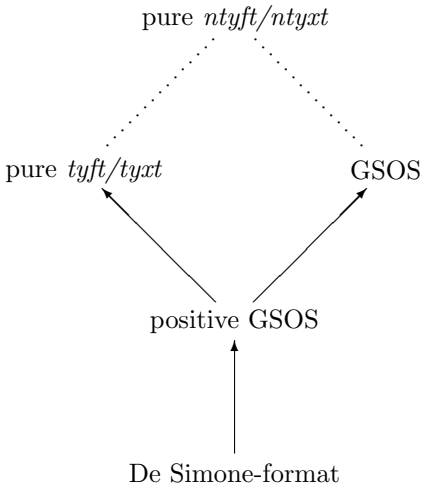


Figure 4.1: Pure *ntyft/ntyxt* extends both GSOS and pure *tyft/tyxt*

GSOS-format. Below the positive GSOS-format we find the De Simone-format [14] which was already defined by R. de Simone in 1984. The De Simone-format is powerful enough to define all the usual operators of CCS, SCCS, ACP and Meije. All formats are explained more precisely in the last section of this paper.

The natural question arises whether a format exists that is more general than both the pure *tyft/tyxt*-format and the GSOS-format. An obvious candidate for such a format is obtained by adding negative premises to the *tyft/tyxt*-format, obtaining the pure *ntyft/ntyxt*-format. The n in the name of the format is added to indicate the possible presence of negative premises. We arrive at the situation depicted by the dotted lines in figure 4.1.

Two problems arise when rules can be in pure *ntyft/ntyxt*-format:

- It is possible to give an inconsistent set of rules. This occurs if one can derive using the rules that a process can perform an action if and only if it cannot do so. In this case the rules do not define an operational semantics.

- Even if the rules are consistent, it is not immediately obvious how these rules determine an operational semantics. The normal notion of provability of transitions where the rules in a TSS are used as inference rules does not work.

We deal with the first problem by formulating a method of checking whether a transition relation is consistent. This method is based on the *local stratifications* [2, 32] that are used in logic programming. The other problem is solved by formulating an explicit definition of the transition relation.

Furthermore, general properties of the *ntyft/ntyxt*-format are studied. It is shown that bisimulation is a congruence for this format. Then, in section 4.5 we define the *sum* of two TSS's and we prove a theorem stating very general conditions under which a TSS can be added *conservatively* to another TSS.

In [16] the completed trace congruences induced by the pure *tyft/tyxt*-format and the GSOS format are characterised. It is interesting to know the impact of the more powerful testing capabilities of the pure *ntyft/ntyxt*-format. Surprisingly, it turns out that the (completed) trace congruence induced by the pure *ntyft/ntyxt*-format is exactly strong bisimulation. This is shown by a small test system that provides an alternative for the test systems of [1] and [10]. We do not need the *global testing* operators like the ones used in these articles. The combination of *copying*, *lookahead* and negative premises turns out to be powerful enough.

4.2 Transition system specifications and stratifications

This section describes a TSS as a general framework for defining an operational semantics in Plotkin style. A condition is developed that guarantees the existence of transition relations agreeing with a TSS. This condition is comparable to local stratification as used in logic programming. Next, we define which transition relation is associated with a TSS. Finally, some remarks are made about a class of TSS's which determine a transition relation in a unique way. We start off by defining the basic notations that are used throughout the paper. We assume the presence of an infinite set V of *variables* with typical elements $x, y, z....$

Definition 4.2.1. A *(single sorted) signature* is a structure $\Sigma = (F, r)$ where:

- F is a set of *function names* disjoint with V,
- $r : F \to \mathbf{N}$ is a *rank function* which gives the arity of a function name; if $f \in F$ and $r(f) = 0$ then f is called a *constant name*.

Let $W \subseteq V$ be a set of variables. The set of Σ-*terms* over W, notation $T(\Sigma, W)$, is the least set satisfying:

- $W \subseteq T(\Sigma, W)$,
- if $f \in F$ and $t_1, ..., t_{r(f)} \in T(\Sigma, W)$, then $f(t_1, ..., t_{r(f)}) \in T(\Sigma, W)$.

$T(\Sigma, \emptyset)$ is abbreviated by $T(\Sigma)$; elements from $T(\Sigma)$ are called *ground* or *closed* terms. $\mathbb{T}(\Sigma)$ is used to abbreviate $T(\Sigma, V)$, the set of *open terms*. Clearly, $T(\Sigma) \subset \mathbb{T}(\Sigma)$. $Var(t) \subseteq V$ is the set of variables in a term $t \in \mathbb{T}(\Sigma)$. A *substitution* σ is a mapping in $V \to \mathbb{T}(\Sigma)$. A substitution σ is extended to a mapping $\sigma : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ in a standard way by the following definition:

- $\sigma(f(t_1, ..., t_{r(f)})) \;=\; f(\sigma(t_1), ..., \sigma(t_{r(f)}))$ for $f \in F$ and $t_1, ..., t_{r(f)} \in \mathbb{T}(\Sigma)$.

A substitution is *ground* if it maps all variables onto ground terms.

Definition 4.2.2. A *TSS* (*Transition System Specification*) is a triple $P = (\Sigma, A, R)$ with $\Sigma = (F, r)$ a signature, A a set of labels and R a set of rules of the form:

$$\frac{\{t_k \xrightarrow{a_k} t'_k \mid k \in K\} \cup \{t_l \xrightarrow{b_l}\!\!\!\!\!\!/\ \ \mid l \in L\}}{t \xrightarrow{a} t'}$$

with K, L index sets, $t_k, t'_k, t_l, t, t' \in \mathbb{T}(\Sigma)$, $a_k, b_l, a \in A$ $(k \in K,\ l \in L)$. An expression of the form $t \xrightarrow{a} t'$ is called a (*positive*) *literal*. Here t is called the *source* and t' the *target* of the literal. $t \xrightarrow{a}\!\!\!\!\!\!/\ \ $ is called a *negative literal*. ϕ, ψ, χ are used to range over literals. The literals above the line are called the *premises* and the literal below the line is called the *conclusion*. A rule is called an *axiom* if its set of premises is empty. An axiom

$$\frac{\emptyset}{t \xrightarrow{a} t'}$$

is often written as $t \xrightarrow{a} t'$. The notions 'substitution', 'Var' and 'ground' extend to literals and rules as expected.

Note that this definition differs from the definition of a TSS in [16] because it allows an infinite number of premises and premises may now be negative. The purpose of a TSS is to define a *transition relation* $\longrightarrow \subseteq Tr(\Sigma, A) = T(\Sigma) \times A \times T(\Sigma)$. A transition relation states under what actions ground terms over the signature can evolve into one another. This expresses the operational behaviour of these terms. Elements (t, a, t') of a transition relation are written as $t \xrightarrow{a} t'$. We say that a positive literal ψ *holds* in $\longrightarrow$, notation $\longrightarrow \models \psi$, if $\psi \in \longrightarrow$. A negative literal $t \xrightarrow{a}\!\!\!\!\!\!/\ \ $ *holds* in $\longrightarrow$, notation $\longrightarrow \models t \xrightarrow{a}\!\!\!\!\!\!/\ \ $, if for no $t' \in T(\Sigma)$: $t \xrightarrow{a} t' \in \longrightarrow$.

For TSS's without negative premises the notion of a transition relation that must be associated with it is rather straightforward. All literals that can be proved by a well founded proof tree where the rules of the TSS P are used as inference rules, are in the transition relation associated with P. For TSS's with negative premises these proof trees cannot be used. It is not so obvious which transition relation should be associated with such a TSS. In [10] Bloom, Istrail and Meyer require that a transition relation *agrees with* a TSS. In terms of logic programming this means that the transition relation is a supported model of the TSS (see also definition 5.3.3, 5.3.4 and 5.3.5 in the next chapter).

Definition 4.2.3. Let $P = (\Sigma, A, R)$ be a TSS. Let $\longrightarrow \subseteq Tr(\Sigma, A)$ be a transition relation. $\longrightarrow$ *agrees with* P iff:

$$\psi \in \longrightarrow \quad \Leftrightarrow \quad \exists \frac{\{\chi_k \mid k \in K\}}{\chi} \in R \text{ and } \exists \sigma : V \to T(\Sigma) \text{ such that :}$$
$$\sigma(\chi) = \psi \text{ and } \forall k \in K : \longrightarrow \models \sigma(\chi_k).$$

Unfortunately, for a given TSS P it is not guaranteed that a transition relation that agrees with P exists and if it exists it need not be unique. We give three examples illustrating these points. The last example already occurred in [10].

Example 4.2.4. It is possible to give a TSS P such that there is no transition relation that agrees with it. Let P consist of one constant f, one label a and the rule

$$\frac{f \overset{a}{\not\longrightarrow}}{f \overset{a}{\longrightarrow} f}.$$

For any transition relation $\longrightarrow$ that agrees with P, $f \overset{a}{\longrightarrow} f \in \longrightarrow$ iff $f \overset{a}{\longrightarrow} f \notin \longrightarrow$. Clearly, such a transition relation does not exist.

Example 4.2.5. This example shows that if a transition relation that agrees with a TSS exists, it need not be unique. Take for example a TSS with the only rule:

$$\frac{f \overset{a}{\longrightarrow} f}{f \overset{a}{\longrightarrow} f}.$$

Both the empty transition relation and the transition relation $\{f \overset{a}{\longrightarrow} f\}$ agree with this TSS.

Example 4.2.6. If we only use variables in the premises, we can still have that there is no transition relation agreeing with the rules. Suppose we have a TSS which consists of constants a and δ and two unary function names f and g. Furthermore, we have exactly one label a and the following rules:

$$\frac{x \overset{a}{\longrightarrow} y \quad y \overset{a}{\longrightarrow} z}{f(x) \overset{a}{\longrightarrow} \delta},$$

$$\frac{x \overset{a}{\not\longrightarrow}}{g(x) \overset{a}{\longrightarrow} \delta},$$

$$a \overset{a}{\longrightarrow} g(f(a)).$$

No transition relation agrees with this TSS since if it would exist we would have that $f(a) \overset{a}{\longrightarrow} \delta$ is an element of this relation iff it is not.

In this section we develop a condition on TSS's which guarantees the existence of transition relations that agree with them. The idea is that a transition relation is constructed in a stepwise manner. Whenever it is assumed that some literal does not exist in a transition relation, it must be guaranteed that there is no way to derive the opposite from this assumption. It can be visualised how literals can be derived from each other in a *literal dependency graph* of a TSS $P = (\Sigma, A, R)$. In this graph it is recorded by directed edges how literals depend on each other.

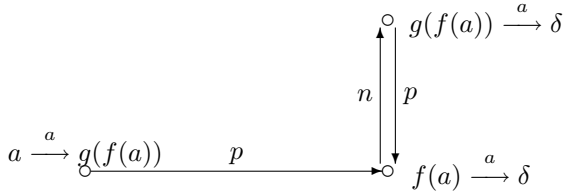


Figure 4.2: The LDG belonging to example 4.2.6

An edge from literal ϕ to ψ is labeled by 'p' to express that ψ is the conclusion and ϕ a positive premise of $\sigma(r)$ for some ground substitution σ and rule $r \in R$. An edge from $t \xrightarrow{a} t'$ to ψ is labeled with 'n' if ψ is the conclusion of $\sigma(r)$ and $t \xrightarrow{a}\!\!\!\!\!/\;$ is a negative premise. If there is a cycle in the literal dependency graph with a negative edge then one may derive from the assumption that for any t'', literal $t \xrightarrow{a} t''$ is not an element of a transition relation $\longrightarrow$ agreeing with P, that $t \xrightarrow{a} t'$ must be an element of $\longrightarrow$, which is a contradiction. As an example a part of the literal dependency graph of example 4.2.6 is depicted in figure 4.2.

Definition 4.2.7. Let $P = (\Sigma, A, R)$ be a TSS. The *(labeled) Literal Dependency Graph (LDG)* G related to P has as nodes the literals in $Tr(\Sigma, A)$ and as labels p and n. The edges of G are given by the triples:

- $\langle \sigma(\phi), p, \sigma(\psi) \rangle$ where σ is a ground substitution such that there is a rule $r \in R$ with a positive premise ϕ and a conclusion ψ

combined with

- $\langle \phi, n, \sigma(\psi) \rangle$ where σ is a ground substitution such that there is a rule $r \in R$ with a negative premise $t \xrightarrow{b}\!\!\!\!\!/\;$ and a conclusion ψ such that for some $t' \in T(\Sigma)$ $\sigma(t \xrightarrow{b} t') = \phi$.

If there is a path between two literals ϕ and ψ of which all edges are labeled with p, it is said that there is a *positive dependency* between ϕ and ψ. If this path contains at least one edge with label n, we say that ψ *depends negatively* on ϕ.

In the next definition the notion of a *stratifiable* TSS is introduced. It is shown that for stratifiable TSS's there exists a transition relation that agrees with it. As the adjective *stratifiable* suggests, it is possible to make a 'stratification'. This will be shown later.

Definition 4.2.8. Let P be a TSS. P is *stratifiable* iff there is no node in the literal dependency graph G of P, such that a path ending in this node contains an infinite number of negative edges.

The following definition assigns an ordinal to each positive literal ϕ. This ordinal represents the number of negative edges in paths ending in ϕ.

Definition 4.2.9. Let P be a stratifiable TSS with a literal dependency graph G. Nodes that have no incoming paths containing a negative edge are called *LDG basic nodes*. Furthermore, ρ is the equivalence relation between literals such that $\phi \,\rho\, \psi$ iff $\phi \equiv \psi$ or there is a path in G from ϕ to ψ and vice versa. Note that if $\phi \,\rho\, \psi$ then ϕ is an LDG basic node iff ψ is an LDG basic node. Define $rank_P$ on the equivalence classes of $Tr(\Sigma, A)/\rho$ as follows:

- $rank_P(\phi/\rho) = 0$ if ϕ is an LDG basic node,
- $rank_P(\phi/\rho) = sup(\{rank_P(\psi/\rho) + 1 \mid (\psi, n, \chi)$ is an edge in G and $\chi \in \phi/\rho\} \cup \{rank_P(\psi/\rho) \mid (\psi, p, \chi)$ is an edge in G, $\chi \in \phi/\rho$ and $\psi \notin \phi/\rho\})$ otherwise.

Here $sup(X)$ gives the least ordinal $\geq$ all ordinals in the set X. Define $rank_P(\phi) = rank_P(\phi/\rho)$.

Example 4.2.10. Here we give an example of a TSS P for which the $rank_P$ function uses infinite ordinals. Take the TSS P with one constant f and as labels the natural numbers. Take as rules:

$$\frac{f \overset{n}{\not\longrightarrow}}{f \overset{}{\longrightarrow} f} \quad n \geq 0,$$

$$\frac{f \overset{n}{\not\longrightarrow}}{f \overset{0}{\longrightarrow} f} \quad \text{for } n \text{ odd.}$$

$rank_P : Tr(\Sigma, A) \to \omega \cdot 2$ is defined by $rank_P(f \overset{n}{\longrightarrow} f) = (n-1)/2$ if n odd and $rank_P(f \overset{n}{\longrightarrow} f) = \omega + n/2$ if n even.

Checking whether or not a literal dependency graph contains cycles with negative edges is laborious and therefore not very useful to check the consistency of a set of rules. The literal dependency graph can be used more fruitfully to construct examples showing that a given TSS is inconsistent. *Local stratifications* [2, 32] provide a more useful technique to show consistency. A stratification of a TSS is given by the following definition.

Definition 4.2.11. Let $P = (\Sigma, A, R)$ be a TSS. A function $S : Tr(\Sigma, A) \to \alpha$, for some ordinal α, is called a *stratification* of P iff for every rule

$$\frac{\{t_k \overset{a_k}{\longrightarrow} t'_k \mid k \in K\} \cup \{t_l \overset{b_l}{\not\longrightarrow} \mid l \in L\}}{t \overset{a}{\longrightarrow} t'} \in R$$

and every substitution $\sigma : V \to T(\Sigma)$ it holds that:

$$\text{for all } k \in K : \; S(\sigma(t_k \overset{a_k}{\longrightarrow} t'_k)) \leq S(\sigma(t \overset{a}{\longrightarrow} t'))$$

$$\text{for all } l \in L \text{ and } t'_l \in T(\Sigma) : \; S(\sigma(t_l \overset{b_l}{\longrightarrow} t'_l)) < S(\sigma(t \overset{a}{\longrightarrow} t'))$$

If P has a stratification, we say that P is *stratified*. For $\beta < \alpha$, $S_\beta = \{\phi \mid S(\phi) = \beta\}$ is called a *stratum*. If all literals with the same label are in the same stratum then we speak about a *label independent* stratification. In the same way we speak about a *source independent* and a *target independent* stratification.

Lemma 4.2.12. *Let $P = (\Sigma, A, R)$ be a TSS. P is stratifiable iff P is stratified.*

Proof. "⇒" As P is stratifiable, the function $rank_P : Tr(\Sigma, A) \to \alpha$ for some ordinal α is defined. It is easy to check that $rank_P$ is a stratification of P.
"⇐" Suppose P is stratified by a stratification $S : Tr(\Sigma, A) \to \alpha$. Construct the literal dependency graph G of P. By transfinite induction on β it is shown that if $S(\phi) = \beta$ then there is no path ending in ϕ in the literal dependency graph, containing an infinite number of negative edges. Suppose the induction holds for all $\beta' < \beta$, $S(\phi) = \beta$ and there is a path ending in ϕ labeled with an infinite number of n's. Then this means that there is a tail of the path

$$...\psi, \phi_n ... \phi_2, \phi_1, \phi$$

such that ϕ depends positively on ϕ_1, ϕ_1 depends positively on ϕ_2 etc., while ϕ_n is the first literal that depends negatively on a literal ψ. Hence, $S(\psi) < S(\phi) = \beta$. Using the induction hypothesis there is no path labeled with an infinite number of n's ending in ψ. But this contradicts the assumption that there was one from ϕ. □

As remarked in example 4.2.5 there is not always one unique transition relation that agrees with P. Therefore, we define, given a TSS P with a stratification S, a relation $\longrightarrow_{P,S}$ which we call the transition relation *associated with* P (and *based on* S). The construction of the transition relation $\longrightarrow_{P,S}$ from a transition system specification is as follows: a literal ϕ with $S(\phi) = 0$ is in $\longrightarrow_{P,S}$ if it can be 'derived' using rules of P, which do not have negative premises, in the ordinary sense. We now know which literals ϕ with $S(\phi) = 0$ are not in $\longrightarrow_{P,S}$. We use this information to 'derive' the literals ϕ with $S(\phi) = 1$ are in $\longrightarrow_{P,S}$. In this way we can continue for all strata.

The transition relation associated with P has two nice properties. When we have a TSS P without negative premises, then the transition relation associated with P exactly coincides with the transition relation containing all provable literals [16]. Moreover, the $\longrightarrow_{P,S}$ is independent of the stratification S. This last statement is proved in lemma 4.2.16.

First the $degree(r)$ of a rule r in a TSS is defined. It is a cardinal that is greater than the number of positive premises in r. Moreover, it is regular. This means that if an ordinal $\alpha_\phi < degree(r)$ is assigned to each positive premise ϕ of r, then there is still some ordinal β such that $\alpha_\phi < \beta < degree(r)$ for all premises ϕ. If r has a finite number of premises, then $degree(r) = \omega$. *degree* is introduced to avoid taking the union over the class of all ordinals in definition 4.2.14. In the proof of theorem 4.2.15 the regularity of $degree(r)$ is crucial.

Definition 4.2.13. Let $P = (\Sigma, A, R)$ be a TSS. Let $r \in R$ be a rule in R. $degree(r)$ is the smallest regular cardinal greater than $|K|$ where K is the index set of positive premises of r. $degree(P)$ is the smallest regular cardinal such that $degree(P) \geq degree(r)$ for each $r \in R$.

Definition 4.2.14. Let $P = (\Sigma, A, R)$ be a TSS. Let $S : Tr(\Sigma, A) \to \alpha$ for some ordinal α be a stratification of P. The transition relation $\longrightarrow_{P,S}$ *associated with* P (and *based on* S) is defined as:

$$\longrightarrow_{P,S} = \bigcup_{0 \leq i < \alpha} \longrightarrow_i^P .$$

where transition relations $\longrightarrow_i^P \subseteq Tr(\Sigma, A)$ $(0 \leq i < \alpha)$, $\longrightarrow_{ij}^P \subseteq Tr(\Sigma, A)$ $(0 \leq i < \alpha,\ 0 \leq j < degree(P))$ are inductively defined by:

$$\longrightarrow_i^P = \bigcup_{0 \leq j < degree(P)} \longrightarrow_{ij}^P \text{ for } 1 \leq i < \alpha$$

$$\longrightarrow_{ij}^P = \{\phi \mid S(\phi) = i,$$

$$\exists \frac{\{\chi_k \mid k \in K\}}{\chi} \in R,\ \exists \sigma : V \to T(\Sigma) :$$
$$\sigma(\chi) = \phi \text{ and } \forall k \in K$$
$$[\chi_k \text{ is positive } \Rightarrow \bigcup_{0 \leq j' < j} \longrightarrow_{ij'}^P \cup \bigcup_{0 \leq i' < i} \longrightarrow_{i'}^P \models \sigma(\chi_k)] \text{ and}$$
$$[\chi_k \text{ is negative } \Rightarrow \bigcup_{0 \leq i' < i} \longrightarrow_{i'}^P \models \sigma(\chi_k)]\}$$

for $0 \leq i < \alpha$ and $0 \leq j < degree(P)$.

Theorem 4.2.15. *Let $P = (\Sigma, A, R)$ be a TSS with the stratification $S : Tr(\Sigma, A) \to \alpha$ for some ordinal α. Then there is a transition relation, namely $\longrightarrow_{P,S}$, that agrees with P.*

Proof. We show that $\longrightarrow_{P,S}$ agrees with P:

$\Rightarrow$) Suppose that for a rule

$$r = \frac{\{t_k \xrightarrow{a_k} t'_k \mid k \in K\} \cup \{t_l \xrightarrow{a_l} \not\rightarrow \mid l \in L\}}{t \xrightarrow{a} t'} \in R$$

and a ground substitution σ all premises hold in $\longrightarrow_{P,S}$. Define $\beta = S(\sigma(t \xrightarrow{a} t'))$. For a negative premise $t_l \xrightarrow{a_l} \not\rightarrow$ it trivially holds that for every $t'' \in T(\Sigma)$ $t_l \xrightarrow{a_l} t'' \notin \bigcup_{0 \leq i < \beta} \longrightarrow_i^P$. For a positive premise $t_k \xrightarrow{a_k} t'_k$ it holds that either $\sigma(t_k \xrightarrow{a_k} t'_k) \in \bigcup_{0 \leq i < \beta} \longrightarrow_i^P$ or $\sigma(t_k \xrightarrow{a_k} t'_k) \in \longrightarrow_\beta^P$. Consider the set $T = \{j \mid j < degree(P)$ and for some $k \in K$ j is the

smallest ordinal such that $\sigma(t_k \xrightarrow{a_k} t'_k) \in \longrightarrow^P_{\beta j}\}$. $|T| \leq |K| < degree(P)$. As $degree(P)$ is a regular cardinal, there is some $0 \leq j' \leq degree(P)$ such that $j'' < j' < degree(P)$ for every $j'' \in T$. Hence, for this j': $\sigma(t \xrightarrow{a} t') \in \longrightarrow^P_{\beta j'}$ by definition. Hence, $\sigma(t \xrightarrow{a} t') \in \longrightarrow_{P,S}$.

$\Leftarrow$) Suppose $\psi \in \longrightarrow_{P,S}$. Then for some $0 \leq i < \alpha$, $0 \leq j < degree(P)$ $\psi \in \longrightarrow^P_{ij}$. According to the definition of $\longrightarrow_{P,S}$ this means that there is a ground substitution σ and a rule

$$r = \frac{\{\chi_k \mid k \in K\}}{\chi} \in R$$

such that $\sigma(\chi) = \psi$ and if χ_k is positive

$$\sigma(\chi_k) \in \bigcup_{0 \leq j' < j} \longrightarrow^P_{ij'} \cup \bigcup_{0 \leq i' < i} \longrightarrow^P_{i'} .$$

But then $\sigma(\chi_k) \in \longrightarrow_{P,S}$. If $\chi_k \equiv t \overset{a}{\not\longrightarrow}$ then for every $t' \in T(\Sigma)$:

$$\sigma(t \xrightarrow{a} t') \notin \bigcup_{0 \leq i' < i} \longrightarrow^P_{i'} .$$

Due to the stratification $S(\sigma(t \xrightarrow{a} t')) < i$. Hence, $\sigma(t \xrightarrow{a} t') \notin \longrightarrow^P_{i'}$ for $i' \geq i$ and therefore $\sigma(t \xrightarrow{a} t') \notin \longrightarrow_{P,S}$. So all premises of $\sigma(r)$ hold in $\longrightarrow_{P,S}$.

□

We show here that the particular stratification used in the construction of $\longrightarrow_{P,S}$ is not of any importance.

Lemma 4.2.16. *Let P be a TSS which is stratified by stratifications S and S'. The transition relation associated with P and based on S is equal to the transition relation associated with P and based on S'.*

Proof. Assume $P = (\Sigma, A, R)$. Suppose $\longrightarrow_{P,S} \neq \longrightarrow_{P,S'}$. This means that there is some ϕ such that either $\phi \in \longrightarrow_{P,S} - \longrightarrow_{P,S'}$ or $\phi \in \longrightarrow_{P,S'} - \longrightarrow_{P,S}$. Assume that ϕ is minimal with respect to S, i.e. $S(\phi) \leq S(\psi)$ for all $\psi \in (\longrightarrow_{P,S} - \longrightarrow_{P,S'}) \cup (\longrightarrow_{P,S'} - \longrightarrow_{P,S})$. Define $i = S(\phi)$.

- Suppose $\phi \in \longrightarrow_{P,S} - \longrightarrow_{P,S'}$. Then $\phi \in \longrightarrow^P_{ij}$ for some $0 \leq j < degree(P)$ (see definition 4.2.2). Assume that ϕ is minimal with respect to $\longrightarrow^P_{ij}$, i.e. for all ψ with $S(\psi) = i$ and $\psi \in \longrightarrow_{P,S} - \longrightarrow_{P,S'}$: $\psi \notin \longrightarrow^P_{ij'}$ with $j' < j$. As $\longrightarrow_{P,S}$ agrees with P there is a ground instantiated rule $\sigma(r)$ with conclusion ϕ and premises χ_k ($k \in K$) such that $\longrightarrow_{P,S} \models \chi_k$. As $\phi \notin \longrightarrow_{P,S'}$ it

cannot be that all premises χ_k ($k \in K$) hold in $\longrightarrow_{P,S'}$. Hence, $\longrightarrow_{P,S'} \not\models \chi_{k'}$ for some $k' \in K$. If $\chi_{k'}$ is a positive literal then $\chi_{k'} \in \bigcup_{0 \leq j'' < j} \stackrel{P}{\longrightarrow}_{ij''} \cup \bigcup_{0 \leq i'' < i} \stackrel{P}{\longrightarrow}_{i''}$ and $\chi_{k'} \notin \longrightarrow_{P,S'}$. But this contradicts one of the assumptions that ϕ is minimal.
If $\chi_{k'} \equiv t \stackrel{a}{\not\longrightarrow}$ then for some $t' \in T(\Sigma)$ $t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_{P,S'} - \longrightarrow_{P,S}$ and $S(t \stackrel{a}{\longrightarrow} t') < i$. But this contradicts the minimality assumption with respect to S.

- Considering $\phi \in \longrightarrow_{P,S'} - \longrightarrow_{P,S}$ leads to a contradiction in almost the same way as the former case.

□

This last lemma allows us to drop the stratification as a subscript in the transition relation $\longrightarrow_{P,S}$ associated to a stratifiable TSS P. Further, it provides the following technique to give an operational semantics in Plotkin style when there are negative premises around: define a TSS P and prove with a convenient stratification that P is stratifiable. Then P alone determines the transition relation $\longrightarrow_P$ associated with P.

In this remainder of this section we show that if we strengthen the requirements on stratifications, then the transition relation that agrees with P is unique.

Definition 4.2.17. Let $P = (\Sigma, A, R)$ be a TSS and let $S : Tr(\Sigma, A) \to \alpha$ for some ordinal α be a stratification of P. S is a *strict stratification* of P if for every substitution σ and every rule

$$r = \frac{\{t_k \stackrel{a_k}{\longrightarrow} t'_k \mid k \in K\} \cup \{t_l \stackrel{a_l}{\not\longrightarrow} \mid l \in L\}}{t \stackrel{a}{\longrightarrow} t'} \in R$$

$\sigma(t \stackrel{a}{\longrightarrow} t')$ is in a strictly higher stratum than $\sigma(t_k \stackrel{a_k}{\longrightarrow} t'_k)$ for $k \in K$ and in a strictly higher stratum than $\sigma(t_l \stackrel{a_l}{\longrightarrow} t'')$ for $l \in L$ and any $t'' \in T(\Sigma)$. In this case we call P *strictly stratifiable*.

If P is strictly stratifiable then this is equivalent to stating that the literal dependency graph of P contains no infinite path ending in some literal ϕ.

Theorem 4.2.18. *Let P be a strictly stratifiable TSS. Then the transition relation that is associated with P is the unique relation that agrees with P.*

Proof. Let $P = (\Sigma, A, R)$. Suppose $\longrightarrow_1$ is a transition relation that agrees with P. P has a strict stratification $S : T(\Sigma) \to \alpha$ for some ordinal α. Let $\longrightarrow_{P,S}$ be the transition relation that is associated with P. Assume, in order to generate a contradiction, that $\longrightarrow_{P,S} \neq \longrightarrow_1$. This implies that there is some literal ϕ such that $\phi \in \longrightarrow_{P,S} - \longrightarrow_1$ or $\phi \in \longrightarrow_1 - \longrightarrow_{P,S}$. Assume furthermore that ϕ is

minimal, i.e. for all $\psi \in (\longrightarrow_{P,S} - \longrightarrow_1) \cup (\longrightarrow_1 - \longrightarrow_{P,S})$: $S(\phi) \leq S(\psi)$. For reasons of symmetry it is enough to consider only one case: $\phi \in \longrightarrow_{P,S} - \longrightarrow_1$. The case where $\phi \in \longrightarrow_1 - \longrightarrow_{P,S}$ goes in exactly the same way. As $\longrightarrow_{P,S}$ agrees with P there is a rule

$$\frac{\{\chi_k \mid k \in K\}}{\chi} \in R$$

and a substitution $\sigma : V \to T(\Sigma)$ such that $\phi = \sigma(\chi)$, $\longrightarrow_{P,S} \models \sigma(\chi_k)$ for all $k \in K$. Then for some $k' \in K$ $\longrightarrow_1 \not\models \sigma(\chi_{k'})$ because otherwise, as $\longrightarrow_1$ agrees with P, $\phi \in \longrightarrow_1$ contradicting the assumption.
If $\sigma(\chi_{k'})$ is a positive literal then $\sigma(\chi'_k) \in \longrightarrow_{P,S}$, $\sigma(\chi_{k'}) \notin \longrightarrow_1$ and $S(\chi_{k'}) < S(\phi)$. This contradicts the minimality of ϕ. If $\sigma(\chi_{k'}) \equiv t \stackrel{a}{\not\longrightarrow}$ then for some $t' \in T(\Sigma)$ $t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_1$, but $t \stackrel{a}{\longrightarrow} t' \notin \longrightarrow_{P,S}$ and $S(t \stackrel{a}{\longrightarrow} t') < S(\phi)$. This contradicts the minimality of ϕ as well. □

4.3 Examples showing the use of stratifications

The techniques of the previous section are introduced to show that specifications using negative premises define a transition relation in a neat way. Here two examples illustrate the use of these techniques.

Example 4.3.1. Here the GSOS-format is defined. It differs slightly from the GSOS-format as given by Bloom, Istrail and Meyer [10] because we do not consider a special rule for guarded recursion. Suppose we have a TSS P with signature $\Sigma = (F, r)$, labels A and rules of the form

$$\frac{\{x_k \stackrel{a_{kl}}{\longrightarrow} y_{kl} \mid k \in K_1, l \in L_1\} \cup \{x_k \stackrel{a_{kl}}{\not\longrightarrow} \mid k \in K_2, l \in L_2\}}{f(x_1, ..., x_{r(f)}) \stackrel{a}{\longrightarrow} t}$$

with $f \in F$, $x_1, ..., x_{r(f)}, y_{kl}$ pairwise different variables, $K_1, K_2 \subseteq \{1, ..., r(f)\}$, L_1, L_2 finite disjoint index sets and $t \in \mathbb{T}(\Sigma)$. There is a unique transition relation that agrees with the rules. This can be seen by giving the strict stratification $S : Tr(\Sigma, A) \to \omega$:

$$S(t \stackrel{a}{\longrightarrow} t') = n \quad \text{if } t \text{ contains } n \text{ function names.}$$

S is strict as the source in the conclusion of any rule contains more function names than any source in the premises.

Example 4.3.2. In [7] a priority operator is defined on process graphs. In [16] an operational definition is given to the priority operator using rules with negative premises. However, the combination of unguarded recursion, the priority operator and renaming [5] gives rise to inconsistencies. Here we show that simple

conditions on either the relabeling operator or recursion can circumvent this problem.

We base this example on the rules for $\mathrm{BPA}_\delta^\epsilon$ as given in [16] (see rules 1-6 in table 4.1). The TSS $P_{prio} = (\Sigma_{prio}, A_{prio}, R_{prio})$ with $\Sigma_{prio} = (F_{prio}, r_{prio})$ contains constant names a for all $a \in Act$ where Act is a given set of atomic actions. We suppose that there is a 'backwardly' well-founded ordering $<$ on Act, which is used to construct a stratification. The signature also contains constant names ϵ for the *empty process*, and δ representing *inaction*, resembling *NIL* in CCS [22].

There is a unary function name θ, the *priority operator*. If x can perform several actions, say $x \xrightarrow{a} x'$ and $x \xrightarrow{b} x''$ then $\theta(x)$ allows only those transitions which are the highest in the ordering $<$. So if $a > b$ then $\theta(x) \xrightarrow{a} \theta(x')$ is an allowed transition while $\theta(x) \xrightarrow{b} \theta(x'')$ is not possible. We have another unary function name ρ_f, the *renaming operator*. f is a renaming function from Act to Act. $\rho_f(x)$ renames the labels of the transitions of x by f. There are two binary operators. *Sequential composition* is denoted by $\cdot$ (this symbol is usually omitted). *Alternative composition* is denoted by $+$.

For recursion it is assumed that there is some given set Ξ with *process names*. Each name in Ξ is a constant in the signature. E is a set of *process declarations* of the form $X \Leftarrow t_X$ for all process names $X \in \Xi$ ($t_X \in T(\Sigma_{prio})$). In $X \Leftarrow t_X$, t_X is the *body* of process name X.

The labels in A_{prio} are given by $Act_\surd$ $(= Act \cup \{\surd\})$. $\surd$ is an auxiliary symbol that is introduced to represent termination of a process. The rules are given in table 4.1. Here a, b range over $Act_\surd$. In rule 9 of table 4.1 we use the abbreviation $\forall b > a\ x \xrightarrow{b} \not\rightarrow$ in the premises. It means that for all $b > a$ there is a premise $x \xrightarrow{b} \not\rightarrow$. As an infinite number of negative premises are allowed in the premises of a rule, rule scheme 9 generates proper rules. With these rules we have the following inconsistency (cf. [6]). Define

$$X \Leftarrow \theta(\rho_f(X) + b)$$

with $f(b) = a$, $f(a) = c$, $f(d) = d$ for all $d \in Act - \{a, b\}$ and $a > b$. Now $X \xrightarrow{b} \epsilon$ iff $X \xrightarrow{b} \not\rightarrow$.

As a first solution for this problem we consider renaming functions satisfying the requirement that if $a > b$ then not $f(b) = a$ for all $a, b \in Act$, i.e. we may not rename actions to ones with higher priority. It is now easy to see that a transition relation associated with P_{prio} exists using the following stratification of P_{prio}. Define $rk(a)$ for all $a \in A_{prio}$ by:

$$rk(a) = sup(\{rk(b) + 1 \mid a < b\}) \quad \text{for } a \in Act$$

where $sup(\emptyset) = 0$ and $rk(\surd) = 0$. Define $S : Tr(\Sigma_{prio}, A_{prio}) \to \alpha$ for some ordinal α by:

$$S(t \xrightarrow{a} t') = rk(a)$$

1.	$a \xrightarrow{a} \epsilon$	$a \neq \surd$	2.	$\epsilon \xrightarrow{\surd} \delta$
3.	$\dfrac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$		4.	$\dfrac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$
5.	$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	$a \neq \surd$	6.	$\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$
7.	$\dfrac{x \xrightarrow{a} x'}{\rho_f(x) \xrightarrow{a} \rho_f(x')}$	$a \neq \surd$	8.	$\dfrac{x \xrightarrow{\surd} x'}{\rho_f(x) \xrightarrow{\surd} \rho_f(x')}$
9.	$\dfrac{x \xrightarrow{a} x' \quad \forall b > a \; x \overset{b}{\not\rightarrow}}{\theta(x) \xrightarrow{a} \theta(x')}$	$a, b \neq \surd$	10.	$\dfrac{x \xrightarrow{\surd} x'}{\theta(x) \xrightarrow{\surd} \theta(x')}$
11.	$\dfrac{t \xrightarrow{a} x'}{X_t \xrightarrow{a} x'}$	for $X_t \Leftarrow t \in E$		

Table 4.1: $\mathrm{BPA}^{\epsilon}_{\delta}$ with renaming and priorities

(it is straightforward to check that S is a stratification of P_{prio}).

Another solution is to disallow that the priority operator appear in the body of a process name. In this case a stratification can be given by:

$$S(t \xrightarrow{a} t') = n \quad \text{where } n \text{ is the total number of occurrences of } \theta\text{'s in } t.$$

A last possibility is obtained by disallowing unguarded recursion in the bodies of process definitions. A stratification can now be constructed as follows: Suppose one has a literal $t \xrightarrow{a} t'$. Let n be the number of θ's in t. Moreover, let m be the number of the θ's in the bodies t'' of all process names X'' ($X'' \Leftarrow t_{X''} \in E$) that occur unguarded in t. Then we define a stratification $S : Tr(\Sigma_{prio}, A_{prio}) \to \omega$ by $S(t \xrightarrow{a} t') = n + m$. One can check that S is a stratification of P_{prio}.

4.4 The *ntyft/ntyxt*-format and the congruence theorem

Often one considers bisimulation equivalence as the finest extensional equivalence that one wants to impose. If bisimulation is not a congruence then one can distinguish bisimilar processes by putting them in appropriate contexts. Therefore, it is a nice property of a format of rules if it guarantees that all operators defined by this format respect bisimulation.

The notion of strong bisimulation equivalence as defined below is from Park [27].

Definition 4.4.1. Let $P = (\Sigma, A, R)$ be a stratifiable TSS. A relation $R \subseteq T(\Sigma) \times T(\Sigma)$ is a (*strong*) (P-) *bisimulation relation* if it satisfies:

1. whenever $t\,R\,u$ and $t \xrightarrow{a}_P t'$ then, for some $u' \in T(\Sigma)$, we have $u \xrightarrow{a}_P u'$ and $t'R\,u'$,

2. conversely, whenever $t\,R\,u$ and $u \xrightarrow{a}_P u'$ then, for some $t' \in T(\Sigma)$, we have $t \xrightarrow{a}_P t'$ and $t'R\,u'$.

We say that two terms $t, t' \in T(\Sigma)$ are (P-)*bisimilar*, notation $t \leftrightarrows_P t'$, iff there is a P-bisimulation relation R such that $t\,R\,t'$. We write $t \leftrightarrows t'$ if P is clear from the context. Note that $\leftrightarrows_P$ is an equivalence relation.

For TSS's without negative premises, the *tyft/tyxt*-format [16] is the most general format for which bisimulation is a congruence. Here we introduce the *ntyft/ntyxt*-format as the most general extension of the *tyft/tyxt*-format with negative premises such that for operators defined in this format bisimulation is again a congruence.

Definition 4.4.2. Let $\Sigma = (F, r)$ be a signature. Let $P = (\Sigma, A, R)$ be a stratifiable TSS. A rule $r \in R$ is in *ntyft-format* if it has the form:

$$\frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \ \cup \ \{t_l \xrightarrow{b_l} \!\!\!\!/ \ \mid l \in L\}}{f(x_1, ..., x_{r(f)}) \xrightarrow{a} t}$$

with K and L index sets, y_k, x_i $(1 \leq i \leq r(f))$ all different variables, $a_k, b_l, a \in A$, $f \in F$ and $t_k, t_l, t \in \mathbb{T}(\Sigma)$. A rule $r \in R$ is in *ntyxt-format* if it fits:

$$\frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \ \cup \ \{t_l \xrightarrow{b_l} \!\!\!\!/ \ \mid l \in L\}}{x \xrightarrow{a} t}$$

with K, L index sets, y_k, x all different variables, $a_k, b_l, a \in A$, t_k, t_l and $t \in \mathbb{T}(\Sigma)$. P is in *ntyft-format* if all its rules are in *ntyft*-format and P is in *ntyft/ntyxt-format* if all its rules are either in *ntyft*- or in *ntyxt*-format.

From examples given in [16] it follows that the *tyft/tyxt*-format cannot be generalised in any obvious way without endangering the congruence property of bisimulation equivalence. This implies that for the *ntyft/ntyxt*-format, the positive premises cannot be generalised. Since the negative premises are already as general as possible, the *ntyft/ntyxt*-format cannot be generalised in any obvious way without losing the congruence property of strong bisimulation.

In the remainder of this section we show that the congruence theorem holds for the *ntyft/ntyxt*-format. In order to do so, we need a same well-foundedness restriction on the premises of the rules as was necessary to prove the congruence theorem for the *tyft/tyxt*-format. It is an open question whether both congruence theorems can be proved without this restriction.

Definition 4.4.3 *(well-founded).* Let $P = (\Sigma, A, R)$ be a TSS. Let $W = \{t_k \xrightarrow{a_k} t'_k \mid k \in K\} \subseteq \mathbb{T}(\Sigma) \times A \times \mathbb{T}(\Sigma)$ be a set of positive literals over Σ and A. The *Variable Dependency Graph (VDG)* of W is a directed (unlabeled) graph with:

- Nodes: $\bigcup_{k \in K} Var(t_k \xrightarrow{a_k} t'_k)$,
- Edges: $\{\langle x, y \rangle \mid x \in Var(t_k),\ y \in Var(t'_k) \text{ for some } k \in K\}$.

W is called *well-founded* if any backward chain of edges in the variable dependency graph is finite. A rule is called *well-founded* if its set of positive premises is well-founded. A TSS is called *well-founded* if its rules are well-founded.

Note that it is not useful to include negative premises in this definition as they do not have a target and therefore do not determine values of variables.

Example 4.4.4. The variable dependency graph of

$$\{f(x', y_1) \xrightarrow{a} y_2, g(x, y_2) \xrightarrow{a} y_1\}$$

is given in figure 4.3. The set of rules is not well-founded because the graph contains a cycle.

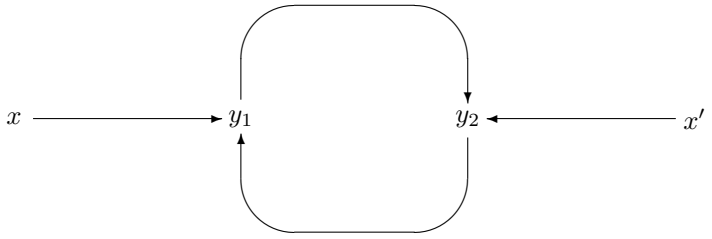


Figure 4.3: A VDG with a cycle

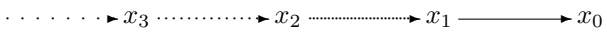


Figure 4.4: A VDG that is not well-founded

Example 4.4.5. Consider the variable dependency graph G of $\{x_{n+1} \xrightarrow{a_n} x_n \mid n \in \mathbb{N}\}$. G is not well-founded because for any variable x_i ($i \in \mathbb{N}$) that acts as a node in G, there is an infinite path ending in this node. A part of G is depicted in figure 4.4.

The following lemma says that for well-founded TSS's in *ntyft/ntyxt*-format, it is sufficient to consider only target independent stratifications.

Lemma 4.4.6. *Let P be a well-founded stratifiable TSS in ntyft/ntyxt-format. Then P has a target independent stratification.*

Proof. Let $P = (\Sigma, A, R)$ with $\Sigma = (F, r)$ and let $S : Tr(\Sigma, A) \to \alpha$ for some ordinal α be a stratification of P. Define a mapping $S' : Tr(\Sigma, A) \to \alpha + 1$ by:

$$S'(t \xrightarrow{a} t') = sup(\{S(t \xrightarrow{a} u) + 1 \mid u \in T(\Sigma)\}).$$

We show that S' is a stratification of P. As S' is clearly target independent, this is sufficient to finish the proof.

Consider a rule in *ntyxt*-format (the argument for a rule in *ntyft*-format is exactly the same)

$$r = \frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \cup \{t_l \not\xrightarrow{t_l} \mid l \in L\}}{x \xrightarrow{a} t} \in R$$

and some ground substitution σ. For each positive premise $t_k \xrightarrow{a} y_k$ we have that for each term $u \in T(\Sigma)$:

$$\begin{aligned} S(\sigma(t_k) \xrightarrow{a} u)) &= S(\sigma'(t_k \xrightarrow{a_k} y_k)) \\ &\leq S(\sigma'(x \xrightarrow{a} t)) \\ &= S(\sigma(x) \xrightarrow{a} \sigma'(t)) \end{aligned} \tag{4.1}$$

where σ' is a ground substitution defined by:

$$\sigma'(z) = \begin{cases} \sigma(z) & \text{if } z \not\equiv y_k \\ u & \text{if } z \equiv y_k. \end{cases}$$

As P is well-founded and in *ntyft/ntyxt*-format, $\sigma'(t_k) = \sigma(t_k)$ and $\sigma'(x) = \sigma(x)$. Now it is easy to see that S' is a stratification:

$$\begin{array}{rcl} S'(\sigma(t_k \xrightarrow{a_k} y_k)) & = & sup(\{S(\sigma(t_k) \xrightarrow{a_k} u) + 1 \mid u \in T(\Sigma)\}) \\ & \overset{(4.1)}{\leq} & sup(\{S(\sigma(x) \xrightarrow{a} u') + 1 \mid u' \in T(\Sigma)\}) \\ & = & S'(\sigma(x \xrightarrow{a} t)) \end{array}$$

and

$$\begin{array}{rcl} S'(\sigma(t_l) \xrightarrow{a_l} u)) & = & sup(\{S(\sigma(t_l) \xrightarrow{a_l} u') + 1 \mid u' \in T(\Sigma)\}) \\ & \leq & S(\sigma(x \xrightarrow{a} t)) \\ & < & sup(\{S(\sigma(x) \xrightarrow{a} u'') + 1 \mid u'' \in T(\Sigma)\}) \\ & = & S'(\sigma(x \xrightarrow{a} t)). \end{array}$$

□

Definition 4.4.7. Let W be a set of positive literals which is well-founded and let G be the variable dependency graph of W. Let $Var(W)$ be the set of variables occurring in literals in W. Define for each $x \in Var(W)$: $n_{VDG}(x) = sup(\{n_{VDG}(y) + 1 \mid \langle y, x\rangle \text{ is an edge of } G\})$ $(sup(\emptyset) = 0)$.

If W is a set of positive premises of a rule in *ntyft/ntyxt*-format then $n_{VDG}(x) \in \mathbf{N}$ for each $x \in Var(W)$; every variable y_k only occurs once in the right hand side of a positive literal in the premises. As the term t_k is finite, it contains only a finite number of variables x. Therefore the set $U = \{n_{VDG}(x) + 1 \mid \langle x, y_k\rangle$ is an edge of $G\}$ is finite. Hence, $n_{VDG}(y_k) = sup(U)$ is a natural number.

Definition 4.4.8. Two stratifiable TSS's P and P' are *transition equivalent* if $\longrightarrow_P = \longrightarrow_{P'}$.

Hence, two TSS's are transition equivalent if they have the same signature, the same set of labels and if the sets of rules determine the same associated transition relation. The particular form of the rules is not of importance.

Lemma 4.4.9. *Let $P = (\Sigma, A, R)$ be a stratifiable TSS in ntyft/ntyxt-format. Then there is a stratifiable TSS $P' = (\Sigma, A, R')$ in ntyft-format that is transition equivalent with P.*

Proof. Let $\Sigma = (F, rank)$. Let R' contain every rule $r \in R$ that is in *ntyft*-format together with the rules $\sigma_f(r)$ for every rule $r \in R$ in *ntyxt*-format and every function name $f \in F$ where σ_f is defined as:

$\sigma_f(x) = f(z_1, ..., z_{rank(f)})$ if x is the source in the conclusion of r. $z_1, ..., z_{rank(f)}$ are variables not occurring in r.
$\sigma_f(x) = x$ otherwise

Note that R' is in *ntyft*-format. As P is stratifiable, there is a stratification $S : Tr(\Sigma, A) \to \alpha$ of P. It is not hard to see that this stratification is also a stratification for P'. It is enough to show that $\longrightarrow_{P,S} = \longrightarrow_{P',S}$. In order to see this we only need to prove that $\longrightarrow_{ij}^{P} = \longrightarrow_{ij}^{P'}$ for all $0 \leq i < \alpha, 0 \leq i < degree(P)$. This is done by induction on i and within this induction, an induction on j.

$\subseteq$) Suppose $\phi \in \longrightarrow_{ij}^{P}$ for some i and j. According to the definition of $\longrightarrow_{ij}^{P}$ this means that there is a ground instantiated rule $\sigma(r)$ with conclusion ϕ and premises χ_k $(k \in K)$ such that

$$\bigcup_{0 \leq j' < j} \longrightarrow_{ij'}^{P} \cup \bigcup_{0 \leq i' < i} \longrightarrow_{i'}^{P} \models \chi_k.$$

If χ_k is positive then, inductively,

$$\bigcup_{0 \leq j' < j} \longrightarrow_{ij'}^{P'} \cup \bigcup_{0 \leq i' < i} \longrightarrow_{i'}^{P'} \models \chi_k.$$

If $\chi_k \equiv t \stackrel{a}{\not\longrightarrow}$ then for all $t' \in T(\Sigma)$: $t \stackrel{a}{\longrightarrow} t' \notin \bigcup_{0 \leq i' < i} \longrightarrow_{i'}^{P}$ and therefore $t \stackrel{a}{\longrightarrow} t' \notin \bigcup_{0 \leq i' < i} \longrightarrow_{i'}^{P'}$. Hence, in both cases

$$\bigcup_{0 \leq j' < j} \longrightarrow_{ij'}^{P'} \cup \bigcup_{0 \leq i' < i} \longrightarrow_{i'}^{P'} \models \chi_k$$

for all $k \in K$. If r is an *ntyft*-rule, one can apply $\sigma(r)$ again to obtain $\phi \in \longrightarrow_{ij}^{P'}$. If r is in *ntyxt*-format and the left side of ϕ is $f(t_1, ..., t_{rank(f)})$, apply the instantiated rule $\sigma'(\sigma_f(r))$ where $\sigma'(x) = t_k$ for $x = z_k$ $(1 \leq k \leq rank(f))$ and $\sigma'(x) = \sigma(x)$ otherwise. Hence, $\phi \in \longrightarrow_{ij}^{P'}$.

$\supseteq$) The reverse implication can be shown in the same way.

□

Definition 4.4.10. Let $P = (\Sigma, A, R)$ be a TSS. Let $r \in R$ be a rule. A variable x is called $free$ in r if it occurs in r but not in the source of the conclusion or in the target of a positive premise. The rule r is called *pure* if it is well-founded and does not contain free variables. P is called *pure* if all rules in R are pure.

Lemma 4.4.11. *Let $P = (\Sigma, A, R)$ be a stratifiable and well-founded TSS in ntyft/ntyxt*-format. Then there is a stratifiable TSS $P' = (\Sigma, A, R')$ in pure *ntyft/ntyxt*-format which is transition equivalent with P. If P is in *ntyft*-format then P' is in pure *ntyft*-format.

Proof. R' contains a rule $\sigma(r)$ for every rule $r \in R$ and substitution σ satisfying:

$$\begin{array}{ll} \sigma(x) = t \in T(\Sigma) & \text{if } x \text{ is free in } r, \\ \sigma(x) = x & \text{otherwise.} \end{array}$$

Note that P' constructed in this way is pure, if P is in *ntyft*-format then P' is also in *ntyft*-format and any stratification for P is also a stratification for P'. The remainder of the proof proceeds in the same way as the proof of lemma 4.4.9. □

Next, we state the congruence theorem.

Theorem 4.4.12. *Let P be a well-founded stratifiable TSS in ntyft/ntyxt-format. Then $\leftrightarrows_P$ is a congruence relation.*

Proof. This proof closely resembles the proof of the same theorem in [16]. Assume $P = (\Sigma, A, R_0)$ with $\Sigma = (F, r)$. According to lemma 4.4.9 and lemma 4.4.11 we may assume that P is in pure *ntyft*-format. As P is stratifiable, there is a target independent stratification $S : Tr(\Sigma, A) \to \alpha$ for some ordinal α of P. Furthermore, there is a transition relation $\longrightarrow_P$ associated with P. We must show that for all $f \in F$, $u_1, ..., u_{r(f)}$, $v_1, ..., v_{r(f)} \in T(\Sigma)$:

$$\forall 1 \leq k \leq r(f) : \quad u_k \leftrightarrows_P v_k \quad \Rightarrow \quad f(u_1, ..., u_{r(f)}) \leftrightarrows_P f(v_1, ..., v_{r(f)}).$$

In order to do so, we define a relation $R \subseteq T(\Sigma) \times T(\Sigma)$ as the minimal relation satisfying:

1. $\leftrightarrows_P \subseteq R$,

and for all function names $f \in F$

2. $\forall 1 \leq k \leq r(f) : \; u_k \, R \, v_k \quad \Rightarrow \quad f(u_1, ..., u_{r(f)}) \, R \, f(v_1, ..., v_{r(f)})$

For the relation R we have the following useful fact.

Fact 1. Let $t \in \mathbb{T}(\Sigma)$ and let $\sigma, \sigma' : V \to T(\Sigma)$ be substitutions such that for all x in $Var(t) : \; \sigma(x) \, R \, \sigma'(x)$. Then $\sigma(t) \, R \, \sigma'(t)$.

Proof of fact 1. Straightforward induction on the structure of t. □

If we show that R is a bisimulation relation then it immediately follows that $R = \leftrightarrows_P$ and consequently that $\leftrightarrows_P$ is a congruence relation. In order to see that R is a bisimulation relation we must check that R has the transfer property: if $u \, R \, v$ and $u \xrightarrow{a} u'$ then there is a v' with $v \xrightarrow{a} v'$ and $u' \, R \, v'$ and vice versa. If $u \leftrightarrows_P v$ then this is trivial. So suppose $u = f(u_1, ..., u_{r(f)})$, $v = f(v_1, ..., v_{r(f)})$ and $u_k \, R \, v_k$ for $1 \leq k \leq r(f)$. We are ready if we have shown (by induction on β) that the following holds for all β:

If $\mathcal{L}(f(u_1, ..., u_{r(f)}), a) + \mathcal{L}(f(v_1, ..., v_{r(f)}), a) = \beta$ then

– $f(u_1, ..., u_{r(f)}) \xrightarrow{a} u' \in \longrightarrow_P$ *and* $u_k R\, v_k$ *for* $1 \leq k \leq r(f)$ *implies* $\exists v'\ f(v_1, ..., v_{r(f)}) \xrightarrow{a} v' \in \longrightarrow_P$ *and* $u' R\, v'$

– *vice versa.*

Here we define $\mathcal{L}(t, a) = S(t, a, t')$ for some $t' \in T(\Sigma)$. The definition of $\mathcal{L}(t, a)$ is correct because S is target independent. As the induction hypothesis is symmetric we need only check one halve of it. Suppose the induction hypothesis holds for all $\beta' < \beta$. The validity of the induction hypothesis for β follows immediately if the following fact holds for all $1 \leq i < \alpha$ and $1 \leq j \leq degree(P)$:

If $\mathcal{L}(f(u_1, ..., u_{r(f)}), a) + \mathcal{L}(f(v_1, ..., v_{r(f)}), a) = \beta$,
$f(u_1, ..., u_{r(f)}) \xrightarrow{a} u' \in \longrightarrow^P_{ij}$ and $u_k R\, v_k$ for $1 \leq k \leq r(f)$ then
$\exists v'\ f(v_1, ..., v_{r(f)}) \xrightarrow{a} v' \in \longrightarrow_P$ and $u' R\, v'$

We prove this statement with induction on i and within that with induction on j. So suppose the second induction hypothesis holds for $i' < i$ or for $i' = i$ if $j' < j$. Assume $\mathcal{L}(u, a) + \mathcal{L}(v, a) = \beta$ and $u \xrightarrow{a} u' \in \longrightarrow^P_{ij}$. As $\longrightarrow_P$ agrees with P, there is a rule

$$r = \frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \ \cup\ \{t_l \xrightarrow{a_l} \not\rightarrow \mid l \in L\}}{f(x_1, ..., x_{r(f)}) \xrightarrow{a} t} \in R_0$$

and a substitution σ such that:

- $\sigma(f(x_1, ..., x_{r(f)})) = u$,
- $\sigma(x_i) = u_i$ for $1 \leq i \leq r(f)$,
- $\sigma(t) = u'$,
- $\longrightarrow_P \models \sigma(t_k \xrightarrow{a_k} y_k)$ and $\longrightarrow_P \models \sigma(t_l) \xrightarrow{a_l} \not\rightarrow$.

We will use rule r again in order to show that for some v' $v \xrightarrow{a} v' \in \longrightarrow_P$ and $u' R v'$. Consider the VDG G of the positive premises of r. With induction on n we show that the following fact holds for all n:

Fact 2. There is a ground substitution σ' such that for any $x \in nodes(G)$ with $n_{VDG}(x) < n$ $\sigma(x)\, R\, \sigma'(x)$, if $x = y_k$ for some $k \in K$ then $\sigma'(t_k \xrightarrow{a} y_k) \in \longrightarrow_P$ and if $x = x_i$, then $\sigma'(x_i) = v_i$.

Proof of fact 2. Suppose x is a node of G with $n_{VDG}(x) = n$ and the claim holds for $n' < n$. As r is pure there are two cases.

- $x = x_i$ $(1 \leq i \leq r(f))$. The claim holds for n as $\sigma(x) = u_i\, R\, v_i = \sigma'(x)$.

- $x = y_k$ $(k \in K)$ and $t_k \xrightarrow{a} y_k$ is a premise of r. By induction it holds that there is a ground substitution σ' such that for all $y \in Var(t_k)$: $\sigma(y)\, R\, \sigma'(y)$. By fact 1 $\sigma(t_k)\, R\, \sigma'(t_k)$. Now distinguish between two cases:

  1. $\sigma(t_k) \leftrightarrows_P \sigma'(t_k)$. Hence there is a $w \in T(\Sigma)$ such that $\sigma'(t_k) \xrightarrow{a_k} w \in \longrightarrow_P$ and $\sigma(y_k)\, R\, w$.
  2. There is a function name g in F and there are terms $w_{k'}, w'_{k'}$ for $1 \leq k' \leq r(g)$ such that:

$$\sigma(t_k) = g(w_1, ..., w_{r(g)}),$$

$$\sigma'(t_k) = g(w'_1, ..., w'_{r(g)}) \text{ and}$$

$$w_j\, R\, w'_j \text{ for } 1 \leq j \leq r(g).$$

  Furthermore, we know that $\mathcal{L}(\sigma(t_k), a_k) + \mathcal{L}(\sigma'(t_k), a_k) \leq \mathcal{L}(u, a) + \mathcal{L}(v, a)$. Also $\sigma(t_k \xrightarrow{a_k} y_k) \in \bigcup_{i'<i} \overset{P}{\longrightarrow}_{i'} \cup \bigcup_{j'<j} \overset{P}{\longrightarrow}_{ij'}$. Now we can apply the first or second induction hypothesis which gives that there is a w such that $g(w'_1, ..., w'_{r(g)}) \xrightarrow{a_k} w \in \longrightarrow_P$ and $\sigma(y_k)\, R\, w$.

  So, for any x with $n_{VDG}(x) = n$ we can find a w_x such that $\sigma(x)\, R\, w_x$. Define a ground substitution σ'' such that $\sigma''(x') = \sigma'(x')$ if $n_{VDG}(x') \neq n$ and $\sigma''(x') = w_{x'}$ if $n_{VDG}(x') = n$. Clearly, all inductive properties hold for σ''.

□

Now the proof of the theorem can be finished. For all positive premises ϕ of r it follows that we can prove that $\sigma'(\phi) \in \longrightarrow_P$ for some ground substitution σ' satisfying the properties of fact 2. We show that for each negative premise $t_l \xrightarrow{a_l /\!\!\rightarrow}$ in r $\sigma'(t_l) \xrightarrow{a_l /\!\!\rightarrow}$ also holds in $\longrightarrow_P$. We know using fact 1 that $\sigma(t_l)\, R\, \sigma'(t_l)$ because $\sigma(x) R \sigma'(x)$ for all variables x in t_l. By definition of R there are two possibilities.

- $\sigma(t_l) \leftrightarrows_P \sigma'(t_l)$. In this case $\sigma'(t_l) \xrightarrow{a_l /\!\!\rightarrow}$ clearly holds in $\longrightarrow_P$.
- $\sigma(t_l) = g(w_1, ..., w_{r(g)})$ and $\sigma'(t_l) = g(w'_1, ..., w'_{r(g)})$, $g \in F$ and $w_i\, R\, w'_i$ $(1 \leq i \leq r(g))$. In order to arrive at a contradiction we assume that for some $w \in T(\Sigma)$ $\sigma'(t_l) \xrightarrow{a_k} w$. Clearly, $\mathcal{L}(\sigma(t_l), a_l) + \mathcal{L}(\sigma'(t_l), a_l) < \mathcal{L}(u, a) + \mathcal{L}(v, a)$. So by applying the first induction hypothesis we know that $\exists w'\ \sigma(t_l) \xrightarrow{a_l} w'$. But this contradicts that $\sigma(t_l) \xrightarrow{a_l /\!\!\rightarrow}$ holds in $\longrightarrow_P$. So for every negative premise $t_l \xrightarrow{a_l /\!\!\rightarrow}$ of r: $\longrightarrow_P \models \sigma'(t_l) \xrightarrow{a_l /\!\!\rightarrow}$.

Now as all premises of $\sigma'(r)$ hold, we may conclude that $\sigma'(f(x_1, ..., x_{r(f)}) \xrightarrow{a} t) \in \longrightarrow_P$. Define $v' = \sigma'(t)$. For all $x \in Var(t) :\ \sigma(x)\, R\, \sigma'(x)$. By an application of fact 1 it follows that $\sigma(t)\, R\, \sigma'(t)$ or equivalently, $u'\, R\, v'$. This completes the induction step for the second induction hypothesis. □

4.5 Modular properties of TSS's

Sometimes one wants to extend a TSS with new functions and constants. Therefore the *sum* of two TSS's is introduced [16]. The combination of two TSS's P_0 and P_1 is denoted by $P_0 \oplus P_1$ where we generally assume that P_1 is the extension of P_0. With negative premises care is needed to guarantee that $P_0 \oplus P_1$ still defines a transition relation.

If P_1 is added to P_0 it would be nice if all literals with source $t \in T(\Sigma_0)$ in $\longrightarrow_{P_0 \oplus P_1}$ are exactly the literals in $\longrightarrow_{P_0}$. In this case we say that $P_0 \oplus P_1$ is a *conservative extension* of P_0.

Definition 4.5.1. Let $\Sigma_i = (F_i, r_i)$ $(i = 0, 1)$ be two signatures such that $f \in F_0 \cap F_1 \Rightarrow r_0(f) = r_1(f)$. The *sum* of Σ_0 and Σ_1, notation $\Sigma_0 \oplus \Sigma_1$, is the signature:

$$\Sigma_0 \oplus \Sigma_1 = (F_0 \oplus F_1, \lambda f.\text{if } f \in F_0 \text{ then } r_0(f) \text{ else } r_1(f)).$$

Definition 4.5.2. Let $P_i = (\Sigma_i, A_i, R_i)$ $(i = 0, 1)$ be two TSS's with $\Sigma_0 \oplus \Sigma_1$ defined. The *sum* of P_0 and P_1, notation $P_0 \oplus P_1$, is the TSS:

$$P_0 \oplus P_1 = (\Sigma_0 \oplus \Sigma_1, A_0 \cup A_1, R_0 \cup R_1).$$

Definition 4.5.3. Let $P_i = (\Sigma_i, A_i, R_i)$ $(i = 0, 1)$ be two TSS's with $P = P_0 \oplus P_1$ defined. Let $P = (\Sigma, A, R)$. We say that P is a *conservative extension* of P_0 and that P_1 *can be added conservatively* to P_0 if $P_0 \oplus P_1$ is stratifiable and for all $t \in T(\Sigma_0)$, $a \in A$ and $t' \in T(\Sigma)$:

$$t \xrightarrow{a} t' \in \longrightarrow_P \quad \Leftrightarrow \quad t \xrightarrow{a} t' \in \longrightarrow_{P_0}.$$

Remark 4.5.4. If $P_0 \oplus P_1 = (\Sigma, A, R)$ is a conservative extension of $P_0 = (\Sigma_0, A_0, R_0)$ then it follows immediately that for all $t, u \in T(\Sigma_0) : t \leftrightarrows_{P_0} u \Leftrightarrow t \leftrightarrows_{P_0 \oplus P_1} u$.

The following theorem gives conditions under which a TSS P_1 can be added conservatively to P_2. The theorem is the same as the one that holds for TSS's without negative premises [16], except for the constraint that $P_0 \oplus P_1$ is stratifiable. By an example it will be shown that this condition is necessary. That the other conditions cannot be weakened, is shown in [16].

Theorem 4.5.5. *Let $P_0 = (\Sigma_0, A_0, R_0)$ be a TSS in pure ntyft/ntyxt-format and let $P_1 = (\Sigma_1, A_1, R_1)$ be a TSS in ntyft-format such that there is no rule in R_1 containing a function name from Σ_0 in the source of its conclusion. Let $P = P_0 \oplus P_1$ be defined and stratifiable. Then P_1 can be added conservatively to P_0.*

Proof. Let $P = (\Sigma, A, R)$. As P is stratifiable there is a stratification $S : Tr(\Sigma, A) \to \alpha$ for some ordinal α for P. Define $S^0 : Tr(\Sigma_0, A_0) \to \alpha$ by $S^0(\phi) =$

$S(\phi)$. It is not hard to check that S^0 is a stratification of P_0. Hence, $\longrightarrow_P$ and $\longrightarrow_{P_0}$ are the transition relations associated to P and P_0, respectively.

It is sufficient to prove that

$$t \in T(\Sigma_0),\ a \in A_0,\ t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_P \quad \Leftrightarrow \quad t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_{P_0},\ t' \in T(\Sigma_0).$$

This is done by induction on the ordinal β ($0 \leq \beta < \alpha$) with $S(t \stackrel{a}{\longrightarrow} t') = S^0(t \stackrel{a}{\longrightarrow} t') = \beta$.

Assume that the induction hypothesis holds for all $\beta' < \beta$.
"$\Rightarrow$" Suppose $t \stackrel{a}{\longrightarrow} t' \in \stackrel{P}{\longrightarrow}_{\beta j}$ for some j. Here $\stackrel{P}{\longrightarrow}_{\beta j}$ is the relation from definition 4.2.14 to construct $\longrightarrow_P$. By induction on j it is shown that:

$$t \in T(\Sigma_0),\ a \in A_0,\ t \stackrel{a\ P}{\longrightarrow}_{\beta j} t' \quad \Rightarrow \quad t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_{P_0},\ t' \in T(\Sigma_0).$$

As $\longrightarrow_P$ agrees with P there is a rule $r \in R$ with conclusion $u \stackrel{a}{\longrightarrow} u'$ and a substitution $\sigma : V \to T(\Sigma)$ such that $\sigma(u) = t,\ \sigma(u') = t'$. $r \notin R_1$ as all rules in R_1 are in *ntyft*-format, containing function names not occurring in Σ_0 in the left hand side of their conclusions. So $r \in R_0$. In the remainder we only deal with the case that r is in *ntyft*-format. The case that r is in *ntyxt*-format goes in the same way. So assume r is equal to ($u = f(x_1, ..., x_{r(f)})$):

$$\frac{\{s_k \stackrel{a_k}{\longrightarrow} y_k \mid k \in K\} \cup \{u_l \stackrel{b_l}{\not\longrightarrow} \mid l \in L\}}{f(x_1, ..., x_{r(f)}) \stackrel{a}{\longrightarrow} u'}$$

Now we use induction on $n_{VDG}(x)$ of the variable dependency graph G of the premises of r to prove that for all $x \in Var(r)$: $\sigma(x) \in \Sigma_0$ and if $x = y_k$ ($k \in K$) then $\sigma(s_k \stackrel{a_k}{\longrightarrow} y_k) \in \longrightarrow_{P_0}$. Suppose $n_{VDG}(x) = n \in \mathbb{N}$. As P_0 is pure, we distinguish two cases:

- $x = x_i$ ($1 \leq i \leq r(f)$). As $t \in T(\Sigma_0)$, $\sigma(x) \in T(\Sigma_0)$.
- $x = y_k$ ($k \in K$) and $s_k \stackrel{a_k}{\longrightarrow} y_k$ is a positive premise of r. By induction we know that for all $y \in Var(s_k)$ $\sigma(y) \in T(\Sigma_0)$. As $r \in R_0$, $\sigma(s_k) \in T(\Sigma_0)$. By induction and $\sigma(s_k \stackrel{a_k}{\longrightarrow} y_k) \in \longrightarrow_{P_0 \oplus P_1}$, we can derive $\sigma(s_k \stackrel{a_k}{\longrightarrow} y_k) \in \longrightarrow_{P_0}$ and $\sigma(y_k) \in T(\Sigma_0)$.

As a consequence of this inductive proof it holds for all positive premises ϕ of r that $\sigma(\phi) \in \longrightarrow_{P_0}$. For a negative premise $u_l \stackrel{b_l}{\not\longrightarrow}$ we assume, in order to generate a contradiction that $\exists\ u_l' \in T(\Sigma_0)\ \sigma(u_l \stackrel{b_l}{\longrightarrow} u_l') \in \longrightarrow_{P_0}$. As $\sigma(u_l \stackrel{b_l}{\longrightarrow} u_l')$ is in a strictly lower stratum than $t \stackrel{a}{\longrightarrow} t'$ in S^0, it follows by induction that $\sigma(u_l \stackrel{b_l}{\longrightarrow} u_l') \in \longrightarrow_P$. This contradicts $\sigma(u_l) \stackrel{b_l}{\not\longrightarrow}$.

As $\longrightarrow_{P_0}$ agrees with P_0 and all premises of $\sigma(r)$ hold in $\longrightarrow_{P_0}$ it follows that $\sigma(u \xrightarrow{a} u')$ also holds in $\longrightarrow_{P_0}$. As for all variables in $Var(r)$, $\sigma(r) \in T(\Sigma_0)$, it also holds that $\sigma(u') \in T(\Sigma_0)$.

"$\Leftarrow$" This case has the same structure as the proof of "$\Rightarrow$" Take as intermediate induction hypothesis:

$$t \xrightarrow{a} t' \in \longrightarrow_{P_0} \quad \Rightarrow \quad t \xrightarrow{a} t' \in \longrightarrow_{P} .$$

We skip the details but we remark that induction on n_{VDG} is not necessary. From the induction hypothesis it follows that :

$$t \xrightarrow{a} t' \in \longrightarrow_{P_0} \quad \Rightarrow \quad t \xrightarrow{a} t' \in \longrightarrow_{P},\ t \in T(\Sigma_0),\ a \in A_0.$$

After the combination of this result with "$\Rightarrow$" the outermost induction step is proved. From this the theorem follows immediately. □

In the remainder of this section we study how we can combine stratifications of two stratifiable TSS's P_0 and P_1 to a stratification of $P_0 \oplus P_1$. The following examples show that in general the sum of two stratifiable TSS's is not stratifiable.

Example 4.5.6. This example shows that under certain circumstances it can even be dangerous to extend the signature of a TSS. Let P_0 be a TSS with unary function name f, a label a and a rule:

$$\frac{f(x) \xrightarrow{a}\!\!\!\!\!/\ \ }{f(x) \xrightarrow{a} f(x)}$$

This TSS is stratifiable as there are no ground instances of literals. Adding a TSS P_1 that only contains the single constant c already leads to an inconsistency. If $\longrightarrow$ is a relation that agrees with $P_0 \oplus P_1$ then $\longrightarrow \models f(c) \xrightarrow{a} f(c)$ if and only if $\longrightarrow \models f(c) \xrightarrow{a}\!\!\!\!\!/\ \ $.

Example 4.5.7. This is a less trivial example that shows a problem that can occur when stratifying the sum of stratifiable TSS's. Let P_0 consist of a unary function name g, a constant δ, labels a, b and a rule:

$$\frac{x \xrightarrow{b}\!\!\!\!\!/\ \ }{g(x) \xrightarrow{a} \delta}.$$

P_1 consists of unary function names g and f, constant δ, labels a, b and a rule:

$$\frac{g(f(x)) \xrightarrow{a} y}{f(x) \xrightarrow{b} \delta}.$$

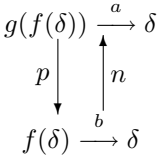


Figure 4.5: The LDG belonging to example 4.5.7

Both P_0 and P_1 have an associated transition relation. $P_0 \oplus P_1$, however, makes it possible to show that $f(\delta) \xrightarrow{b} \delta$ iff $f(\delta) \not\xrightarrow{b}$ for any transition relation $\longrightarrow$ agreeing with $P_0 \oplus P_1$. In figure 4.5 the dependency graph of $P_0 \oplus P_1$ is drawn. The negative edge comes from P_0 and the positive edge from P_1, together constituting a cycle with a negative edge.

Checking the stratifiability of the sum of two stratifiable TSS's can be done by giving a stratification for $P_0 \oplus P_1$. Sometimes the following theorem is helpful.

Theorem 4.5.8. *Let $\Sigma_0 = (F_0, \bar{r_0})$ and $\Sigma_1 = (F_1, \bar{r_1})$ be signatures such that for some constants a_0, a_1: $a_0 \in F_0$ and $a_1 \in F_1$. Let $P_0 = (\Sigma_0, A_0, R_0)$, $P_1 = (\Sigma_1, A_1, R_1)$ be stratifiable TSS's. Let $\Sigma_0 \oplus \Sigma_1$ be defined. If for all ground substitutions σ_0 and σ_1 and rules $r_0 \in R_0$ and $r_1 \in R_1$ such that*

- *ϕ is the conclusion of r_1,*
- *ψ is a positive premise of r_0, or $\psi = t \xrightarrow{a} t'$ and $t \not\xrightarrow{a}$ is a negative premise of r_0 and*
- *$\sigma_0(\psi) \neq \sigma_1(\phi)$*

then $P_0 \oplus P_1$ is a stratifiable TSS.

Proof. Assume that P_0 has stratification $S^0 : Tr(\Sigma_0, A_0) \to \alpha_0$ and that P_1 has stratification $S^1 : Tr(\Sigma_1, A_1) \to \alpha_1$. Construct a stratification S for $P_0 \oplus P_1$ as follows: define $U \subseteq Tr(\Sigma_0 \oplus \Sigma_1, A_0 \cup A_1)$ as the set of all literals that fit a premise of a rule $r_0 \in R_0$. If literal $\phi \in U$ then construct a literal $\bar{\phi}$ by replacing all subterms $f(\bar{u})$ for $f \in F_1$ in ϕ by a_0. As the label of ϕ is in A_0, $\bar{\phi} \in Tr(\Sigma_0, A_0)$ and thus $\bar{\phi}$ occurs in a stratum β in S^0. Define $S(\phi) = \beta$.
Assume $\phi \notin U$. If the label of ϕ is not in A_1 then $S(\phi) = \alpha_0$. If the label of ϕ is in A_1 then construct $\bar{\phi}$ from ϕ by replacing every subterm $f(\bar{u})$ in ϕ with $f \in \Sigma_0$ by a_1. Now $\bar{\phi} \in Tr(\Sigma_1, A_1)$. So it must hold that $\bar{\phi}$ is in a stratum β in S^1. Define $S(\phi) = \alpha_0 + \beta$. Now every literal $\phi \in Tr(\Sigma_0 \oplus \Sigma_1, A_0 \cup A_1)$ has a place in S.

We now check that S is a stratification of $P_0 \oplus P_1$. Take a rule $r \in R_0 \oplus R_1$. Suppose σ is a ground substitution and ψ is the conclusion, ϕ a positive premise (if present in $\sigma(r)$) and $t \not\xrightarrow{a}$ a negative premise (also if present) of $\sigma(r)$. We proceed by case analysis.

- $\psi \in U$. By the condition in this theorem ψ is not an instance of a conclusion in a rule R_1 and thus $r \in R_0$. Hence, for all $t' \in T(\Sigma_0 \oplus \Sigma_1) : \phi, t \xrightarrow{a} t' \in U$. ϕ, ψ and $t \xrightarrow{a} t'$ are related in S in the same way as $\bar{\phi}, \bar{\psi}$ and $t \xrightarrow{a} t'$ are related in S^0. As $\bar{\phi}, \bar{\psi}$ and $t \xrightarrow{a} t'$ are also instances of r for some σ' they satisfy the conditions for a proper stratification in S_0 and therefore ϕ, ψ and $t \xrightarrow{a} t'$ satisfy these conditions in S.
- $\psi \notin U$.
  - If ψ has a label $a \notin A_1$ then r cannot be a rule of R_1 and so $r \in R_0$. As ϕ and $t \xrightarrow{a} t'$ (for all t') are elements of U, ψ is in a strictly higher stratum than all its premises. Hence r satisfies the stratification condition in this case.
  - If ψ has a label in A_1 then $\psi \in S_{\alpha_0+\beta}$ if $\bar{\psi}$ is in stratum S^1_β. If $\phi \in U$ then ϕ is in a strictly lower stratum than ψ and if $t \xrightarrow{a} t' \in U$ then $t \xrightarrow{a} t'$ is in a strictly lower stratum than ψ. If $\phi \notin U$ and $\bar{\phi} \in S^1_\gamma$, then $S(\phi) = \alpha_0 + \gamma$ as the label of ϕ comes from A_1. If $t \xrightarrow{a} t' \notin U$ and $t \xrightarrow{a} t' \in S^1_{\gamma_{t'}}$ then $S(t \xrightarrow{a} t') = \alpha_0 + \gamma_{t'}$ because $a \in A_1$. Now as $\bar{\psi}$, $\bar{\phi}$ and $t \xrightarrow{a} t'$ are all instances of r for some substitution σ', $\gamma \leq \beta$ and $\gamma_{t'} < \beta$. Hence, ψ is in an equal or higher stratum than ϕ in S and $t \xrightarrow{a} t'$ is in a strictly lower stratum than ψ. This shows that also in the last case the stratifiability condition for r is satisfied.

□

4.6 Congruences induced by *ntyft/ntyxt*

In this section we show that if we define operators using the pure *ntyft/ntyxt*-format, then for image finite processes the trace congruence and completed trace congruence induced by this format are exactly (strong) bisimulation equivalence. First we give the definition of a trace congruence induced by a format and the definition of image finite processes. In figure 4.6 we show how we will then prove our result. The arrows denote set inclusion and 'IF' indicates that we need image finiteness.

Definition 4.6.1. Let $P = (\Sigma, A, R)$ be a stratifiable TSS and let $\longrightarrow_P$ be the transition relation associated with P. Let $t \in T(\Sigma)$. A sequence $a_1 \star ... \star a_n \in A^\star$ is a *(P-)trace* from t iff there are terms $t_1, ..., t_n \in T(\Sigma)$ for some $n \in \mathbb{N}$ such that $t \xrightarrow{a_1}_P t_1 \xrightarrow{a_2}_P ... \xrightarrow{a_n}_P t_n$. $Tr(t)$ is the set of all P-traces from t. Two process terms $t, t' \in T(\Sigma)$ are *trace equivalent with respect to* P iff $Tr(t) = Tr(t')$. This is also denoted as $t \equiv^T_P t'$.

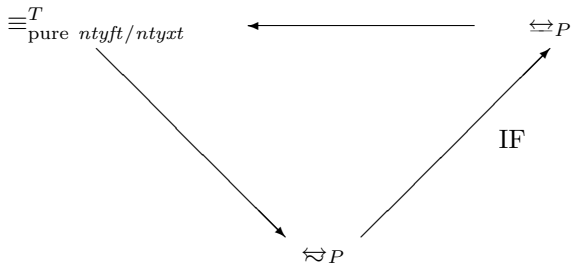


Figure 4.6: Inclusions among several process equivalences

Note that if two terms t and t' are bisimilar, then they are also trace equivalent.

Definition 4.6.2. Let $\mathcal{F}$ be some format of TSS rules. Let $P = (\Sigma, A, R)$ be a stratifiable TSS in $\mathcal{F}$ format. Two terms $t, t' \in T(\Sigma)$ are *trace congruent with respect to* $\mathcal{F}$ rules, notation $t \equiv^T_{\mathcal{F}} t'$, iff for every TSS $P' = (\Sigma', A', R')$ in $\mathcal{F}$ format which can be added conservatively to P and for every $\Sigma \oplus \Sigma'$-context $C[]$: $C[t] \equiv^T_{P \oplus P'} C[t']$.

Definition 4.6.3. Let $P = (\Sigma, A, R)$ be a stratifiable TSS. Let $\longrightarrow_P$ be the transition relation associated with P. $\longrightarrow_P$ is called *image finite* iff for all $t \in T(\Sigma)$ and $a \in A$ the set $\{u \mid t \xrightarrow{a}_P u\}$ is finite.

Definition 4.6.4. Let $P = (\Sigma, A, R)$ be a stratifiable TSS with associated transition relation $\longrightarrow_P$. A family of relations $R^n \subseteq T(\Sigma) \times T(\Sigma)$, for $n \in \mathbf{N}$, are called an *n-bounded bisimulation* relations iff:

- $R^0 = T(\Sigma) \times T(\Sigma)$,
- $\forall t, u \in T(\Sigma)\ t\, R^{n+1}\, u$ and $t \xrightarrow{a}_P t' \ \Rightarrow\ \exists u'\ u \xrightarrow{a}_P u'$ and $t' R^n u'$,
- $\forall t, u \in T(\Sigma)\ t\, R^{n+1}\, u$ and $u \xrightarrow{a}_P u' \ \Rightarrow\ \exists t'\ t \xrightarrow{a}_P t'$ and $t' R^n u'$.

Two process expressions $t, t' \in T(\Sigma)$ are *n-bounded bisimilar (for P)*, notation $t \leftrightsquigarrow^n_P t'$ iff there is an n-bounded bisimulation relation R^n such that $t\, R^n\, t'$. Two terms $t, t' \in T(\Sigma)$ are *bounded bisimilar for P*, notation $t \leftrightsquigarrow_P t'$, iff for all $n \in \mathbf{N}$ $t \leftrightsquigarrow^n_P t'$.

The following lemma gives a condition under which bounded bisimilar states are bisimilar.

Lemma 4.6.5. *Let $P = (\Sigma, A, R)$ be a stratifiable TSS such that $\longrightarrow_P$ is image finite. Let $t, u \in T(\Sigma)$. Then:*

$$t \leftrightsquigarrow_P u \quad \Leftrightarrow \quad t \leftrightarrows_P u.$$

Proof. "$\Leftarrow$" is trivial. See for "$\Rightarrow$" [15]. □

$B^0(x,y) \xrightarrow{yes} \delta$		1
$\dfrac{y \xrightarrow{a} y' \quad B^{n-1}(x',y') \xrightarrow{yes} z}{Q_a^n(x',y) \xrightarrow{ok} \delta}$	for $n > 0,\ a \in A$	2
$\dfrac{x \xrightarrow{a} x' \quad Q_a^n(x',y) \overset{ok}{\not\longrightarrow}}{B^n(x,y) \xrightarrow{no} \delta}$	for $n > 0,\ a \in A$	3
$\dfrac{B^n(x,y) \overset{no}{\not\longrightarrow} \quad B^n(y,x) \overset{no}{\not\longrightarrow}}{B^n(x,y) \xrightarrow{yes} \delta}$	for $n > 0$	4

Table 4.2: A bisimulation tester

We now give the basic definitions and lemmas to prove that $\equiv^T_{\text{pure } ntyft/ntyxt} \subseteq \leftrightsquigarrow_P$. The main component is the following test system. We show that this test system is stratifiable and that it can test equality between n-bounded bisimilar processes.

Definition 4.6.6. Let $P = (\Sigma, A, R)$ be a TSS. The *bisimulation tester of* P $P_T = (\Sigma_T, A_T, R_T)$ is a TSS with signature $\Sigma_T = (F_T, r_T)$ containing binary function names B^n and Q_a^n for all $n \in \mathbb{N}$, $a \in A$ and a constant δ. The labels of P_T are $A_T = A \cup \{ok, yes, no\}$. The rules in R_T are given in table 4.2.

The rules in table 4.2 are based on the following meaning of the transitions $\xrightarrow{yes}$, $\xrightarrow{no}$ and $\xrightarrow{ok}$:

- $B^n(x,y) \xrightarrow{yes} \delta$ if x and y are n-bounded bisimilar.
- $B^n(x,y) \xrightarrow{no} \delta$ $(n > 0)$ if x can perform a step that cannot be done by y such that the results are $(n-1)$-bounded bisimilar.
- $Q_a^n(x,y) \xrightarrow{ok} \delta$ $(n > 0)$ means that y can perform an a-step such that the result is $(n-1)$-bounded bisimilar with x.

The rules in table 4.2 just encode n-bounded bisimilarity. The negative premises model the universal quantifiers in definition 4.6.4.

Remark 4.6.7. The test system P_T is able to test equivalences between terms $t, u \in T(\Sigma)$. However, it cannot test processes over $T(\Sigma \oplus \Sigma_T)$. The reason for

this is that in rule 2 and 3 of table 4.2 $a \neq ok, yes, no$. If a would be allowed range over $A \cup \{ok, yes, no\}$, then it is impossible to give a stratification as is done in this paper.

Lemma 4.6.8. *Let $P = (\Sigma, A, R)$ be a TSS. Let P_T be the bisimulation tester of P. P_T is stratifiable.*

Proof. It is enough to show that P has a stratification. Construct a mapping $S : Tr(\Sigma_T, A_T) \to \omega$ as follows:

- for all $a \in A$ and $t, t' \in T(\Sigma_T)$ $S(t \xrightarrow{a} t') = 1$,
- for $n \in \mathbf{N}$ and $t, u, v \in T(\Sigma_T)$ $S(B^n(t,u) \xrightarrow{yes} v) = 2n + 1$,
- for $n \in \mathbf{N} - \{0\}$, $a \in A_T$ and $t, u, v \in T(\Sigma_T)$ $S(Q^n_a(t,u) \xrightarrow{ok} v) = 2n - 1$,
- for $n \in \mathbf{N} - \{0\}$ and $t, u, v \in T(\Sigma_T)$ $S(B^n(t,u) \xrightarrow{no} v) = 2n$.

It is straightforward to check that S is a stratification for P_T. □

Lemma 4.6.9. *Let $P = (\Sigma, A, R)$ be a stratifiable TSS in pure ntyft/ntyxt-format containing at least one constant in its signature. Furthermore, A must not contain the labels ok, no, yes and Σ must not contain function names B^n and Q^n_a for all $a \in A, n \in \mathbf{N}$. Let $t, u \in T(\Sigma)$ then*

$$B^n(t,u) \xrightarrow{yes} \delta \in \longrightarrow_{P \oplus P_T} \quad \Leftrightarrow \quad t \leftrightarrows^n_P u.$$

Proof. As $yes, no, ok \notin A$, conclusions of rules in R_T never fit a premise of rules in R. Furthermore, P and P_T are stratifiable and contain at least one constant in their signatures. Hence by theorem 4.5.8, $P \oplus P_T$ is stratifiable. So $P \oplus P_T$ has an associated transition relation $\longrightarrow_{P \oplus P_T}$. As a consequence of theorem 4.5.5 $P \oplus P_T$ is a conservative extension of P.
"$\Rightarrow$" Use induction on n. *Basis.* For $n = 0$ $t \leftrightarrows^n_P u$ for any $t, u \in T(\Sigma)$. Hence, the theorem holds in this case.
Induction. We have to show that (1):

If $B^{n+1}(t,u) \xrightarrow{yes} \delta \in \longrightarrow_{P \oplus P_T}$ and $t \xrightarrow{a} t' \in \longrightarrow_P$ then
$\exists u'$ s.t. $u \xrightarrow{a} u' \in \longrightarrow_P$ and $t' \leftrightarrows^n_P u'$,

and vice versa (2):

If $B^{n+1}(t,u) \xrightarrow{yes} \delta \in \longrightarrow_{P \oplus P_T}$ and $u \xrightarrow{a} u' \in \longrightarrow_P$ then
$\exists t'$ s.t. $t \xrightarrow{a} t' \in \longrightarrow_P$ and $t' \leftrightarrows^n_P u'$.

As $B^{n+1}(t,u) \xrightarrow{yes} \delta \in \longrightarrow_{P\oplus P_T}$ and $\longrightarrow_{P\oplus P_T}$ agrees with $P \oplus P_T$, it must be the case that using rule 4 $B^{n+1}(t,u) \xrightarrow{no}$ and $B^{n+1}(u,t) \xrightarrow{no}$ hold in $\longrightarrow_{P\oplus P_T}$. Therefore, it cannot be the case that the premises of rule 3 all hold with $\sigma(x) = t$, $\sigma(y) = u$. But we know that $t \xrightarrow{a} t' \in \longrightarrow_P$ and by conservativity also $t \xrightarrow{a} t' \in \longrightarrow_{P\oplus P_T}$. Hence for some v $Q_a^{n+1}(t',u) \xrightarrow{ok} v \in \longrightarrow_{P\oplus P_T}$. But then the premises of rule 2 must be true with $\sigma(y) = u$ and $\sigma(x') = t'$. Hence for some u' $u \xrightarrow{a} u' \in \longrightarrow_{P\oplus P_T}$ and $B^n(t',u') \xrightarrow{yes} \delta \in \longrightarrow_{P\oplus P_T}$. By conservativity $u \xrightarrow{a} u' \in \longrightarrow_P$. With the induction hypothesis $t' \leftrightarrows_P^n u'$. We can show (2) in the same way. Hence if $B^{n+1}(t,u) \xrightarrow{yes} \delta \in \longrightarrow_{P\oplus P_T}$ then $t \leftrightarrows_P^{n+1} u$.

"$\Leftarrow$" Again, we use induction on n. *Basis.* If $n = 0$, the theorem is trivial as $B^0(t,u) \xrightarrow{yes} \delta \in \longrightarrow_{P\oplus P_T}$ for all $t,u \in T(\Sigma)$.
Induction. Suppose $t \leftrightarrows_P^{n+1} u$. We show that $B^{n+1}(t,u) \xrightarrow{yes} \in \longrightarrow_{P\oplus P_T}$. By rule 4 it is sufficient to show that $B^{n+1}(t,u) \xrightarrow{no}$ and $B^{n+1}(u,t) \xrightarrow{no}$ hold in $\longrightarrow_{P\oplus P_T}$. This means that we have to show that rule 3 can never be applied, i.e. either (3): $t \xrightarrow{a} t'$ or $Q_a^{n+1}(t',u) \xrightarrow{ok}$ nor (4): $u \xrightarrow{a} u'$ or $Q_a^{n+1}(u',t) \xrightarrow{ok}$ for any $a \in A$ holds in $\longrightarrow_{P\oplus P_T}$. Suppose for some $a \in A$ $t \xrightarrow{a}$ holds in $\longrightarrow_{P\oplus P_T}$. Then (3) trivially does not hold. Now suppose $t \xrightarrow{a} t' \in \longrightarrow_{P\oplus P_T}$ for some t'. As P_T conservatively extends P, $t \xrightarrow{a} t' \in \longrightarrow_P$. Then using $t \leftrightarrows_P^{n+1} u$ $\exists u' \in T(\Sigma)$ $u \xrightarrow{a} u' \in \longrightarrow_P$ and $t' \leftrightarrows_P^n u'$. By conservativity $u \xrightarrow{a} u' \in \longrightarrow_{P\oplus P_T}$. Using the induction hypothesis $B^n(t',u') \xrightarrow{yes} \delta \in \longrightarrow_{P\oplus P_T}$. Applying rule 2 yields $Q_a^{n+1}(t',u) \xrightarrow{ok} \delta \in \longrightarrow_{P\oplus P_T}$ and hence $Q_a^{n+1}(t',u) \xrightarrow{ok}$ does not hold in $\longrightarrow_{P\oplus P_T}$. We can prove (4) in the same way. □

The following theorem relates all notions.

Theorem 4.6.10. *Let $P = (\Sigma, A, R)$ be a stratifiable TSS in pure ntyft/ntyxt-format containing at least one constant in its signature. Furthermore, $\longrightarrow_P$ is image finite, A does not contain labels ok, no, yes and Σ does not contain function names B^n, Q_a^n for all $a \in A, n \in \mathbb{N}$.*

$$t \equiv^T_{\text{pure } ntyft/ntyxt} u \quad \Leftrightarrow \quad t \leftrightarrows_P u \quad \Leftrightarrow \quad t \leftrightarrows_P u$$

Proof. Suppose $t \leftrightarrows_P u$. Let $P' = (\Sigma', A', R')$ be a TSS in pure *ntyft/ntyxt*-format such that $P \oplus P'$ is a conservative extension of P. Then $t \leftrightarrows_{P\oplus P'} u$. By the congruence theorem, for any $\Sigma \oplus \Sigma'$-context C $C[t] \leftrightarrows_{P\oplus P'} C[u]$. Hence, $t \equiv^T_{\text{pure } ntyft/ntyxt} u$.

Suppose $t \not\leftrightarrows_P u$. This means that for some $n \in \mathbb{N}$ $t \not\leftrightarrows_P^n u$. Construct the context $B^n(t,[\,])$. Now by lemma 4.6.9 $B^n(t,u) \xrightarrow{yes}$ holds in $\longrightarrow_{P\oplus P_T}$ while $B^n(t,t) \xrightarrow{yes} \delta \in \longrightarrow_{P\oplus P_T}$. Hence, $t \not\equiv^T_{\text{pure } ntyft/ntyxt} u$ or in other words:

$$t \equiv^T_{\text{pure } ntyft/ntyxt} u \quad \Rightarrow \quad t \leftrightarrows_P u.$$

The last case, $t \leftrightsquigarrow_P u \Rightarrow t \leftrightarrows_P u$, follows directly from lemma 4.6.5. □

The condition that $ok, no, yes \notin A$ and B^n, Q^n_a are not in Σ is not a real restriction. It can be circumvented by simply renaming labels and function names. The requirement that Σ contains at least one constant is also natural: without such a constant there are no terms $t \in T(\Sigma)$ and hence a bisimulation tester would not be useful.

The bisimulation tester uses an infinite number of function names. For every $n \in \mathbf{N}$ and $a \in A$ there are binary operators B^n and Q^n_a. It is natural to ask whether a test system with a finite number of binary operators can be formulated. Here such a test system is given. This test system has as additional property that if the number of labels in a tested system is finite, then there are only a finite number of rules necessary.

Definition 4.6.11. Let $P = (\Sigma, A, R)$ be a TSS with a countable set of labels A. Assume that there is a function $n : A \to \mathbf{N}$ that gives a unique number for each label, satisfying that if for $a \in A$ $n(a) = m > 0$ then $\exists b \in A$ $n(b) = m - 1$. The *finite bisimulation tester* $P_{FT} = (\Sigma_{FT}, A_{FT}, R_{FT})$ contains constants $0, 1$ and δ, unary function names S and S_0, a ternary function name B and a quaternary function name Q. The labels in P_{FT} are given by $A_{FT} = A \cup \bar{A} \cup \{ok, yes, no, 0, 1\}$. Here $\bar{A} = \{\bar{a} \mid a \in A\}$. The definition of n is extended to $\bar{A}$ by $n(\bar{a}) = n(a)$. The rules in R_{FT} are given in table 4.3. Here, $l, l', n, n', x, x', y, y'$ are variables. a ranges over A and b, c range over $\bar{A}$. $S_0^{n(a)}(1)$ is an abbreviation for $n(a)$ applications of S_0 to 1.

The main difference between P_T and P_{FT} is that labels and numbers do not occur any more as sub- and superscripts at Q and B, but they are coded by zeroes and successor functions and included in the list of arguments. We have the same results for P_{FT} as for P_T. We only give here the main lemmas and we omit the proofs. With these results it can be shown in exactly the same way as in the proof of theorem 4.6.10 that P_{FT} is also powerful enough to distinguish between non bisimilar processes.

Lemma 4.6.12. *Let $P = (\Sigma, A, R)$ be a TSS with a countable set of labels A. The finite bisimulation tester P_{FT} of P is stratifiable.*

Lemma 4.6.13. *Let $P = (\Sigma, A, R)$ be a stratifiable TSS in pure ntyft/ntyxt-format with a countable set of labels A not containing labels $yes, no, ok, 0, 1$ and at least one constant in Σ. Function names 0, S, 1, S_0, B, Q must not occur in Σ. Let $t, u \in T(\Sigma)$. $S^n(0)$ is an abbreviation for n applications of S on 0. Then:*

$$B(S^n(0), t, u) \xrightarrow{yes}_{P \oplus P_{FT}} \delta \quad \Leftrightarrow \quad t \leftrightarrows^n_P u.$$

$0 \xrightarrow{0} \delta$		1
$S(x) \xrightarrow{1} x$		2
$1 \xrightarrow{b} \delta$	for $n(b) = 0$	3
$\dfrac{x \xrightarrow{b} x'}{S_0(x) \xrightarrow{c} \delta}$	if $n(c) = n(b) + 1$	4
$\dfrac{n \xrightarrow{0} n'}{B(n,x,y) \xrightarrow{yes} \delta}$		5
$\dfrac{l \xrightarrow{\bar{a}} l' \quad y \xrightarrow{a} y' \quad B(n,x',y') \xrightarrow{yes} z}{Q(n,l,x',y) \xrightarrow{ok} \delta}$	for $a \in A$	6
$\dfrac{n \xrightarrow{1} n' \quad x \xrightarrow{a} x' \quad Q(n', S_0^{n(a)}(1), x', y) \not\xrightarrow{ok}}{B(n,x,y) \xrightarrow{no} \delta}$	for $n > 0,\ a \in A$	7
$\dfrac{B(n,x,y) \not\xrightarrow{no} \quad B(n,y,x) \not\xrightarrow{no}}{B(n,x,y) \xrightarrow{yes} \delta}$	for $n > 0$	8

Table 4.3: A finite bisimulation tester

	trace congruence	completed trace congruence
De Simone-format	trace equivalence	failure equivalence
positive GSOS-format	simulation equivalence	2/3 bisimulation
GSOS-format	2/3 bisimulation	2/3 bisimulation
pure *tyft/tyxt*-format	simulation equivalence	2-nested simulation equivalence
pure *ntyft/ntyxt*-format	bisimulation	bisimulation

Table 4.4: An overview of (completed) trace congruences

4.7 An overview of trace and completed trace congruences

There are nowadays several different formats of rules for describing a Plotkin style operational semantics. All these formats induce their own trace and completed trace congruences. Below in table 4.4 we give an overview of the main results. We do not explicitly define all equivalence notions, but we confine ourselves to giving references. The first column describes the different formats for the rules. The pure *ntyft/ntyxt*-format is the most extensive. All other formats are restricted versions of the pure *ntyft/ntyxt*-format. The pure *tyft/tyxt*-format [16] can be obtained from the pure *ntyft/ntyxt*-format by not allowing negative premises in the rules. The GSOS-format [10] has been defined in example 4.3.1. It is a simplification of the pure *ntyft*-format in the sense that rules in GSOS-format only have conclusions of the form $f(x_1, ..., x_{r(f)}) \stackrel{a}{\longrightarrow} t$ and premises of the form $x_i \stackrel{a_i}{\longrightarrow} x_i'$ for $1 \leq i \leq r(f)$ and $x_j \stackrel{b_j}{\not\longrightarrow}$ for $1 \leq j \leq r(f)$. In example 4.3.1 it has been shown that a TSS in GSOS-format has a unique associated transition relation.
The positive GSOS-format [16] is almost equal to the GSOS-format, the only difference being that rules in the positive GSOS-format do not have negative premises. A typical example of a rule in positive GSOS format is:

$$\frac{x \stackrel{a}{\longrightarrow} x_1' \quad x \stackrel{b}{\longrightarrow} x_2'}{f(x) \stackrel{c}{\longrightarrow} g(x, x_1', x_1')}.$$

One can clearly see that variables may be used more than once in the source of the premises or the target of the conclusion. This is called *copying* [1]. The positive GSOS-format is not only more restricted than the GSOS-format, but also every rule satisfying the positive GSOS-format is in the pure *tyft/tyxt*-format (see figure 4.1).
The oldest format is the De Simone-format [14]. It is equal to the positive GSOS-format except that it does not allow copying. Every variable in the left hand side of the conclusion may only occur once in the right hand side of the conclusion or in the left hand side of a premise. Every variable in the right hand side of a premise may appear only once in the right hand side of the conclusion.
The second and third column of table 4.4 give the trace and completed trace con-

gruences belonging to these formats. The notion of completed trace congruences is:

Definition 4.7.1. Let $P = (\Sigma, A, R)$ be a TSS with associated transition relation $\longrightarrow_P$. Let $t \in T(\Sigma)$. t is a *deadlocked process*, notation $t\not\longrightarrow$, iff there are no $u \in (\Sigma)$ and $a \in A$ with $t \xrightarrow{a}_P u$. A sequence $a_1 \star ... \star a_n \in A^\star$ is a *completed trace* of t iff there are process terms $t_1, ..., t_n \in T(\Sigma)$ such that $t \xrightarrow{a_1}_P t_1 \xrightarrow{a_2}_P ... \xrightarrow{a_n}_P t_n\not\longrightarrow$. $CT(t)$ is the set of all completed traces of t. Two process terms $t, u \in T(\Sigma)$ are *completed trace equivalent for* P if $CT(t) = CT(u)$. This is denoted as $t \equiv_P^{CT} u$.

The notion of *completed trace congruence* can be obtained by replacing 'trace' by 'completed trace', $\equiv_{\mathcal{F}}^{T}$ by $\equiv_{\mathcal{F}}^{CT}$ and $\equiv_P^T$ by $\equiv_P^{CT}$ in definition 4.6.2.

The trace and completed trace congruences for the De Simone-format follow directly from an important result of R. de Simone [14]: All operators definable in the De Simone-format can also be defined using *architectural expressions* over MEIJE-SCCS. It is a well known result that trace equivalence is a congruence in MEIJE-SCCS. From this it follows immediately that the trace congruence is trace equivalence. Furthermore, an established result is that the completed trace congruence is failure trace equivalence. For all other results, we refer to [16] where all completed trace congruences, except for the pure *ntyft/ntyxt*-format, are given. The notion of 2/3-bisimulation was first mentioned in [21] and simulation equivalence and 2-nested simulation equivalence are defined in [16]. The trace congruences for positive GSOS and GSOS are not published anywhere. However, with the help of the lemmas in [16] one can prove the results. In [16] it is shown that the trace congruence for the pure *tyft/tyxt*-format is simulation equivalence.

References

[1] S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[2] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, chapter 10, pages 495–574. North-Holland, 1990.

[3] E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, G. Reggio, and E. Zucca. *The Trial Definition of Ada, Deliverable 7 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815 Ada.* 1986.

[4] D. Austry and G. Boudol. Algèbre de processus et synchronisations. *Theoretical Computer Science*, 30(1):91–131, 1984.

[5] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.

[6] J.C.M. Baeten and J.A. Bergstra. Processen en procesexpressies. *Informatie*, 30(3):177–248, 1988. In Dutch.

[7] 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.

[8] J.A. Bergstra. Put and get, primitives for synchronous unreliable message passing. Logic Group Preprint Series Nr. 3, CIF, State University of Utrecht, 1985.

[9] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Report 842, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1988. To appear in *Science of Computer Programming.*

[10] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: preliminary report. In *Proceedings* 15^{th} *ACM Symposium on Principles of Programming Languages,* San Diego, California, pages 229–239, 1988.

[11] T. Bolognesi, F. Lucidi, and S. Trigila. From timed Petri nets to timed LOTOS. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Proceedings* 10^{th} *IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification,* Ottawa, pages 395–408, 1990.

[12] J. Camilleri and G. Winskel. CCS with priority choice. In *Proceedings* 6^{th} *Annual Symposium on Logic in Computer Science,* Amsterdam, The Netherlands, pages 246–255. IEEE Computer Society Press, 1991.

[13] R. Cleaveland and M. Hennessy. Priorities in process algebra. In *Proceedings* 3^{th} *Annual Symposium on Logic in Computer Science,* Edinburgh, pages 193–202. IEEE Computer Society Press, 1988.

[14] R. De Simone. Higher-level synchronising devices in MEIJE–SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[15] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 336–347. Springer-Verlag, 1987.

[16] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings* 16^{th} *ICALP,* Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation.*

[17] M. Hennessy and T. Regan. A temporal process algebra. Report 2/90, Computer Science Department, University of Sussex, 1990.

[18] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour*, 1987. ISO/TC97/SC21/N DIS8807.

[19] A.S. Klusener. Completeness in realtime process algebra. Technical Report CS-R9106, CWI, Amsterdam, 1991.

[20] K.G. Larsen. Modal specifications. Technical Report R 89-09, Institute for Electronic Systems, The University of Aalborg, February 1989.

[21] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Proceedings* 16^{th} *ACM Symposium on Principles of Programming Languages,* Austin, Texas, pages 344–352, 1989.

[22] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[23] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[24] R. Milner. *Communication and concurrency.* Prentice Hall International, 1989.

[25] F. Moller. *Axioms for concurrency.* PhD thesis, Department of Computer Science, University of Edinburgh, 1989. CST-59-89.

[26] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. Report RT - C26, IMAG, Laboratoire de Génie informatique, Grenoble, 1990.

[27] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[28] I.C.C. Phillips. CCS with broadcast stability. Unpublished manuscript.

[29] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[30] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II,* Garmisch, pages 199–225, Amsterdam, 1983. North-Holland.

[31] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In W. Brauer, editor, *Proceedings* 12^{th} *ICALP,* Nafplion, volume 194 of *Lecture Notes in Computer Science*, pages 15–32. Springer-Verlag, 1985.

[32] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

5

The Meaning of Negative Premises in Transition System Specifications

(Roland Bol & Jan Friso Groote)

The stratification technique of the previous chapter is not always satisfactory, because, as is shown in this paper, there are examples it cannot deal with. In this paper we consider the problem of negative premises again, but put everything in a broader context[1]. We present a general theory for the use of negative premises in the rules of Transition System Specifications (TSS's). We formulate a criterion that should be satisfied by a TSS in order to be meaningful, i.e. to unequivocally define a transition relation. We also propose various techniques for proving that a TSS satisfies this criterion. Among these is stratification, taken from the previous chapter, and a stronger reduction technique. Both the criterion and the techniques originate from logic programming [11, 10] to which TSS's are close. In the last section we provide an extensive comparison between them.

As in the article in the previous chapter, we show that the bisimulation relation induced by a TSS is a congruence, provided that it is in *ntyft/ntyxt*-format and can be proved meaningful using the reduction technique. We also extend the conservativity theorems of [14] and the previous chapter considerably. As a running example, we study the combined addition of priorities and abstraction to Basic Process Algebra (BPA). Under some reasonable conditions we show that this TSS is indeed meaningful, which could not be shown by the techniques presented in the article in chapter 4. Finally, we provide a sound and complete axiomatisation for this example.

[1]Although this paper extends the previous chapter in various aspects, it is completely self contained and it can be read independently of it. But wherever appropriate, we have put references to the last chapter, in order to ease a comparison

5.1 Introduction

In the article in chapter 4 stratifiable TSS's are studied. In this article we provide a way to treat TSS's with negative premises in general and we study some of the consequences of this treatment. The fundamental problem of negative premises in TSS's is that they cannot be proved in the same way positive premises can. In order to overcome this problem, we resort to a non-classical treatment of negation, similar to default logic [3, 23] and logic programming [11]. Without negative premises the notion of proof is standard. With negative premises we may only use the rules of which the negative premises hold. A negative premise holds by default, that is unless the opposite can be proved. Now suppose $\longrightarrow$ contains all transitions that can be proved in this way. Then it must satisfy:

> $\longrightarrow$ is the set of transitions that are provable by those rules of which the negative premises are consistent with $\longrightarrow$.

Following [11], we call such a transition relation *stable* for the TSS.

It is possible that a TSS has zero, one or more stable transition relations. If a TSS P has exactly one stable transition relation then we propose to define the semantics of a TSS P as this relation and we say that this relation is *associated with* P. If a TSS has zero or more than one stable transition relations, it is hard to imagine that any specific transition relation can be associated with it on reasonable grounds. That is, unless one is prepared to associate with a TSS a transition relation that is not decisive about all transitions. But this is not considered appropriate for the field of operational semantics.

In general it is difficult to show that a TSS has a unique stable transition relation. However, some techniques have been developed to do so.

The first technique, called *stratification*, has been presented in the article in the previous chapter. It is based on the notion of local stratification in logic programming [21]. In this article we show that the transition relation associated with a stratified TSS is indeed the unique stable transition relation for it. This also implies the same fact for positive TSS's and TSS's in the so-called GSOS-format [5], as they are stratified.

Stratification is an intuitively appealing technique, and quite easy to use, but it is not always strong enough. Here, we introduce a more powerful technique, based on *well-founded models* in logic programming [10, 22]. This technique, which we call *reduction*, is more powerful than stratification, but also more difficult to use. The two techniques can be amalgamated, using reduction when necessary and stratification when possible. This is demonstrated on our running example combining unguarded recursion, abstraction and priorities, showing that under some reasonable conditions a transition relation can be associated with it.

A desirable property for a TSS is that the strong bisimulation equivalence induced by it [17, 18] is a congruence. In [14] the *tyft/tyxt*-format was introduced, as a syntactical condition on TSS's that guarantees this property for positive TSS's. In the article in chapter 4 this condition has been generalised to the *ntyft/ntyxt*-format for stratified TSS's. Here we show that the same condition is

sufficient for all TSS's for which the reduction technique works. In contrast we show that the condition is not sufficient for TSS's having an associated transition relation that is not produced by reduction.

It can be useful to enrich a given language with additional language constructs. In order to do this in a systematic way, the *sum* of two TSS's has been introduced [14]. The sum of two TSS's P_0 and P_1 is called a conservative extension of P_0 if certain relevant properties of terms over the signature of P_0 are preserved. In chapter 4 syntactical conditions on stratified TSS's were given ensuring that their sum is conservative. Here we generalise these conditions considerably and we extend them to TSS's for which the reduction technique works.

Throughout the paper we use an example to illustrate these techniques: a TSS specifying the operational semantics of Basic Process Algebra (BPA) extended with priorities [1] and abstraction [12, 17]. We show using reduction and stratification that this TSS is meaningful. In section 5.10 we give a sound and complete axiomatisation of strong bisimulation equivalence induced by this TSS. It turns out that most of the standard techniques for positive TSS's can still be used.

5.2 Preliminaries

In this section we provide the basic concepts of this paper: *transition relations* and *Transition System Specifications* (*TSS's*). An example of a TSS is given in which priorities and abstraction are integrated in BPA. This example will serve as a running example throughout this paper.

We assume the presence of an infinite set V of *variables* with typical elements $x, y, z....$

Definition 5.2.1. A *(single sorted) signature* is a structure $\Sigma = (F, rank)$ where:

- F is a set of *function names* disjoint with V,
- $rank : F \to \mathbb{N}$ is a *rank function* which gives the arity of a function name; if $f \in F$ and $rank(f) = 0$ then f is called a *constant name*.

Let $W \subseteq V$ be a set of variables. The set of Σ-*terms* over W, notation $T(\Sigma, W)$, is the least set satisfying:

- $W \subseteq T(\Sigma, W)$,
- if $f \in F$ and $t_1, ..., t_{rank(f)} \in T(\Sigma, W)$, then $f(t_1, ..., t_{rank(f)}) \in T(\Sigma, W)$.

$T(\Sigma, \emptyset)$ is abbreviated by $T(\Sigma)$; elements from $T(\Sigma)$ are called *closed* or *ground terms*. $\mathbb{T}(\Sigma)$ is used to abbreviate $T(\Sigma, V)$, the set of *open terms*. Clearly, $T(\Sigma) \subset \mathbb{T}(\Sigma)$. $Var(t) \subseteq V$ is the set of variables in a term $t \in \mathbb{T}(\Sigma)$. A *substitution* σ is a mapping in $V \to \mathbb{T}(\Sigma)$. A substitution σ is extended to a mapping $\sigma : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ in a standard way ($f \in F$ and $t_1, ..., t_{rank(f)} \in \mathbb{T}(\Sigma)$):

- $\sigma(f(t_1, ..., t_{rank(f)})) \;=\; f(\sigma(t_1), ..., \sigma(t_{rank(f)}))$.

A substitution is *closed* (or *ground*) if it maps all variables onto closed terms.

A transition relation prescribes what activities, represented by labeled transitions, can be performed by terms over some signature. Generally, the signature represents a programming or a specification language and the terms are programs. The transition relation models the operational behaviour of these terms.

Definition 5.2.2. Let Σ be a signature and A a set of labels. A (labeled) *transition relation* is a subset $\longrightarrow$ of $Tr(\Sigma, A)$ where $Tr(\Sigma, A) = T(\Sigma) \times A \times T(\Sigma)$. Elements (t, a, t') of a transition relation are written as $t \xrightarrow{a} t'$.

A transition relation is often defined by means of a *Transition System Specification* (*TSS*). PLOTKIN [15, 19] defended the use of TSS's to give an operational semantics, and therefore a TSS is sometimes called an *operational semantics in Plotkin style*. The term TSS was first coined in [14] for a system in which rules had only positive premises. Negative premises were added in the paper in the previous chapter.

Definition 5.2.3. A *TSS* (*Transition System Specification*) is a triple $P = (\Sigma, A, R)$ with Σ a signature, A a set of labels and R a set of rules of the form:

$$\frac{\{t_k \xrightarrow{a_k} t'_k \mid k \in K\} \cup \{t_l \overset{b_l}{\not\longrightarrow} \mid l \in L\}}{t \xrightarrow{a} t'}$$

with K, L (possibly infinite) index sets, $t_k, t'_k, t_l, t, t' \in \mathbb{T}(\Sigma)$, $a_k, b_l, a \in A$ ($k \in K$, $l \in L$). An expression of the form $t \xrightarrow{a} t'$ is called a (*positive*) *literal*. $t \overset{a}{\not\longrightarrow}$ is called a *negative literal*. φ, ψ, χ are used to range over literals. For a literal ψ, $source(\psi)$ denotes the term at the left hand side of ψ and, if ψ is positive, $target(\psi)$ denotes the term at the right hand side. For any rule $r \in R$ the literals above the line are called the *premises* of r, notation $prem(r)$, and the literal below the line is called the *conclusion* of r, denoted as $conc(r)$. Furthermore, we write $pprem(r)$ for the set of positive premises of r and $nprem(r)$ for the set of negative premises of r. A rule r is called *positive* if there are no negative premises, i.e. $nprem(r) = \emptyset$. A TSS is called *positive* if it has only positive rules. A rule is called an *axiom* if its set of premises is empty. An axiom $\dfrac{\emptyset}{t \xrightarrow{a} t'}$ is often written as $t \xrightarrow{a} t'$. The notions 'substitution', 'Var' and 'closed' extend to literals and rules as expected.

Throughout this paper we use the following Transition System Specification scheme to illustrate the techniques we introduce. It describes the semantics of a small basic process language extended with priorities and abstraction. This combination has not been studied before due to the technical complications that are involved. Priorities are investigated in [1, 2, 7, 8]. We follow the line set out by BAETEN, BERGSTRA and KLOP [1] who introduced a priority operator θ. For abstraction we take observation equivalence as introduced by MILNER [17] although technically we follow VAN GLABBEEK [12]. We base our example on

ϵ :	R1:	$\epsilon \xrightarrow{\surd} \delta$	
a :	R2:	$a \xrightarrow{a} \epsilon$ if $a \in Act_\tau$	
$+$:	R3.1:	$\dfrac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	R3.2: $\dfrac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$
$\cdot$:	R4.1:	$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$ if $a \in Act_\tau$	R4.2: $\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$
θ :	R5.1:	$\dfrac{x \xrightarrow{a} x' \quad \forall b > a \; x \overset{b}{\not\rightarrow}}{\theta(x) \xrightarrow{a} \theta(x')}$ $a \in Act_\tau$	R5.2: $\dfrac{x \xrightarrow{\surd} x'}{\theta(x) \xrightarrow{\surd} \theta(x')}$
$\triangleleft$:	R6.1:	$\dfrac{x \xrightarrow{a} x' \quad \forall b > a \; y \overset{b}{\not\rightarrow}}{x \triangleleft y \xrightarrow{a} x'}$ $a \in Act_\tau$	R6.2: $\dfrac{x \xrightarrow{\surd} x'}{x \triangleleft y \xrightarrow{\surd} x'}$
τ_I :	R7.1:	$\dfrac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{a} \tau_I(x')}$ if $a \notin I$	R7.2: $\dfrac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')}$ $a \in I$
rec:	R8:	$\dfrac{t_X \xrightarrow{a} y}{X \xrightarrow{a} y}$ if $X \Leftarrow t_X \in E$	
τ:	R9.1:	$a \xrightarrow{a} \tau$ if $a \in Act_\tau$	
	R9.2:	$\dfrac{x \xrightarrow{\tau} y \quad y \xrightarrow{a} z}{x \xrightarrow{a} z}$	R9.3: $\dfrac{x \xrightarrow{a} y \quad y \xrightarrow{\tau} z}{x \xrightarrow{a} z}$

Table 5.1: $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities ($a \in Act_{\tau\surd}$, $b \in Act_\tau$)

$\mathrm{BPA}_{\delta\epsilon\tau}$, *Basic Process Algebra* with τ, ϵ and δ as introduced in [14], and extend it with recursion and priorities.

Example 5.2.4 *(*$\mathrm{BPA}_{\delta\epsilon\tau}$ *with priorities).* We assume that we have a given set Act of actions that represent the basic activities that can be performed by processes. $Act_\tau = Act \cup \{\tau\}$ is a set of actions containing the symbol τ representing internal or hidden activity. Moreover, we assume a partial ordering $<$ on Act_τ, which we call the *priority relation*: actions higher in the ordering have a higher priority than actions lower in the ordering. We assume that $<$ is backwardly well-founded, i.e. the inverse of $<$ constitutes a well-founded ordering.

Our signature contains a constant a for each action $a \in Act_\tau$. Moreover, we have two special constants δ and ϵ. δ is called *inaction* (or *deadlock*) and it represents the process that cannot do anything at all. In particular, δ cannot terminate. ϵ is called the *empty process* which cannot do anything but terminate.

Two basic operators compose smaller into larger processes: *sequential composition* is written as '$\cdot$' and *alternative composition* is denoted by $+$. We often leave out the '$\cdot$' and assume that '$\cdot$' binds stronger than $+$.

Actions can be abstracted away: for all $I \subseteq Act$ the unary *abstraction operator* τ_I performs this task by renaming all actions in I to τ.

For recursion it is assumed that there is some given set Ξ of *process names*. Each process name $X \in \Xi$ is treated as a constant in the signature. Furthermore, we assume that a set E of *process declarations* is given. For each process name X in Ξ there is a declaration $X \Leftarrow t_X \in E$ where t_X is a closed term over the signature. Terms that do not contain process names are called *recursion free*.

The remaining operators in the signature deal with priorities. The *priority operator* θ acts as a sieve: $\theta(x)$ only allows those actions from x that have highest priority. For the axiomatisation of $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities, which is given in section 5.10, we need the *unless* operator $\lhd$, which was introduced in [1]. This operator is applied on two operands and only allows an action in its left hand side provided the right hand side cannot do any action with higher priority.

When $(Act_\tau, <)$ and (Ξ, E) are fixed, we obtain a TSS which is an *instance* of $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities. Such an instance will be denoted as $P_\theta = (\Sigma_\theta, A_\theta, R_\theta)$. The signature $\Sigma_\theta = (F_\theta, rank_\theta)$ is described above. The labels in A_θ are exactly those in Act_τ together with one special symbol $\surd$ which is used to signal termination. If a process term t can perform a $\surd$-step, i.e. $t \xrightarrow{\surd} t'$, this means that t has an option to terminate.

The rules in R_θ are given in table 5.1. Here, the action a ranges over $Act_{\tau\surd} = Act_\tau \cup \{\surd\}$ and b ranges over Act_τ. In rules R5.1 and R6.1 we use the notation $\forall b > a\ x \overset{b}{\not\rightarrow}$ which means that for all b with higher priority than a, there is a negative premise $x \overset{b}{\not\rightarrow}$. Rule R5.1 is intuitively appealing. It says that $\theta(x)$ may do an a-action if x can do this action and x cannot do any action with higher priority. But there is a snag in it. Due to the negative premises, it is not at all straightforward to see that P_θ defines a transition relation. In fact, in example 5.4.8 we will present a case in which it does not make sense at all.

Rules R9.1-R9.3 model the properties of τ. R9.2 and R9.3 say that whenever an action a is observed in some time interval, numerous unobservable τ-actions can also happen during the same time, both before and after a. Rule R9.1 says that if an action a is observed, some internal activity may exist before the next action can take place.

We think that the remaining rules are self explanatory, although we like to point out that rule R4.2 makes use of a process that explicitly signals termination.

5.3 Transition relations for TSS's

We have introduced TSS's as a formalism for specifying transition relations. Thus a most fundamental question is which transition relation is actually defined by a TSS. In this section we outline some answers proposed in the literature for several classes of TSS's. Then we show that these techniques are not capable of handling our running example satisfactorily. In the next sections we show how to solve this problem.

As a first step, a link between the transitions in a transition relation and the literals in TSS's is established.

Definition 5.3.1. Let $\longrightarrow$ be a transition relation. A positive ground literal ψ *holds* in $\longrightarrow$ or ψ is *valid* in $\longrightarrow$, notation $\longrightarrow\models \psi$, if the transition $\psi \in \longrightarrow$. A negative ground literal $t\stackrel{a}{\not\longrightarrow}$ *holds* in $\longrightarrow$, notation $\longrightarrow\models t\stackrel{a}{\not\longrightarrow}$, if for no $t' \in T(\Sigma)$ the transition $t \stackrel{a}{\longrightarrow} t' \in \longrightarrow$. For a set of literals Ψ, we write $\longrightarrow\models \Psi$ iff $\forall \psi \in \Psi$: $\longrightarrow\models \psi$.

Remark 5.3.2. Suppose we have two transition relations $\longrightarrow_1$ and $\longrightarrow_2$ such that $\longrightarrow_1 \subseteq \longrightarrow_2$. For any set of positive literals Ψ it is clear that $\longrightarrow_1\models \Psi$ implies $\longrightarrow_2\models \Psi$. However, if Ψ is a set of negative literals, then $\longrightarrow_2\models \Psi$ implies $\longrightarrow_1\models \Psi$. We shall often use this kind of reasoning.

What is the transition relation defined by a TSS? At least one may require that a transition relation associated with a TSS P obeys the rules of P, i.e. if the premises of a ground instance of a rule in P are valid in $\longrightarrow$, then the conclusion is also valid in $\longrightarrow$. (In terms of logic: the rules of P, interpreted as implications, are true in $\longrightarrow$).

Definition 5.3.3. Let $P = (\Sigma, A, R)$ be a TSS and let $\longrightarrow \subseteq Tr(\Sigma, A)$ be a transition relation. $\longrightarrow$ is a *model* of P if:

$$\psi \in \longrightarrow \quad \Leftarrow \quad \exists r \in R \text{ and } \exists \sigma : V \to T(\Sigma) \text{ such that : } \begin{cases} \longrightarrow\models prem(\sigma(r)) \text{ and} \\ conc(\sigma(r)) = \psi. \end{cases}$$

On the other hand, a transition ψ should not be incorporated in the transition relation $\longrightarrow$ of a TSS P unless there is a good reason to do so, namely a rule in P with valid premises in $\longrightarrow$ concluding ψ.

Definition 5.3.4. Let $P = (\Sigma, A, R)$ be a TSS. Let $\longrightarrow \subseteq Tr(\Sigma, A)$ be a transition relation. $\longrightarrow$ is *supported* by P if:

$$\psi \in \longrightarrow \quad \Rightarrow \quad \exists r \in R \text{ and } \exists \sigma : V \to T(\Sigma) \text{ such that : } \begin{cases} \longrightarrow \models prem(\sigma(r)) \text{ and} \\ conc(\sigma(r)) = \psi. \end{cases}$$

Combining the previous definitions, we get:

Definition 5.3.5. Let $P = (\Sigma, A, R)$ be a TSS. Let $\longrightarrow \subseteq Tr(\Sigma, A)$ be a transition relation. $\longrightarrow$ is a *supported model* of P if $\longrightarrow$ is supported by P and $\longrightarrow$ is a model of P.

The notion of $\longrightarrow$ being a supported model of P was introduced in [5] as '$\longrightarrow$ *agrees with* P' (see also definition 4.2.3. Although the transition relation associated with a TSS should certainly be a supported model of it, the notion of supportedness is generally not sufficient to exclude all superfluous transitions from the transition relation. This is shown by the following example.

Example 5.3.6. Suppose we have a TSS P with one constant f, one label a and the following rule:

$$\frac{f \xrightarrow{a} f}{f \xrightarrow{a} f}.$$

We would like P to define the transition relation $\longrightarrow_P = \emptyset$. We feel that there is not enough reason to add $f \xrightarrow{a} f$ to $\longrightarrow_P$ as it can only be 'derived' by assuming that it is already in $\longrightarrow_P$.

However, both $\emptyset$ and $\{f \xrightarrow{a} f\}$ are supported models of P.

For positive TSS's this shortcoming is easily remedied by associating with a TSS P the *least* transition relation (w.r.t. set inclusion) that is a model of P. The existence of this least model follows from the model intersection property stated below.

Lemma 5.3.7 *(Model intersection property).* *Let P be a TSS and let $\mathcal{C}$ be a collection of models of P. Then $\bigcap \mathcal{C}$ is a model of P.*

Proof. Let r be a ground instance of a rule of P. If $\bigcap \mathcal{C} \models prem(r)$, then for every $\longrightarrow \in \mathcal{C}$: $\longrightarrow \models prem(r)$, thus for every $\longrightarrow \in \mathcal{C}$: $\longrightarrow \models conc(r)$, as $\mathcal{C}$ is a collection of models. Thus $\bigcap \mathcal{C} \models conc(r)$. □

Thus we have the following definition.

Definition 5.3.8. The transition relation $\longrightarrow_P$ *associated with* a positive TSS P is the least model of P w.r.t. set inclusion.

Traditionally ([14, 15, 19]) a different definition of the transition relation associated with a positive TSS was given, based on the *provability* of transitions. We show that these two characterisations are equivalent.

Definition 5.3.9. Let $P = (\Sigma, A, R)$ be a positive TSS. A *proof* of a positive literal ψ from P is a well-founded, upwardly branching tree of which the nodes are labeled by literals $t \xrightarrow{a} t'$ with $t, t' \in \mathbb{T}(\Sigma)$ and $a \in A$, such that:

- the root is labeled with ψ,
- if χ is the label of a node q and $\{\chi_i \mid i \in I\}$ is the set of labels of the nodes directly above q, then there is a rule $\dfrac{\{\varphi_i \mid i \in I\}}{\varphi}$ in R and a substitution $\sigma{:}V \to \mathbb{T}(\Sigma)$ such that $\chi = \sigma(\varphi)$ and $\chi_i = \sigma(\varphi_i)$ for $i \in I$.

A proof is *closed* if it only contains closed literals. A positive literal ψ is *provable* in P, notation $P \vdash \psi$, if there exists a proof of ψ from P.

Theorem 5.3.10. *Let $P = (\Sigma, A, R)$ be a positive TSS, $\longrightarrow_P$ the transition relation associated with P and $\psi \in Tr(\Sigma, A)$. Then*

$$P \vdash \psi \quad \Leftrightarrow \quad \psi \in \longrightarrow_P .$$

Proof.

$\Rightarrow$ By straightforward induction on the proof of ψ from P.

$\Leftarrow$ It is straightforward to show that $\{\psi \mid P \vdash \psi\}$ is a model of P. As $\longrightarrow_P$ is the least model of P, it follows that $\longrightarrow_P \subseteq \{\psi \mid P \vdash \psi\}$.

□

From this theorem it also follows that the least model of a positive TSS is supported by it.

For TSS's with negative premises it is much more difficult to find an appropriate associated transition relation as is shown by the following example.

Example 5.3.11. Suppose we have a TSS P with one constant f, two labels a and b and the following rules:

$$\frac{f \xrightarrow{a} f}{f \xrightarrow{a} f} \qquad\qquad \frac{f \not\xrightarrow{a}}{f \xrightarrow{b} f}.$$

We would like P to define the transition relation $\longrightarrow_P = \{f \xrightarrow{b} f\}$. However, P has exactly two minimal models, $\{f \xrightarrow{a} f\}$ and $\{f \xrightarrow{b} f\}$, which are both supported.

Thus in the presence of negative premises there may be several minimal models, some of them may be supported. So other characterisations for associated transition relations must be sought. The notion of provability also needs a revision, as it is not a priori clear how the negative premises of a rule must be proved.

Similar problems concerning negative premises have been studied in the context of logic programming. The correspondence between TSS's and logic programs is treated in section 5.11. As first solution in logic programming the notion of (local) stratification was introduced. In the paper in chapter 4 this notion has been tailored for TSS's. We repeat the most important definitions and facts to make this paper self-contained.

A TSS P is stratified if there exists a *stratification* of the transitions with respect to the rules of P. The stratification guarantees that the validity of any literal does not depend on the validity of its negation.

Definition 5.3.12. Let $P = (\Sigma, A, R)$ be a TSS. A function $S : Tr(\Sigma, A) \to \alpha$, where α is an ordinal, is called a *stratification* of P if for every rule $r \in R$ and every substitution $\sigma : V \to T(\Sigma)$ it holds that:

$$\begin{array}{l}\text{for all } \psi \in pprem(\sigma(r)) : \ S(\psi) \leq S(conc(\sigma(r)) \text{ and} \\ \text{for all } t\stackrel{a}{\not\longrightarrow} \in nprem(\sigma(r)) \text{ and } t' \in T(\Sigma) : \ S(t \stackrel{a}{\longrightarrow} t') < S(conc(\sigma(r))).\end{array}$$

If P has a stratification, we say that P is *stratified*. For all ordinals $\beta < \alpha$, $S_\beta = \{\varphi \mid S(\varphi) = \beta\}$ is called a *stratum*.

Example 5.3.13. The TSS of example 5.3.11 can be stratified by a stratification S as follows:

$$S(f \stackrel{a}{\longrightarrow} f) = 0 \text{ and } S(f \stackrel{b}{\longrightarrow} f) = 1.$$

Each positive transition system specification is trivially stratified by putting all positive literals in stratum 0.

We now define how a transition relation $\longrightarrow_{P,S}$ is constructed from a TSS P with stratification S, (see also definition 4.2.14 in the paper in chapter 4). The idea of the construction is that one first considers the positive literals in stratum 0. As each literal in stratum 0 can only fit the conclusion of a rule without negative premises, one can determine which of these literals hold and which do not hold in $\longrightarrow_{P,S}$ in the same way as is done for positive transition system specifications. If a literal in stratum 1 fits the conclusion of a rule, then this instance of that rule can only have negative premises in stratum 0. If these negative premises hold (which has already been determined), they can be discarded. If they do not hold, the rule cannot be applied. Then we can prove the literals in stratum 1 in the ordinary way and we proceed with stratum 2 etc.

Definition 5.3.14. Let $P = (\Sigma, A, R)$ be a TSS with a stratification $S : Tr(\Sigma, A) \to \alpha$ for some ordinal α. The transition relation $\longrightarrow_{P,S}$ *associated with* P (and based on S) is defined as:

$$\longrightarrow_{P,S} = \bigcup_{0 \leq i < \alpha} \longrightarrow_{P_i} .$$

where $\longrightarrow_{P_i}$ is defined by the (positive) TSS $P_i = (\Sigma, A, R_i)$ with R_i given by:

$$R_i = \{r' \mid \exists r \in R \text{ and } \exists \sigma : V \to T(\Sigma) :$$

$$\bigcup_{0 \leq j < i} \longrightarrow_{P_j} \models nprem(\sigma(r)) \cup \{\varphi \in pprem(\sigma(r)) \mid S(\varphi) < i\},$$

$$S(conc(\sigma(r))) = i \text{ and}$$

$$r' = \frac{\{\varphi \in pprem(\sigma(r)) \mid S(\varphi) = i\}}{conc(\sigma(r))}\}.$$

Theorem 5.3.15 *(See also lemma 4.2.16). Let P be a TSS which is stratified by stratifications S and S'. Then $\longrightarrow_{P,S} = \longrightarrow_{P,S'}$.*

This theorem allows us to write $\longrightarrow_P$ for the transition relation associated with a stratified TSS P. Note that the definition of $\longrightarrow_P$ based on the notion of 'stratification' extends the definition of $\longrightarrow_P$ for positive TSS's.

Theorem 5.3.16 *(See also theorem 4.2.15 and theorem 4 in [21]). Let P be a stratified TSS. Then $\longrightarrow_P$ is a minimal and supported model of P.*

Thus we have the scheme of characterisations depicted in figure 5.1, where $A \to B$ means that characterisation A implies characterisation B. For positive TSS's, the characterisations marked by a $*$ coincide.

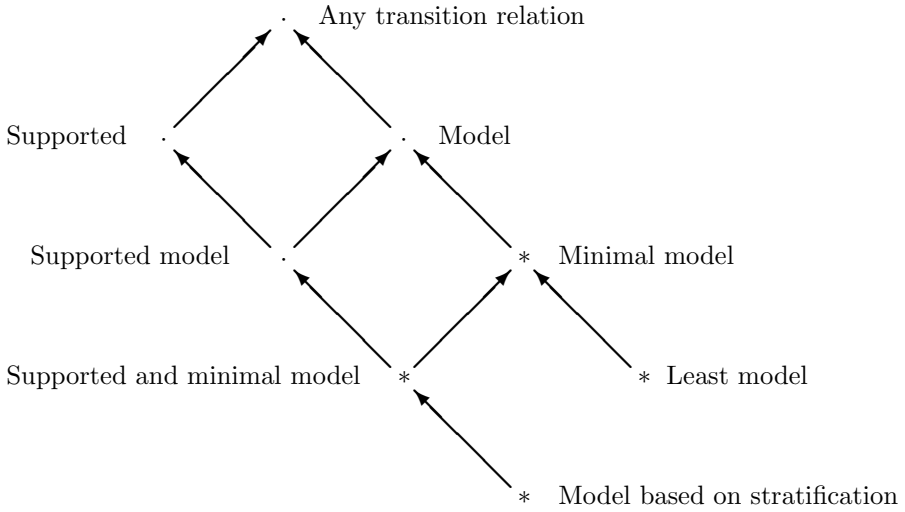


Figure 5.1: Relations among several models

Although the stratification technique is often applicable, there are examples of TSS's that have an intuitive meaning while not being stratified. One such example is $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities.

Example 5.3.17. Suppose we have an instance P_θ of $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities based on a set of actions Act containing at least two elements a and b such that $a < b$. Consider for arbitrary terms t and u the following instances of rules:

$$\text{R5.1:} \quad \frac{t \xrightarrow{a} u \quad \forall b > a \; t \not\xrightarrow{b}}{\theta(t) \xrightarrow{a} \theta(u)},$$

$$\text{R7.2:} \quad \frac{\theta(t) \xrightarrow{a} \theta(u)}{\tau_{\{a\}}(\theta(t)) \xrightarrow{\tau} \tau_{\{a\}}(\theta(u))},$$

$$\text{R9.3:} \quad \frac{t \xrightarrow{b} \tau_{\{a\}}(\theta(t)) \quad \tau_{\{a\}}(\theta(t)) \xrightarrow{\tau} \tau_{\{a\}}(\theta(u))}{t \xrightarrow{b} \tau_{\{a\}}(\theta(u))}.$$

For any stratification S of P_θ it should thus hold that

$$S(t \xrightarrow{b} \tau_{\{a\}}(\theta(u))) < \qquad \text{(R5.1)}$$

$$S(\theta(t) \xrightarrow{a} \theta(u)) \leq \qquad \text{(R7.2)}$$

$$S(\tau_{\{a\}}(\theta(t)) \xrightarrow{\tau} \tau_{\{a\}}(\theta(u))) \leq \qquad \text{(R9.3)}$$

$$S(t \xrightarrow{b} \tau_{\{a\}}(\theta(u))).$$

Of course, such a stratification cannot exist.

Again, this problem has been recognised earlier in logic programming, and several more powerful techniques were introduced there [20, 4, 11, 10]. In the following two sections we adapt [11] and [10] for TSS's.

5.4 TSS's and their associated transition relations

So far no meaning has been given to TSS's that are not stratified. There are however TSS's, like $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities, that seem to be perfectly meaningful while not being stratified. This brings us back to the fundamental question what transition relation should be associated with a TSS. Our answer is essentially that the transition relation must be the unique *stable model* in the sense of logic programming [11]. We have not found any example of a TSS that has no unique stable transition relation and yet has a plausible meaning.

The definition of a stable transition relation is motivated as follows. Our first observation is that positive and negative premises in a rule of a TSS P have a

different status. In order to prove the conclusion of a rule, the positive premises of the rule must be proved from P. However, as P contains only rules defining which literals hold, but not which literals do not hold, negative premises must be treated differently.

Conceptually, $t \stackrel{a}{\not\longrightarrow}$ holds by default, i.e. if for no t': $t \stackrel{a}{\longrightarrow} t'$ can be proved. But we are still trying to determine which literals can be proved. So instead of an immediate characterisation of the set of provable literals $\longrightarrow$, we have an equation with this set both on the left and on the right side, namely:

> $\longrightarrow$ equals the set of literals that are provable by those rules of the TSS of which the negative premises hold in $\longrightarrow$.

This equation does not give us a means to compute the transition relation $\longrightarrow$, but we can easily check whether a given transition relation satisfies our criterion.

We now formalise these ideas. In section 5.4, 5.5 and 5.6 we use only ground TSS's, i.e. we identify a set of rules R with the set of ground instances of R.

Definition 5.4.1. Let $P = (\Sigma, A, R)$ be a TSS. Let $\longrightarrow \subseteq Tr(\Sigma, A)$.

$$\mathit{Strip}(P, \longrightarrow) = (\Sigma, A, \mathit{Strip}(R, \longrightarrow))$$

where

$$\mathit{Strip}(R, \longrightarrow) = \{r' \mid \exists r \in R : \ \longrightarrow \models \mathit{nprem}(r) \text{ and } r' = \frac{\mathit{pprem}(r)}{\mathit{conc}(r)}\}.$$

Given a transition relation $\longrightarrow$, the function *Strip* removes all rules in R that have negative premises that do not hold in $\longrightarrow$. Furthermore, it drops the negative premises from the remaining rules. The following lemma is therefore obvious.

Lemma 5.4.2. *Let $P = (\Sigma, A, R)$ be a TSS and let $\longrightarrow \subseteq Tr(\Sigma, A)$ be a transition relation. Then $\mathit{Strip}(P, \longrightarrow)$ is a positive TSS.*

Using the fact that the notion of provability is already captured in the definition of the transition relation associated with a positive TSS, we can now easily formalise the previously stated equation.

Definition 5.4.3 *(Stable transition relation).* Let $P = (\Sigma, A, R)$ be a TSS. A transition relation $\longrightarrow \subseteq Tr(\Sigma, A)$ is *stable* for P if $\longrightarrow = \longrightarrow_{\mathit{Strip}(P, \longrightarrow)}$.

Remark 5.4.4. In general, for a TSS P there may be 0, 1 or more transition

relations that are stable for P, e.g.

$$0: \quad \frac{f \overset{a}{\not\longrightarrow}}{f \overset{a}{\longrightarrow} f}$$

$$1: \quad \frac{f \overset{a}{\not\longrightarrow} \quad f \overset{b}{\longrightarrow} f}{f \overset{a}{\longrightarrow} f} \qquad [\longrightarrow = \emptyset]$$

$$2: \quad \frac{f \overset{a}{\not\longrightarrow}}{f \overset{b}{\longrightarrow} f} \qquad \frac{f \overset{b}{\not\longrightarrow}}{f \overset{a}{\longrightarrow} f} \qquad [\longrightarrow = \{f \overset{a}{\longrightarrow} f\} \text{ or } \longrightarrow = \{f \overset{b}{\longrightarrow} f\}]$$

We do not have any idea as to which transition relations should be associated with the first TSS, nor do we know which one of the two transition relations of the third TSS should be preferred. In fact we think that there are no satisfying answers to those questions. Thus we propound the following definition.

Definition 5.4.5. Let P be a TSS. If there is a unique transition relation $\longrightarrow$ stable for P, then $\longrightarrow$ is the transition relation *associated with* P.

In order to avoid confusion, we do not again introduce the notation $\longrightarrow_P$: until section 5.7 this notation remains reserved for stratified TSS's.

Remark 5.4.6. If P is positive, then for every transition relation $\longrightarrow$

$$\mathit{Strip}(P, \longrightarrow) = P$$

and thus $\longrightarrow_P$ is the unique transition relation that is stable for P. Hence, this definition of 'associated with' coincides with the previously given definition for positive TSS's. In section 5.6 we show that our choice also extends the definition of 'associated with' for stratified TSS's.

The following lemma will be used implicitly in almost every proof to follow. Moreover, it shows that our choice that a transition relation must be stable for a TSS is also a refinement of the requirement that a transition relation must be a supported and minimal model of it.

Lemma 5.4.7. *Let $P = (\Sigma, A, R)$ be a TSS and let $\longrightarrow \subseteq Tr(\Sigma, A)$ be a transition relation. If $\longrightarrow$ is stable for P, then*

1. *$\longrightarrow$ is a model of P,*
2. *$\longrightarrow$ is supported by P,*
3. *$\longrightarrow$ is a minimal model of P (Cf. [11], theorem 1).*

Proof. Let $\longrightarrow$ be a transition relation that is stable for P.

1. Suppose $r \in R$ and $\longrightarrow \models prem(r)$. Hence

$$\frac{pprem(r)}{conc(r)} \in Strip(R, \longrightarrow).$$

   As $\longrightarrow = \longrightarrow_{Strip(P,\longrightarrow)}$ is a model of $Strip(P, \longrightarrow)$ and $\longrightarrow \models pprem(r)$, $\longrightarrow \models conc(r)$.

2. Suppose $\varphi \in \longrightarrow$. Hence, $\longrightarrow \models \varphi$ and thus $\longrightarrow_{Strip(P,\longrightarrow)} \models \varphi$. This means that there is a proof for φ using the rules in $Strip(R, \longrightarrow)$. Assume rule r is the last rule used. So $conc(r) = \varphi$. Hence $Strip(P, \longrightarrow) \vdash prem(r)$ and thus $\longrightarrow \models prem(r)$. As $r \in Strip(R, \longrightarrow)$ there is a rule

$$r' = \frac{prem(r) \cup nprem}{conc(r)} \in R$$

   where $nprem$ is a set of negative premises such that $\longrightarrow \models nprem$. Hence, for this rule $r' \in R$ it holds that $\varphi = conc(r')$ and $\longrightarrow \models prem(r')$. Hence $\longrightarrow$ is supported by P.

3. We must show that $\longrightarrow$ is minimal among the models of P. Suppose $\longrightarrow^* \subseteq \longrightarrow$ is a model of P. We show that $\longrightarrow^*$ is a model of $Strip(P, \longrightarrow)$. Let r be a rule of $Strip(P, \longrightarrow)$. This means that there is some

$$r' = \frac{prem(r) \cup nprem}{conc(r)} \in R$$

   for some set $nprem$ of negative premises. As $\longrightarrow^* \subseteq \longrightarrow$ and $\longrightarrow \models nprem$, $\longrightarrow^* \models nprem$. As $\longrightarrow^*$ is a model of P, we have

   if $\longrightarrow^* \models prem(r) \cup nprem$, then $\longrightarrow^* \models conc(r)$.

   Knowing that $\longrightarrow^* \models nprem$, this reduces to

   if $\longrightarrow^* \models prem(r)$, then $\longrightarrow^* \models conc(r)$.

   Thus $\longrightarrow^*$ is a model for every rule r in $Strip(P, \longrightarrow)$. As $\longrightarrow$ is the least model of $Strip(P, \longrightarrow)$, it follows that $\longrightarrow^* = \longrightarrow$.

□

We show how the notion *is stable for* can be applied to our running example. What we in fact show is that in general there is no stable transition relation for $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities instantiated with a set of process declarations where the abstraction operator τ_I is allowed in process terms.

Example 5.4.8. Consider P_θ with at least two actions a and b such that $a > b$ and a process name X with the recursive definition

$$X \Leftarrow \theta(\tau_{\{b\}}(X) \cdot a + b) \in E.$$

Now assume that there is a relation $\longrightarrow$ that is stable for P_θ. We show that this assumption leads to a contradiction. For a more convenient notation, we use $t \stackrel{a}{\longrightarrow}$ as an abbreviation of $\exists u \in T(\Sigma_\theta) : \ \longrightarrow\models t \stackrel{a}{\longrightarrow} u$ $(t \in T(\Sigma_\theta),\ a \in A_\theta)$. We distinguish three cases but we do not present them in full detail. In particular not all possible applications of R9.2 and R9.3 are considered explicitly.

- $\tau_{\{b\}}(X) \stackrel{\surd}{\longrightarrow}$. As $\longrightarrow$ is a model of P_θ, we have that $\tau_{\{b\}}(X) \cdot a \stackrel{a}{\longrightarrow}$ (by rule R4.2) and hence $\tau_{\{b\}}(X) \cdot a + b \stackrel{a}{\longrightarrow}$ (by rule R3.1). Thus $\theta(\tau_{\{b\}}(X) \cdot a + b) \stackrel{a}{\longrightarrow}$ and $\longrightarrow\models \theta(\tau_{\{b\}}(X) \cdot a + b) \stackrel{b}{\not\longrightarrow}$ (by rule R5.1). So $\longrightarrow\models X \stackrel{b}{\not\longrightarrow}$. As obviously $\longrightarrow\models X \stackrel{\surd}{\not\longrightarrow}$ (X must perform at least an a or b action), it follows that $\longrightarrow\models \tau_{\{b\}}(X) \stackrel{\surd}{\not\longrightarrow}$. Contradiction.
- $\longrightarrow\models \tau_{\{b\}}(X) \stackrel{\surd}{\not\longrightarrow}$ and $\longrightarrow\models \tau_{\{b\}}(X) \stackrel{a}{\not\longrightarrow}$. Then obviously $\longrightarrow\models \tau_{\{b\}}(X) \cdot a + b \stackrel{a}{\not\longrightarrow}$, so $\longrightarrow\models \theta(\tau_{\{b\}}(X) \cdot a + b) \stackrel{b}{\longrightarrow} \theta(\epsilon)$ (using R2, R3.2 and R5.1). Hence, $\longrightarrow\models X \stackrel{b}{\longrightarrow} \theta(\epsilon)$, so $\longrightarrow\models \tau_{\{b\}}(X) \stackrel{\surd}{\longrightarrow} \tau_{\{b\}}(\theta(\delta))$ (using R8, R7.2, R1, R5.2, R7.2 and R9.2). Contradiction.
- $\longrightarrow\models \tau_{\{b\}}(X) \stackrel{\surd}{\not\longrightarrow}$ and $\tau_{\{b\}}(X) \stackrel{a}{\longrightarrow} t$ for some $t \in T(\Sigma_\theta)$. This can also not be the case as there is no proof for $\mathit{Strip}(P_\theta, \longrightarrow) \vdash \tau_{\{b\}}(X) \stackrel{a}{\longrightarrow} t$. In order to prove $\tau_{\{b\}}(X) \stackrel{a}{\longrightarrow} t$, we must show that $X \stackrel{a}{\longrightarrow}$, in order to show this we need $\tau_{\{b\}}(X) \cdot a \stackrel{a}{\longrightarrow}$. In combination with the assumption that $\tau_{\{b\}}(X) \stackrel{\surd}{\not\longrightarrow}$, this requires $\tau_{\{b\}}(X) \stackrel{a}{\longrightarrow}$ again. Thus the most 'reasonable' attempt to construct the required proof loops. All other attempts to prove $\tau_{\{b\}}(X) \stackrel{a}{\longrightarrow} t$, e.g. via R9.3:

$$\frac{\tau_{\{b\}}(X) \stackrel{a}{\longrightarrow} u \quad u \stackrel{\tau}{\longrightarrow} t}{\tau_{\{b\}}(X) \stackrel{a}{\longrightarrow} t},$$

  also loop.

5.5 Reducing TSS's

We now present a technique that can be useful for proving that a certain TSS has a unique stable transition relation. This technique is inspired by the *well-founded models* that are introduced in [10]. First we construct a 3-valued 'interpretation' for a TSS P, partitioning the set of transitions in three groups: those that are

certainly true, those of which the truth is unknown and those that are certainly not true. We apply this information to *reduce* P to another TSS with exactly the same stable transition relations as P. In this new TSS, the truth or falsity of more literals may become certain. Repeated reduction may lead to complete information: the unique stable transition relation.

If in the next definition $\longrightarrow_{true}$ contains transitions that certainly hold and $\longrightarrow_{pos}$ contains all transitions that possibly hold, then rules with certainly wrong premises are removed and in the remaining rules all premises that certainly hold are dropped.

Definition 5.5.1. Let $P = (\Sigma, A, R)$ be a TSS. Let $\longrightarrow_{true}, \longrightarrow_{pos} \subseteq Tr(\Sigma, A)$ be transition relations.

$$Reduce(P, \longrightarrow_{true}, \longrightarrow_{pos}) = (\Sigma, A, Reduce(R, \longrightarrow_{true}, \longrightarrow_{pos})),$$

where

$$Reduce(R, \longrightarrow_{true}, \longrightarrow_{pos}) =$$

$$\{r' \mid \exists r \in R : \longrightarrow_{true} \models nprem(r), \quad \longrightarrow_{pos} \models pprem(r) \text{ and}$$
$$r' = \frac{\{\psi \in pprem(r) \mid \longrightarrow_{true} \not\models \psi\} \ \cup \ \{\psi \in nprem(r) \mid \longrightarrow_{pos} \not\models \psi\}}{conc(r)}\}.$$

Thus the reduction of a rule consists of two phases. First it is checked that the premises are *possibly* true. For positive premises this is straightforward: $t \stackrel{a}{\longrightarrow} t'$ is possibly true if $t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_{pos}$. Hence the condition $\longrightarrow_{pos} \models pprem(r)$. A negative premise $t \stackrel{a}{\not\longrightarrow}$ is possibly true if it is not certain that t can perform an a-step, i.e. for no t' it is certain that $t \stackrel{a}{\longrightarrow} t'$ holds. Thus $t \stackrel{a}{\not\longrightarrow}$ is possibly true if for no t', $t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_{true}$. Hence the condition $\longrightarrow_{true} \models nprem(r)$.

If indeed the premises of the rule are possibly true, then the premises that are *certainly* true are removed. A positive premise $t \stackrel{a}{\longrightarrow} t'$ is certainly true if $t \stackrel{a}{\longrightarrow} t' \in \longrightarrow_{true}$. A negative premise $t \stackrel{a}{\not\longrightarrow}$ is certainly true if t cannot possibly perform an a-step, i.e. for no t': $t \stackrel{a}{\longrightarrow} t'$ is possible. Thus $t \stackrel{a}{\not\longrightarrow}$ is certainly true if $\longrightarrow_{pos} \models t \stackrel{a}{\not\longrightarrow}$.

Remark 5.5.2. Note that $Reduce(R, \longrightarrow, \longrightarrow)$ differs from $Strip(R, \longrightarrow)$. In $Strip(R, \longrightarrow)$ only negative premises are checked, yielding a positive TSS; in $Reduce(R, \longrightarrow, \longrightarrow)$ all premises are checked, resulting in a TSS consisting solely of rules without premises.

The 3-valued interpretation required is obtained by means of two positive TSS's: $True(P)$ and $Pos(P)$. $True(P)$ determines the transitions that are certainly true: the transitions that can be proved with positive rules only. $Pos(P)$ determines the transitions that are possibly true, i.e. true or unknown. These are the transitions that can be proved ignoring negative premises. Thus $Pos(P)$ is obtained from P by removing all negative premises of the rules.

Definition 5.5.3. Let $P = (\Sigma, A, R)$ be a TSS.

- $True(P) = (\Sigma, A, True(R))$ where $True(R) = \{r \in R \mid nprem(r) = \emptyset\}$.
- $Pos(P) = (\Sigma, A, Pos(R))$ where $Pos(R) = \{r' \mid \exists r \in R : \ r' = \frac{pprem(r)}{conc(r)}\}$.

Because after the reduction of P the truth or falsity of more literals may become certain, it is worthwhile to iterate the reduction process; if necessary even transfinitely many reduction steps may be considered.

Definition 5.5.4. Let $P = (\Sigma, A, R)$ be a TSS. For every ordinal α, the α-reduction of P, notation $Red^\alpha(P)$, is recursively defined as follows:

- $Red^0(P) = (\Sigma, A, R_{ground})$ where R_{ground} is the set of all ground instances of rules in R,
- $Red^\alpha(P) = Reduce(P, \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))}, \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))})$.

Thus in contrast with [10] and general practice in logic programming, our operator maps TSS's to TSS's rather than interpretations to interpretations; for details see the last section. This allows us in section 5.6 to combine reduction with stratification: as soon as the reduced TSS is stratified, no further reduction is needed.

The following lemma plays an important role in a number of proofs to follow. It shows that the reduction process can never make a certainly true (or false) literal become unknown. Thus reduction is monotonic in this sense.

Lemma 5.5.5 *(Monotonicity of reduction).* *Let $P = (\Sigma, A, R)$ be a TSS. For all ordinals β and α such that $\beta < \alpha$ and for every $\longrightarrow \subseteq Tr(\Sigma, A)$:*

$$\begin{aligned} \longrightarrow_{True(Red^\beta(P))} \subseteq \longrightarrow_{True(Red^\alpha(P))} \quad &\subseteq \\ \longrightarrow_{Strip(Red^\alpha(P), \longrightarrow)} &\subseteq \\ \longrightarrow_{Pos(Red^\alpha(P))} \quad &\subseteq \longrightarrow_{Pos(Red^\beta(P))} . \end{aligned}$$

Proof.

(1) First we show that

$$\longrightarrow_{True(Red^\alpha(P))} \subseteq \longrightarrow_{Strip(Red^\alpha(P), \longrightarrow)} \subseteq \longrightarrow_{Pos(Red^\alpha(P))} .$$

For every TSS $P' = (\Sigma, A, R')$: $True(R') \subseteq Strip(R', \longrightarrow) \subseteq Pos(R')$. As these TSS's are all positive, $\longrightarrow_{True(P')} \subseteq \longrightarrow_{Strip(P', \longrightarrow)} \subseteq \longrightarrow_{Pos(P')}$. Now taking $P' = Red^\alpha(P)$ proves this case.

(2) Here it is shown that

$$\longrightarrow_{Pos(Red^{\alpha}(P))} \subseteq \longrightarrow_{Pos(Red^{\beta}(P))} .$$

Suppose

$$Pos(Red^{\alpha}(P)) \vdash \varphi.$$

Then there is a rule $r' \in Pos(Red^{\alpha}(P))$ such that $conc(r') = \varphi$ and $Pos(Red^{\alpha}(P)) \vdash prem(r')$. Hence, there is a rule $r \in R$ such that $conc(r) = \varphi$ and

$$\bigcup_{\gamma<\alpha} \longrightarrow_{True(Red^{\gamma}(P))} \models nprem(r) \text{ and}$$
$$\bigcap_{\gamma<\alpha} \longrightarrow_{Pos(Red^{\gamma}(P))} \models pprem(r).$$

Now

$$\bigcup_{\gamma<\beta} \longrightarrow_{True(Red^{\gamma}(P))} \subseteq \bigcup_{\gamma<\alpha} \longrightarrow_{True(Red^{\gamma}(P))} \text{ and}$$

$$\bigcap_{\gamma<\beta} \longrightarrow_{Pos(Red^{\gamma}(P))} \supseteq \bigcap_{\gamma<\alpha} \longrightarrow_{Pos(Red^{\gamma}(P))} .$$

Hence,

$$\bigcup_{\gamma<\alpha} \longrightarrow_{True(Red^{\gamma}(P))} \models nprem(r)$$

implies that

$$\bigcup_{\gamma<\beta} \longrightarrow_{True(Red^{\gamma}(P))} \models nprem(r).$$

Conversely,

$$\bigcap_{\gamma<\alpha} \longrightarrow_{Pos(Red^{\gamma}(P))} \models pprem(r)$$

implies

$$\bigcap_{\gamma<\beta} \longrightarrow_{Pos(Red^{\gamma}(P))} \models pprem(r).$$

Hence $Pos(Red^{\beta}(R))$ contains a rule r'' such that $prem(r'') \subseteq pprem(r)$ and $conc(r'') = \varphi$. As also $\bigcap_{\gamma<\alpha} \longrightarrow_{Pos(Red^{\gamma}(P))} \models pprem(r)$ implies $\longrightarrow_{Pos(Red^{\beta}(P))} \models pprem(r)$, it follows that $\longrightarrow_{Pos(Red^{\beta}(P))} \models prem(r'')$. Thus $Pos(Red^{\beta}(P)) \vdash \varphi$.

(3) We are left to show $\longrightarrow_{True(Red^{\beta}(P))} \subseteq \longrightarrow_{True(Red^{\alpha}(P))}$. We show this by induction on α. By induction we may assume that:

$$\text{for all } \zeta \leq \gamma < \alpha : \qquad \longrightarrow_{True(Red^{\zeta}(P))} \subseteq \longrightarrow_{True(Red^{\gamma}(P))} .$$

Suppose $True(Red^\beta(P)) \vdash \varphi$. Then there is an instance r of a rule in R such that $conc(r) = \varphi$,

$$\bigcap_{\gamma<\beta} \longrightarrow_{Pos(Red^\gamma(P))} \models nprem(r)$$

(all negative premises of r must be removed by reduction) and

$$\bigcup_{\gamma\leq\beta} \longrightarrow_{True(Red^\gamma(P))} \models pprem(r)$$

(all positive premises of r must be removed by reduction or proved in $True(Red^\beta(P))$). From this, using the induction hypothesis (i.h.) and (1) and (2) above, we can infer the following two facts:

- $\bigcap_{\gamma<\beta} \longrightarrow_{Pos(Red^\gamma(P))} \models nprem(r) \Rightarrow$
  $\bigcap_{\gamma\leq\beta} \longrightarrow_{Pos(Red^\gamma(P))} \models nprem(r) \overset{(2)}{\Rightarrow}$
  $\longrightarrow_{Pos(Red^\beta(P))} \models nprem(r) \overset{(2)}{\Rightarrow}$
  $\forall\gamma\ (\beta \leq \gamma < \alpha)$: $\longrightarrow_{Pos(Red^\gamma(P))} \models nprem(r) \overset{(1)}{\Rightarrow}$
  $\forall\gamma\ (\beta \leq \gamma < \alpha)$: $\longrightarrow_{True(Red^\gamma(P))} \models nprem(r) \overset{(i.h.)}{\Rightarrow}$
  $\forall\gamma < \alpha$:$\longrightarrow_{True(Red^\gamma(P))} \models nprem(r) \Rightarrow$
  $\bigcup_{\gamma<\alpha} \longrightarrow_{True(Red^\gamma(P))} \models nprem(r).$ $<5.1>$
- $\bigcup_{\gamma\leq\beta} \longrightarrow_{True(Red^\gamma(P))} \models pprem(r) \overset{(i.h.)}{\Rightarrow}$
  $\longrightarrow_{True(Red^\beta(P))} \models pprem(r) \overset{(i.h.)}{\Rightarrow}$
  $\forall\gamma\ (\beta \leq \gamma < \alpha)$: $\longrightarrow_{True(Red^\gamma(P))} \models pprem(r) \overset{(1)}{\Rightarrow}$
  $\forall\gamma\ (\beta \leq \gamma < \alpha)$: $\longrightarrow_{Pos(Red^\gamma(P))} \models pprem(r) \overset{(2)}{\Rightarrow}$
  $\forall\gamma < \alpha$: $\longrightarrow_{Pos(Red^\gamma(P))} \models pprem(r) \Rightarrow$
  $\bigcap_{\gamma<\alpha} \longrightarrow_{Pos(Red^\gamma(P))} \models pprem(r).$ $<5.2>$

$<5.1>$ and $<5.2>$ imply $\exists r' = \dfrac{prem(r) - ...}{\varphi} \in Red^\alpha(R)$.
Furthermore,

$$\bigcap_{\gamma<\beta} \longrightarrow_{Pos(Red^\gamma(P))} \models nprem(r) \quad \Rightarrow$$

$$\bigcap_{\gamma<\alpha} \longrightarrow_{Pos(Red^\gamma(P))} \models nprem(r) \quad \Rightarrow$$

$$nprem(r') = \emptyset \quad \Rightarrow$$

$$r' \in True(Red^\alpha(R)).$$

As for every $\psi \in prem(r)$, the proof of ψ in $True(Red^\beta(P))$ is less deep than the proof of φ in $True(Red^\beta(P))$, we may conclude by induction

that $prem(r) \subseteq \longrightarrow_{True(Red^\alpha(P))}$. As $prem(r') \subseteq prem(r)$, $conc(r') = \varphi \in \longrightarrow_{True(Red^\alpha(P))}$.

□

In order to apply this reduction process, we also need the following lemma.

Lemma 5.5.6. *Let $P = (\Sigma, A, R)$ be a TSS and let $\longrightarrow \subseteq Tr(\Sigma, A)$. For all ordinals α*

$$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \subseteq \longrightarrow \subseteq \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))}$$

implies

$$\longrightarrow_{Strip(P,\longrightarrow)} = \longrightarrow_{Strip(Red^\alpha(P),\longrightarrow)}.$$

Proof.

- We show that $\longrightarrow_{Strip(P,\longrightarrow)} \subseteq \longrightarrow_{Strip(Red^\alpha(P),\longrightarrow)}$. Let $\psi \in \longrightarrow_{Strip(P,\longrightarrow)}$. Hence, $Strip(P, \longrightarrow) \vdash \psi$. We use induction on this proof. There is a rule r' in $Strip(P, \longrightarrow)$ such that $conc(r') = \psi$ and $\longrightarrow_{Strip(P,\longrightarrow)} \models prem(r')$. Hence, there is a rule $r \in R$ such that

$$r' = \frac{pprem(r)}{conc(r)}$$

and $\longrightarrow \models nprem(r)$. By induction on α and lemma 5.5.5 it follows that

$$\forall \beta < \alpha : \quad \longrightarrow_{Strip(P,\longrightarrow)} = \longrightarrow_{Strip(Red^\beta(P),\longrightarrow)} \subseteq \longrightarrow_{Pos(Red^\beta(P))}$$

and hence:

$$\left.\begin{array}{l} \longrightarrow \models nprem(r) \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \subseteq \longrightarrow \end{array}\right\} \Rightarrow \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \models nprem(r),$$

$$\left.\begin{array}{l} \longrightarrow_{Strip(P,\longrightarrow)} \models pprem(r) \\ \longrightarrow_{Strip(P,\longrightarrow)} \subseteq \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \end{array}\right\} \Rightarrow \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \models pprem(r).$$

So there is a rule

$$r'' = \frac{\begin{array}{c}\{\psi' \in pprem(r) \mid \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \not\models \psi'\} \cup \\ \{\psi' \in nprem(r) \mid \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \not\models \psi'\}\end{array}}{conc(r)} \in Red^\alpha(P).$$

Furthermore, $\longrightarrow \models nprem(r'') \subseteq nprem(r)$. So

$$r''' = \frac{\{\psi' \in pprem(r) \mid \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \not\models \psi'\}}{conc(r)} \in Strip(Red^\alpha(P), \longrightarrow).$$

By induction on the depth of the proof of ψ from $Strip(P, \longrightarrow)$ we may assume that $prem(r') \subseteq \longrightarrow_{Strip(Red^\alpha(P),\longrightarrow)}$, so

$$\longrightarrow_{Strip(Red^\alpha(P),\longrightarrow)} \models prem(r') = pprem(r) \supseteq prem(r''').$$

Hence, it follows that

$$Strip(Red^\alpha(P), \longrightarrow) \models conc(r''') = \psi.$$

- Here we show that $\longrightarrow_{Strip(P,\longrightarrow)} \supseteq \longrightarrow_{Strip(Red^\alpha(P,\longrightarrow))}$. So assume that $\psi \in \longrightarrow_{Strip(Red^\alpha(P),\longrightarrow)}$. Hence, it must be that $Strip(Red^\alpha(P), \longrightarrow) \vdash \psi$. So there is a rule r' in $Strip(Red^\alpha(P), \longrightarrow)$ such that $conc(r') = \psi$ and $\longrightarrow_{Strip(Red^\alpha(P),\longrightarrow)} \models prem(r')$. Then there is a rule $r \in R$ such that

$$r' = \frac{\{\psi' \in pprem(r) \mid \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \not\models \psi'\}}{conc(r)}$$

and $\longrightarrow \models \{\psi \in nprem(r) \mid \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \not\models \psi'\}$.

$\longrightarrow \models nprem(r)$: Let $\psi' \in nprem(r)$. If $\bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \not\models \psi'$ then $\longrightarrow \models \psi'$. If $\bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \models \psi'$ then also $\longrightarrow \models \psi'$, as $\longrightarrow \subseteq \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))}$.

So in $Strip(P, \longrightarrow)$ we have the rule

$$r'' = \frac{pprem(r)}{conc(r)}.$$

By induction on the depth of the proof of ψ from $Strip(Red^\alpha(P), \longrightarrow)$ we may conclude that

$$\longrightarrow_{Strip(P,\longrightarrow)} \models prem(r') = \{\psi' \in pprem(r) \mid \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \not\models \psi'\}.$$

By the induction on α and lemma 5.5.5,

$$\forall \beta < \alpha : \quad \longrightarrow_{True(Red^\beta(P))} \subseteq \longrightarrow_{Strip(Red^\beta(P),\longrightarrow)} \subseteq \longrightarrow_{Strip(P,\longrightarrow)}.$$

Hence, if $\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \models \psi'$, then $\longrightarrow_{Strip(P,\longrightarrow)} \models \psi'$. Thus

$$\longrightarrow_{Strip(P,\longrightarrow)} \models prem(r'') = prem(r') \cup \{\psi' \in pprem(r) \mid \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \models \psi'\}.$$

So $\longrightarrow_{Strip(P,\longrightarrow)} \models conc(r'') = \psi$.

□

Our hope is that after sufficiently many reductions we obtain a positive TSS. If this is the case, then our method has succeeded: the transition relation of this positive TSS is the unique transition relation that is stable for the original one. (Example 5.8.12 shows that the converse is not true: a TSS having a unique stable transition relation need not reduce to a positive TSS.)

Theorem 5.5.7 *(Soundness of reduction).* *Let $P = (\Sigma, A, R)$ be a TSS and let $\longrightarrow \subseteq Tr(\Sigma, A)$. For all ordinals α we have:*

$$\longrightarrow \text{ is stable for } P \Leftrightarrow \longrightarrow \text{ is stable for } Red^{\alpha}(P).$$

Proof.

$\Rightarrow$) Let $\longrightarrow = \longrightarrow_{Strip(P,\longrightarrow)}$.
We prove by induction that for all ordinals α:

$$\longrightarrow = \longrightarrow_{Strip(Red^{\alpha}(P),\longrightarrow)}, \tag{5.1}$$

$$\longrightarrow_{True(Red^{\alpha}(P))} \subseteq \longrightarrow \subseteq \longrightarrow_{Pos(Red^{\alpha}(P))}. \tag{5.2}$$

By lemma 5.5.5, always (5.1) $\Rightarrow$ (5.2), so we must prove (5.1).
Basis. $\longrightarrow = \longrightarrow_{Strip(P,\longrightarrow)} = \longrightarrow_{Strip(Red^{0}(P),\longrightarrow)}$ is given.
Induction. By induction it follows from (5.2) that for all $\beta < \alpha$:

$$\longrightarrow_{True(Red^{\beta}(P))} \subseteq \longrightarrow \subseteq \longrightarrow_{Pos(Red^{\beta}(P))}.$$

So

$$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P))} \subseteq \longrightarrow \subseteq \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P))}.$$

So by lemma 5.5.6

$$\longrightarrow = \longrightarrow_{Strip(P,\longrightarrow)} = \longrightarrow_{Strip(Red^{\alpha}(P),\longrightarrow)}.$$

$\Leftarrow$) Let $\longrightarrow = \longrightarrow_{Strip(Red^{\alpha}(P),\longrightarrow)}$.
Then by lemma 5.5.5 for all $\beta < \alpha$:

$$\longrightarrow_{True(Red^{\beta}(P))} \subseteq \longrightarrow \subseteq \longrightarrow_{Pos(Red^{\beta}(P))}.$$

So again

$$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P))} \subseteq \longrightarrow \subseteq \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P))}$$

and by lemma 5.5.6 $\longrightarrow = \longrightarrow_{Strip(P,\longrightarrow)} = \longrightarrow_{Strip(Red^{\alpha}(P),\longrightarrow)}$.

□

Corollary 5.5.8 *(Cf. [10], corollary 6.2). If P reduces to a positive TSS, i.e. $Red^{\alpha}(P)$ is positive for some α, then $\longrightarrow_{Red^{\alpha}(P)}$ is associated with P.*

5.6 Reduction and stratification

We now have two independent methods for associating a transition relation with a TSS with negative premises: reduction and stratification. Three questions arise:

- if both methods are applicable, is their result the same?
- is one method (strictly) stronger than the other?
- is it useful to combine the two methods?

In this section we shall answer these questions affirmatively. We show that for a stratified TSS P, the relation $\longrightarrow_P$ as defined in section 5.3 is stable for P. Furthermore, we show that repeatedly reducing a stratified TSS yields a positive TSS. Thus $\longrightarrow_P$ is the unique transition relation that is stable for P. This is also the answer to our second question: reduction is indeed stronger than stratification (that it is strictly stronger is easily seen by the second TSS in remark 5.4.4).

So it seems that there is no point in combining the two methods: the result could not be stronger than reduction alone. However, for practical purposes the combination appears to be valuable, due to the fact that the existence of a stratification is generally easier to demonstrate. Therefore, we show in this section that the methods can be used cooperatively, rather than being alternatives for each other.

Finally, we use this amalgamation to demonstrate that the TSS $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities has an associated transition relation under some conditions.

Theorem 5.6.1. *If P is stratified, then $\longrightarrow_P$ is stable for P.*

Proof. Let $P = (\Sigma, A, R)$ and let $S : Tr(\Sigma, A) \to \alpha$ be a stratification of P.

1. We show that $\longrightarrow_{Strip(P,\longrightarrow_P)} \subseteq \longrightarrow_P$. Suppose $Strip(P, \longrightarrow_P) \vdash \psi$. We use induction on the structure of the proof of ψ. As $Strip(P, \longrightarrow_P) \vdash \psi$, there exists a rule $r' \in Strip(R, \longrightarrow_P)$ such that $prem(r') \subseteq \longrightarrow_{Strip(P,\longrightarrow_P)}$ and $\psi = conc(r')$. So $\exists r \in R$: $pprem(r) = prem(r')$, $conc(r) = conc(r')$ and $\longrightarrow_P \models nprem(r)$. By induction $pprem(r) \subseteq \longrightarrow_P$. Hence, $\longrightarrow_P \models prem(r)$. As by theorem 5.3.16 $\longrightarrow_P$ is a model of P, $\psi = conc(r) \in \longrightarrow_P$.

2. Here we show that $\longrightarrow_P \subseteq \longrightarrow_{Strip(P,\longrightarrow_P)}$. Recall that

$$\longrightarrow_P = \bigcup_{0 \le i < \alpha} \longrightarrow_{P_i} .$$

   By induction it is shown that for every i, $0 \le i < \alpha$: $\longrightarrow_{P_i} \subseteq \longrightarrow_{Strip(P,\longrightarrow_P)}$. Let $\psi \in \longrightarrow_{P_i}$, hence $P_i \vdash \psi$. With induction on the proof of ψ from P_i we show that $Strip(P, \longrightarrow_P) \vdash \psi$.

Suppose the last rule used to prove ψ from P_i is r'. This means according definition 5.3.14 that there is a rule $r \in R$ and a substitution $\sigma : V \to T(\Sigma)$ such that

$$\bigcup_{0 \leq j < i} \longrightarrow_{P_j} \models nprem(\sigma(r)) \cup \{\varphi \in pprem(\sigma(r)) \mid S(\varphi) < i\},$$

$$r' = \frac{\{\varphi \in pprem(\sigma(r)) \mid S(\varphi) = i\}}{conc(\sigma(r))}$$

and $conc(r) = \psi$. As P is stratified, for all $t \stackrel{a}{\not\longrightarrow} \in nprem(\sigma(r))$ and $t' \in T(\Sigma)$: $S(t \stackrel{a}{\longrightarrow} t') < S(\psi) = i$. Thus $\bigcup_{0 \leq j < i} \longrightarrow_{P_j} \models nprem(\sigma(r))$ implies $\longrightarrow_P \models nprem(\sigma(r))$ and therefore there is a rule

$$r'' = \frac{pprem(\sigma(r))}{conc(\sigma(r))} \in Strip(P, \longrightarrow_P).$$

For all $\chi \in prem(r'')$ with $S(\chi) < i$: $\longrightarrow_{P_{S(\chi)}} \models \chi$, so by induction $\longrightarrow_{Strip(P, \longrightarrow_P)} \models \chi$. For all $\chi \in prem(r'')$ with $S(\chi) = i$, it follows with induction on the proof tree that $\longrightarrow_{Strip(P, \longrightarrow_P)} \models \chi$. So, $Strip(P, \longrightarrow_P) \vdash prem(r'')$ and hence, $Strip(P, \longrightarrow_P) \vdash conc(r'') = \psi$.

□

Theorem 5.6.2. *Let $P = (\Sigma, A, R)$ be a TSS with stratification $S : Tr(\Sigma, A) \to \alpha$. Then $Red^{\alpha}(P)$ is a positive TSS.*

Proof. We show that $\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P))} = \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P))}$. According to remark 5.5.2 this is sufficient.

$\subseteq$. This implication follows immediately from lemma 5.5.5.

$\supseteq$. We claim that for any $\psi \in Tr(\Sigma, A)$:

$$\psi \in \longrightarrow_{Pos(Red^{S(\psi)}(P))} \quad \Rightarrow \quad \psi \in \longrightarrow_{True(Red^{S(\psi)}(P))} .$$

Using the claim, we can easily finish the proof: as $S(\psi) < \alpha$, we have

$$\begin{array}{l} \psi \in \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P))} \quad \Rightarrow \\ \psi \in \longrightarrow_{Pos(Red^{S(\psi)}(P))} \quad \Rightarrow \\ \psi \in \longrightarrow_{True(Red^{S(\psi)}(P))} \quad \Rightarrow \\ \psi \in \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{S(\psi)}(P))} . \end{array}$$

We prove our claim by transfinite induction on $S(\psi)$. Assume the induction hypothesis holds for all $\gamma < \beta$. Take some $\psi \in Tr(\Sigma, A)$ with $S(\psi) = \beta$. Furthermore, assume $\psi \in \longrightarrow_{Pos(Red^{\beta}(P))}$. Hence, there is a

proof of ψ from $Pos(Red^\beta(P))$. With induction on this proof, we show that $True(Red^\beta(P)) \vdash \psi$. As $Pos(Red^\beta(P)) \vdash \psi$, there is a rule $r \in Pos(Red^\beta(R))$ such that $conc(r) = \psi$ and $Pos(Red^\beta(P)) \vdash prem(r)$. Hence, there is some rule $r' \in Red^\beta(R)$ such that $conc(r') = conc(r) = \psi$ and $pprem(r') = prem(r)$. We show that $nprem(r') = \emptyset$. In order to obtain a contradiction, assume $t \overset{a}{\not\longrightarrow} \in nprem(r')$. As $r' \in Red^\beta(P)$, we know:

$$\bigcup_{\zeta<\beta} \longrightarrow_{True(Red^\zeta(P))} \models t \overset{a}{\not\longrightarrow}.$$

So for every $t' \in T(\Sigma)$: $t \overset{a}{\longrightarrow} t' \notin \bigcup_{\zeta<\beta} \longrightarrow_{True(Red^\zeta(P))}$. In particular, as $S(t \overset{a}{\longrightarrow} t') = \gamma' < S(\psi) = \beta$, $t \overset{a}{\longrightarrow} t' \notin \longrightarrow_{True(Red^{\gamma'}(P))}$. By induction, $t \overset{a}{\longrightarrow} t' \notin \longrightarrow_{Pos(Red^{\gamma'}(P))}$ and so $t \overset{a}{\longrightarrow} t' \notin \bigcap_{\zeta<\beta} \longrightarrow_{Pos(Red^\zeta(P))}$. Therefore,

$$\bigcap_{\zeta<\beta} \longrightarrow_{Pos(Red^\zeta(P))} \models t \overset{a}{\not\longrightarrow}.$$

Hence, $t \overset{a}{\not\longrightarrow} \notin nprem(r')$. As $nprem(r') = \emptyset$, $r = r' \in True(Red^\beta(R))$. By induction (on the depth of the proof tree of $Pos(Red^\beta(P)) \vdash \psi$) we know that $True(Red^\beta(P)) \vdash prem(r)$ and thus $True(Red^\beta(P)) \vdash \psi$. So we can conclude $\psi \in \longrightarrow_{True(Red^\beta(P))}$.

□

Corollary 5.6.3 *(Cf. [11], corollary 1 and [10], theorem 6.3). Let $P = (\Sigma, A, R)$ be a TSS with stratification $S : Tr(\Sigma, A) \to \alpha$. Then $\longrightarrow_P = \longrightarrow_{Red^\alpha(P)}$ is associated with P.*

Proof. Directly using theorem 5.6.1, theorem 5.6.2 and corollary 5.5.8. □

Lemma 5.6.4. *Let P be a TSS.*

$$Red^\alpha(Red^\beta(P)) = Red^{\alpha+\beta}(P).$$

Proof. Straightforward with induction on α, using lemma 5.5.5. □

Corollary 5.6.5 *(Combining reduction and stratification). Let $P = (\Sigma, A, R)$ be a TSS and suppose that for ordinals α and β, $S : Tr(\Sigma, A) \to \alpha$ is a stratification of $Red^\beta(P)$. Then $Red^{\alpha+\beta}(P)$ is a positive TSS and $\longrightarrow_{Red^{\alpha+\beta}(P)} = \longrightarrow_{Red^\beta(P)}$ is associated with P.*

Proof. By theorem 5.6.2 and lemma 5.6.4 it follows that $Red^\alpha(Red^\beta(P)) = Red^{\alpha+\beta}(P)$ is a positive TSS. Using corollary 5.6.3 and lemma 5.6.4 we have that

$$\longrightarrow_{Red^\beta(P)} = \longrightarrow_{Red^\alpha(Red^\beta(P))} = \longrightarrow_{Red^{\alpha+\beta}(P)}$$

is the transition relation associated with $Red^\beta(P)$. By theorem 5.5.7 $\longrightarrow_{Red^\beta(P)}$ is associated with P. □

In the remainder of this section we apply this corollary to show that a transition relation is associated with an instance P_θ of $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities, provided that two conditions hold:

1. The abstraction operator τ_I does not occur in the process terms in the right hand side of a recursive equation. The reason for this condition was already shown in example 5.4.8. This conforms to the standard practise in process algebra.

2. There is no $a \in Act$ such that $\tau < a$. The motivation for this second condition is threefold (cf. [26] where it is argued that $\tau > a$ for all actions a seems the most 'intuitive' choice).

   – It is essential that τ-actions are not observable. Thus between two observable actions, any number of τ-actions can take place, and must be possible in any process specification. Indeed, the τ-rules R9.1-R9.3 ensure that (in $\mathrm{BPA}_{\delta\epsilon\tau}$) every specification satisfies this property. However, allowing $\tau < a$ would destroy this property, as in this case e.g. $\theta(a \cdot a)$ specifies a process performing two a-actions, with no τ-actions in between (assuming there is no $b > a$):

   $$True(P_\theta) \vdash \theta(a \cdot a) \xrightarrow{a} \theta(\epsilon \cdot a) \xrightarrow{a} \theta(\epsilon),$$

   but for no $t, t' \in T(\Sigma_\theta)$:

   $$Pos(Red^1(P_\theta)) \vdash \theta(a \cdot a) \xrightarrow{a} t \xrightarrow{\tau} t'.$$

   (If $\theta(a \cdot a) \xrightarrow{a} t \in \longrightarrow_{Pos(Red^1(P_\theta))}$, then $t \equiv \theta(u)$ for some u such that $True(P_\theta) \vdash u \xrightarrow{a} \epsilon$; every rule in P_θ with a conclusion of the form $\theta(u) \xrightarrow{\tau} t'$ has a premise $\theta(u) \xrightarrow{\tau} u'$ (R9.2 and R9.3) or $u \overset{a}{\not\longrightarrow}$ (R5.1)).

   – As a consequence, the axiom $\theta(a \cdot x) = a \cdot \theta(x)$, which is part of the complete axiomatisation of $\mathrm{BPA}_{\epsilon\delta}$ with priorities (without τ, Cf. [1] axiom TH1 and TH2), is no longer valid: when $\tau < a$, $\theta(a \cdot a)$ cannot perform τ after a although $a \cdot \theta(a)$ can.

   – We conjecture that there is only one transition relation stable for P_θ, even for instances with $\tau < a$. However, we have no proof for this. In particular, we do not know whether such an instance of $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities always reduces to a stratified TSS. The problem is caused by the fact that we do not reduce one TSS (with $(Act, <)$ and (Ξ, E) fixed), but try to reduce the whole class of instances of $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities (satisfying condition 1) at once.

Theorem 5.6.6. *If for all $(X \Leftarrow t_X) \in E$: $\tau_I(\cdot)$ does not occur in t_X and for all $a \in Act$ it does not hold that $\tau < a$, then there is a transition relation associated with P_θ.*

Proof. We show that P_θ is stratified after one reduction step. To this end we formulate a useful property of $Red^1(P_\theta)$. Define $N : T(\Sigma_\theta) \to \mathbb{N}$ by:

$$\begin{array}{l} N(a) = N(\epsilon) = N(\delta) = N(\tau) = N(X) = 0 \qquad (X \in \Xi \text{ and } a \in Act), \\ N(x + y) = N(x \cdot y) = N(x \triangleleft y) = max(N(x), N(y)), \\ N(\theta(x)) = N(x), \\ N(\tau_I(x)) = N(x) + 1. \end{array}$$

We show that it is not possible to prove in P_θ a literal $t \xrightarrow{a} u$ when $N(t) < N(u)$ (i.e. the 'N-complexity' of a process, the depth of nestings of $\tau_I(\cdot)$'s in it, cannot increase by performing an action).

Fact 1. For all $a \in A_\theta$ we have:

$$t \xrightarrow{a} u \in \longrightarrow_{Pos(P_\theta)} \quad \Rightarrow \quad N(t) \geq N(u)$$

Proof of fact 1. It can be shown for every ground instance r of a rule in P_θ that if for every literal $t \xrightarrow{a} u \in pprem(r)$ $N(t) \geq N(u)$ holds, then $N(t') \geq N(u')$ holds, where $conc(r) = t' \xrightarrow{b} u'$. Instead of giving a detailed treatment of each rule, we only prove the most important ones here:

R4.1 $N(x \cdot y) = \max(N(x), N(y)) \geq N(x) \geq N(x')$ and $\max(N(x), N(y)) \geq N(y)$. This implies that $\max(N(x), N(y)) \geq \max(N(x'), N(y)) = N(x' \cdot y)$.

R8 $N(X) = 0$. So we must prove $N(y) = 0$. Indeed $N(y) \leq N(t_X) = 0$ as by assumption t_X does not contain τ_I-operators.

□

For example, the literal $t \xrightarrow{b} \tau_{\{a\}}(\theta(t))$ that is used in example 5.3.17 to make $t \xrightarrow{b} \tau_{\{a\}}(\theta(u))$ depend negatively on itself, is not possible. Based on this definition of N we define the preorder $\leq$ on pairs of literals by:

$$(t \xrightarrow{a} u) \leq (t' \xrightarrow{b} u') \text{ iff } \begin{cases} N(t) < N(t') \text{ or} \\ N(t) = N(t') \text{ and } (a = \tau, \surd),\ a > b, \text{ or } a = b. \end{cases}$$

For some ordinal α we can now define a function $S : Tr(\Sigma_\theta, A_\theta) \to \alpha$ obtained by transforming the preorder $\leq$ into a complete well-founded ordering:

$$\begin{array}{l} \varphi \approx \psi \text{ iff } \varphi \leq \psi \text{ and } \psi \leq \varphi, \\ \varphi \approx \psi \ \Rightarrow\ S(\varphi) = S(\psi), \\ \varphi \leq \psi \text{ and not } \varphi \approx \psi \ \Rightarrow\ S(\varphi) < S(\psi). \end{array}$$

(We do not need a more precise definition of S; since such a definition necessarily depends on the size of the set Act, we omit it).

Fact 2. S is a stratification of $Reduce(P_\theta, \longrightarrow_{True(P_\theta)}, \longrightarrow_{Pos(P_\theta)})$.

Proof of fact 2. Let r be a ground instance of a rule in

$$Reduce(P_\theta, \longrightarrow_{True(P_\theta)}, \longrightarrow_{Pos(P_\theta)}).$$

We must show that for every $\psi \in pprem(r)$: $S(\psi) \leq S(conc(r))$. Furthermore, it must hold that for every $\psi = t \stackrel{a}{\not\longrightarrow} \in nprem(r)$ and for every $t' \in T(\Sigma)$: $S(t \stackrel{a}{\longrightarrow} t') < S(conc(r))$. For most rules this is trivial, as the unreduced instances of the rule already satisfy the requirement. We only consider the most interesting cases:

R5.1 $N(\theta(x)) = N(x)$ implies that $S(\theta(x) \stackrel{a}{\longrightarrow} \theta(x')) = S(x \stackrel{a}{\longrightarrow} x')$. For each $b > a$ it holds that $S(x \stackrel{b}{\longrightarrow} t') < S(x \stackrel{a}{\longrightarrow} x')$ for any $t' \in T(\Sigma_\theta)$.

R6.1 $N(x) \leq N(x \triangleleft y)$, so $S(x \stackrel{a}{\longrightarrow} x') \leq S(x \triangleleft y \stackrel{a}{\longrightarrow} x')$. Also $N(y) \leq N(x \triangleleft y)$, so for each $b > a$ and $t' \in T(\Sigma_\theta)$: $S(y \stackrel{b}{\longrightarrow} t') < S(x \triangleleft y \stackrel{a}{\longrightarrow} x')$.

R7.2 $N(\tau_I(x)) = N(x) + 1 > N(x)$. So $S(x \stackrel{a}{\longrightarrow} x') < S(\tau_I(x) \stackrel{\tau}{\longrightarrow} \tau_I(x'))$.

R9.2 and R9.3 By the first fact $Reduce(P_\theta, \longrightarrow_{True(P_\theta)}, \longrightarrow_{Pos(P_\theta)})$ contains only those instances of these rules for which $N(x) \geq N(y) \geq N(z)$. We need $N(x) \geq N(y)$ to prove e.g. $(y \stackrel{\tau}{\longrightarrow} z) \leq (x \stackrel{a}{\longrightarrow} z)$, hence $S(y \stackrel{\tau}{\longrightarrow} z) \leq S(x \stackrel{a}{\longrightarrow} z)$.

□

Using corollary 5.6.5 $\longrightarrow_{Reduce(P_\theta, \longrightarrow_{True(P_\theta)}, \longrightarrow_{Pos(P_\theta)})} = \longrightarrow_{Red^1(P_\theta)}$ is associated with P_θ. □

5.7 Bisimulation relations

We have defined the meaning of a TSS as its associated transition relation and shown how to arrive at this transition relation. Now we switch to the study of properties of transition relations as consequences of properties of their defining TSS's.

An important question (e.g. in process verification) is whether two terms denote the 'same' process. Many process equivalences based on transition relations have been proposed ([13]), of which strong bisimulation equivalence is most often used [17, 18]. In this and the subsequent sections some relations between TSS's and strong bisimulation equivalence are studied.

Definition 5.7.1. Let P be a TSS with associated transition relation $\longrightarrow_P$. A relation R is a *strong bisimulation relation* based on P if it satisfies:

- whenever tRu and $t \xrightarrow{a}_P t'$ then, for some $u' \in T(\Sigma)$, we have $u \xrightarrow{a}_P u'$ and $t'Ru'$,
- whenever tRu and $u \xrightarrow{a}_P u'$ then, for some $t' \in T(\Sigma)$, we have $t \xrightarrow{a}_P t'$ and $t'Ru'$.

Two terms $t, u \in T(\Sigma)$ are (P-)*bisimilar*, notation $t \leftrightarrows_P u$, if there is a strong bisimulation relation R based on P such that tRu. Note that $\leftrightarrows_P$, the strong bisimulation equivalence *induced* by P, is an equivalence relation.

Thus $t \leftrightarrows_P u$ means that if t can do some step, u can do a 'similar' step (and vice versa, hence the name *bi*simulation). In the next section we prove that under specific conditions on P, $\leftrightarrows_P$ is a congruence relation. To this end we shall approximate $\longrightarrow_P$ by other transition relations $\longrightarrow_Q$, and use the notion of $P \Rightarrow Q$-bisimulation, meaning that if t can do some step in $\longrightarrow_P$, u can do a 'similar' step in $\longrightarrow_Q$ (and vice versa, i.e. if u can do a step in $\longrightarrow_P$, t can do a 'similar' step in $\longrightarrow_Q$). In the end, the approximation $\longrightarrow_Q$ will be equal to $\longrightarrow_P$. It may be readily checked that in this case, $P \Rightarrow Q$-bisimulation is exactly P-bisimulation. Thus showing that for every approximation $\longrightarrow_Q$ $P \Rightarrow Q$-bisimulation is a congruence is sufficient to show that P-bisimulation is a congruence.

Formally, we have the following definition.

Definition 5.7.2. Let $P = (\Sigma, A, R_P)$ and $Q = (\Sigma, A, R_Q)$ be TSS's with associated transition relations $\longrightarrow_P$ and $\longrightarrow_Q$. A relation R is a *strong $P \Rightarrow Q$-bisimulation relation* if it satisfies:

- whenever tRu and $t \xrightarrow{a}_P t'$ then, for some $u' \in T(\Sigma)$, we have $u \xrightarrow{a}_Q u'$ and $t'Ru'$,
- whenever tRu and $u \xrightarrow{a}_P u'$ then, for some $t' \in T(\Sigma)$, we have $t \xrightarrow{a}_Q t'$ and $t'Ru'$.

We say that two terms $t, u \in T(\Sigma)$ are *$P \Rightarrow Q$-bisimilar*, notation $t \leftrightarrows_{P \Rightarrow Q} u$, if there is a strong $P \Rightarrow Q$-bisimulation relation R such that tRu. Note that like $\leftrightarrows_P$, $\leftrightarrows_{P \Rightarrow Q}$ is symmetric. In contrast with $\leftrightarrows_P$, $\leftrightarrows_{P \Rightarrow Q}$ need not be reflexive or transitive.

5.8 The *ntyft/ntyxt*-format and the congruence theorem

A desirable property for TSS's is that the induced strong bisimulation equivalence is a congruence. In [14] this led to the observation that if a (positive) TSS is in

the so-called *tyft/tyxt*-format then this is the case. In the paper in chapter 4 this result was extended to stratified TSS's. In order to express the fact that negative premises are allowed, *n*'s were added to the name of the format, obtaining the *ntyft/ntyxt*-format. In this section we show that even for TSS's that are positive after reduction, bisimulation is a congruence if the TSS is in *ntyft/ntyxt*-format. In the end of this section we show that 'positive after reduction' is a necessary requirement for the congruence theorem: we give a TSS in *ntyft/ntyxt*-format with a unique stable transition relation for which strong bisimulation is not a congruence.

Definition 5.8.1. Let $\Sigma = (F, rank)$ be a signature. Let $P = (\Sigma, A, R)$ be a TSS. A rule $r \in R$ is in *ntyft-format* if it has the form:

$$\frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \ \cup \ \{t_l \xrightarrow{b_l} \not\rightarrow \mid l \in L\}}{f(x_1, ..., x_{rank(f)}) \xrightarrow{a} t}$$

with K and L (possibly infinite) index sets, y_k, x_i $(1 \leq i \leq rank(f))$ all different variables, $a_k, b_l, a \in A$, $f \in F$ and $t_k, t_l, t \in \mathbb{T}(\Sigma)$. A rule $r \in R$ is in *ntyxt-format* if it fits:

$$\frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \ \cup \ \{t_l \xrightarrow{b_l} \not\rightarrow \mid l \in L\}}{x \xrightarrow{a} t}$$

with K, L (possibly infinite) index sets, y_k, x all different variables, $a_k, b_l, a \in A$, t_k, t_l and $t \in \mathbb{T}(\Sigma)$. P is in *ntyft-format* if all its rules are in *ntyft*-format and P is in *ntyft/ntyxt*-format if all its rules are either in *ntyft*- or in *ntyxt*-format.

It may be useful to point out why this format is called the *ntyft/ntyxt*-format. As stated above, the '*n*' was added to indicate the possibility of negative premises. The letters *tyft* can be found if one reads first the (positive) premises and then the conclusion from left to right: t represents a term in the left hand side of a premise, y the variable in the right hand side; f is the function name in the left hand side of the conclusion and t the term in the right hand side. Similarly, the other format is called *ntyxt*.

As in [14] and theorem 4.4.12 in the paper in chapter 4, we need the following well-foundedness condition in order to prove the congruence theorem.

Definition 5.8.2 *(Well-foundedness).* Let $P = (\Sigma, A, R)$ be a TSS. Let $W = \{t_k \xrightarrow{a_k} t'_k \mid k \in K\} \subseteq \mathbb{T}(\Sigma) \times A \times \mathbb{T}(\Sigma)$ be a set of positive literals over Σ and A. The *variable dependency graph* of W is a directed (unlabeled) graph VDG with:

- Nodes: $\bigcup_{k \in K} Var(t_k \xrightarrow{a_k} t'_k)$,
- Edges: $\{< x, y > \mid x \in Var(t_k),\ y \in Var(t'_k)$ for some $k \in K\}$.

W is called *well-founded* if any backward chain of edges in the variable dependency graph is finite. A rule is called *well-founded* if its set of positive premises is well-founded. A TSS is called *well-founded* if all its rules are well-founded.

Definition 5.8.3. Let $P = (\Sigma, A, R)$ be a TSS. Let $r \in R$ be a rule. A variable x is called *free* in r if it occurs in r but not in the source of the conclusion or in the target of a positive premise. The rule r is called *pure* if it is well-founded and does not contain free variables. P is called *pure* if all rules in R are pure.

In what follows we state a number of technicalities needed for the proof of theorem 5.8.11. At first reading it is advised to skip the remainder of this section except for this theorem.

Definition 5.8.4. Let W be a set of positive literals which is well-founded and let VDG be the variable dependency graph of W. Let $Var(W)$ be the set of variables occurring in literals in W. Define for each $x \in Var(W)$: $n_{VDG}(x) = sup(\{n_{VDG}(y) + 1 \mid < y, x > \text{ is an edge of } VDG\})$ $(sup(\emptyset) = 0)$.

Remark 5.8.5. If W is a set of positive premises of a rule in *ntyft/ntyxt*-format then $n_{VDG}(x) \in \mathbb{N}$ for each $x \in Var(W)$: Every variable y_k occurs only once in the right hand side of a positive literal in the premises. As the term t_k is finite, it contains only a finite number of variables x. Therefore the set $U = \{n_{VDG}(x) + 1 \mid < x, y_k > \text{ is an edge of } VDG\}$ is finite. Hence, $n_{VDG}(y_k) = sup(U)$ is a natural number.

The following lemma states that any TSS in *ntyft/ntyxt*-format is 'equivalent' to a pure TSS in *ntyft*-format. This allows us to only study *ntyft*-rules.

Lemma 5.8.6. *Let P be a well-founded TSS in ntyft/ntyxt-format and let $\longrightarrow$ be the transition relation associated with P. Then there is a pure TSS P' in ntyft-format such that $\longrightarrow$ is also associated with P'. Moreover, P' is positive after reduction iff P is positive after reduction.*

Proof. Assume $P = (\Sigma, A, R)$ and $\Sigma = (F, rank)$. First we construct a TSS $P'' = (\Sigma, A, R'')$ which is pure and in *ntyft/ntyxt*-format. R'' contains a rule $\sigma(r)$ iff r is a rule in R and $\sigma : V \to \mathbb{T}(\Sigma)$ is a substitution such that for each variable that is free in r: $\sigma(x) \in T(\Sigma)$ and for each variable x that is not free in r $\sigma(x) = x$. From P'' we construct P' as follows: $P' = (\Sigma, A, R')$ where for each $f \in F$, a rule $\sigma_f(r) \in R'$ iff r is a rule in R'' and $\sigma_f : V \to \mathbb{T}(\Sigma)$ is a substitution satisfying:

if r is in *ntyft*-format, then $\sigma_f(z) = z$ for all $z \in V$,
if r is in *ntyxt*-format, then $\sigma_f(z) = z$ for all $z \in V - \{x\}$
and $\sigma_f(x) = f(z_1, ..., z_{rank(f)})$.

Here z_i $(1 \leq i \leq rank(f))$ are variables that do not occur in r. It is easy to see that P' is a pure TSS in *ntyft*-format. Observe that the ground instances of the rules in R, R' and R'' are the same. Also note that 'stable for' and 'positive after reduction' are defined w.r.t. these ground instances. Therefore, $\longrightarrow$ is also the unique transition relation stable for P' and P''. Furthermore P' and P'' are positive after reduction iff P is positive after reduction. □

The relation R_P that is defined now forms the backbone of all remaining proofs in this section.

Definition 5.8.7. Let $\Sigma = (F, rank)$ be a signature and let $P = (\Sigma, A, R)$ be a TSS with an associated transition relation. The relation $R_P \subseteq T(\Sigma) \times T(\Sigma)$ is the minimal relation satisfying:

- $\leftrightarrows_P \subseteq R_P$,
- for all function names $f \in F$:

$$\forall 1 \leq k \leq rank(f) : \; u_k R_P v_k \quad \Rightarrow \quad f(u_1, ..., u_{rank(f)}) R_P f(v_1, ..., v_{rank(f)}).$$

Note that this definition is in fact saying that R_P is the minimal congruence relation that includes $\leftrightarrows_P$. This explains the following lemma, which is a standard fact about congruence relations.

Lemma 5.8.8. *Let $P = (\Sigma, A, R)$ be a TSS with an associated transition relation. Let $t \in \mathbb{T}(\Sigma)$ and let $\sigma, \sigma' : V \to T(\Sigma)$ be substitutions such that for all x in $Var(t)$ $\sigma(x) R_P \sigma'(x)$. Then $\sigma(t) R_P \sigma'(t)$.*

Proof. Straightforward with induction on the structure of t. □

Lemma 5.8.9. *Let P be a pure TSS in ntyft-format. Suppose that $\longrightarrow_P$ is the transition relation that is associated with P. Then for all ordinals $\alpha \geq 0$: R_P is a*

1. *$P \Rightarrow Pos(Red^\alpha(P))$-bisimulation relation.*
2. *$True(Red^\alpha(P)) \Rightarrow P$-bisimulation relation.*

Proof. Assume $P = (\Sigma, A, R)$ and $\Sigma = (F, rank)$. We show the two statements in the lemma by mutual transfinite induction on α.

1. For reasons of symmetry it is enough to show that:

   if $u R_P v$ and $\longrightarrow_P \models u \xrightarrow{a} u'$,
   then $\exists v' \in T(\Sigma)$ such that $\longrightarrow_{Pos(Red^\alpha(P))} \models v \xrightarrow{a} v'$ and $u' R_P v'$.

   We prove this by induction on the proof of $u \xrightarrow{a} u'$ from $Strip(P, \longrightarrow_P)$. As $u R_P v$, two cases arise:

   - $u \leftrightarrows_P v$. Then $\longrightarrow_P \models u \xrightarrow{a} u'$ implies $\exists v' \in T(\Sigma)$: $\longrightarrow_P \models v \xrightarrow{a} v'$ and $u' \leftrightarrows_P v'$. By lemma 5.5.5 and theorem 5.5.7 $\longrightarrow_P \subseteq \longrightarrow_{Pos(Red^\alpha(P))}$. So $\longrightarrow_{Pos(Red^\alpha(P))} \models v \xrightarrow{a} v'$. Furthermore, $u' \leftrightarrows_P v'$ implies $u' R_P v'$.

– For some $f \in F$, $u = f(u_1, ..., u_{rank(f)})$, $v = f(v_1, ..., v_{rank(f)})$ and $u_i R_P v_i$ for $1 \leq i \leq rank(f)$. Then there is a rule:

$$r = \frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \cup \{t_l \xrightarrow{a_l}\!\!\!\!\!/ \mid l \in L\}}{f(x_1, ..., x_{rank(f)}) \xrightarrow{a} t} \in R$$

and a substitution σ such that $\sigma(x_i) = u_i$ $(1 \leq i \leq rank(f))$, $\sigma(t) = u'$, $\longrightarrow_P \models prem(\sigma(r))$ and

$$\frac{pprem(\sigma(r))}{conc(\sigma(r))}$$

is the last rule of the proof of $u \xrightarrow{a} u'$ from $Strip(P, \longrightarrow_P)$. Thus the proof of $\sigma(t_k \xrightarrow{a_k} y_k)$ $(k \in K)$ from $Strip(P, \longrightarrow_P)$ is less deep. As P is pure, $\{x_1, .., x_{rank(f)}\} \cup \{y_k \mid k \in K\} = Var(r)$.

Claim 1. There is a closed substitution σ' such that for all $x \in Var(r)$:

(a) $\sigma(x) R_P \sigma'(x)$,

(b) if $x = x_i$ then $\sigma'(x) = v_i$,

(c) if $x = y_k$ $(k \in K)$ then $\sigma'(t_k \xrightarrow{a_k} y_k) \in \longrightarrow_{Pos(Red^\alpha(P))}$,

(d) for all $l \in L$ and for all $\beta < \alpha$: $\longrightarrow_{True(Red^\beta(P))} \models \sigma'(t_l) \xrightarrow{a_l}\!\!\!\!\!/$.

Proof of claim 1. We prove the first three points of the claim by inductively constructing $\sigma'(x)$ for every $x \in Var(r)$, using induction on the degree of x in the VDG of $pprem(r)$.

For $x \in \{x_1, ..., x_{rank(f)}\}$, $\sigma'(x_i) = v_i$ is prescribed. Also $\sigma(x_i) = u_i R_P v_i = \sigma'(x_i)$ is satisfied.

For $x = y_k$ $(k \in K)$, we have $t_k \xrightarrow{a_k} y_k \in pprem(r)$. For all $y \in Var(t_k)$, $n_{VDG}(y) < n_{VDG}(x)$, so by induction $\sigma(y) R_P \sigma'(y)$. As R_P is a congruence, $\sigma(t_k) R_P \sigma'(t_k)$. Since the proof of $\sigma(t_k \xrightarrow{a_k} y_k)$ is less deep than the proof of $u \longrightarrow_a u'$ from $Strip(P, \longrightarrow_P)$, by induction $\exists w \in T(\Sigma)$:$\sigma'(t_k) \xrightarrow{a_k} w \in \longrightarrow_{Pos(Red^\alpha(P))}$ and $\sigma(y_k) R_P w$. Thus we take $\sigma'(y_k) = w$. Note that the first three points of claim 1 are satisfied which finishes the first part of the proof.

It remains to be shown that $\forall \beta < \alpha$: $\longrightarrow_{True(Red^\beta(P))} \models \sigma'(t_l) \xrightarrow{a_l}\!\!\!\!\!/$ $(l \in L)$. Again $\sigma(t_l) R_P \sigma'(t_l)$. Assume to generate a contradiction that $\exists \beta < \alpha$: $\longrightarrow_{True(Red^\beta(P))} \models \sigma'(t_l) \xrightarrow{a_l} s$ for some s. By simultaneous induction $\forall \beta < \alpha$: $\sigma(t_l) \leftrightarrows_{True(Red^\beta(P)) \Rightarrow P} \sigma'(t_l)$. So we have that $\longrightarrow_P \models \sigma(t_l) \xrightarrow{a_l} s'$ for some s'. This contradicts the fact that $\longrightarrow_P \models prem(\sigma(r))$. Hence, for all $l \in L$ and for all $\beta < \alpha$: $\longrightarrow_{True(Red^\beta(P))} \models \sigma'(t_l) \xrightarrow{a_l}\!\!\!\!\!/$. □

According to claim 1 there is a substitution σ' with the properties (a),(b),(c) and (d). Consider

$$\begin{aligned} V &= Pos(Red^{\alpha}(\{\sigma'(r)\}) \\ &= Pos(Reduce(\{\sigma'(r)\}, \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P))}, \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P))})). \end{aligned}$$

First we show that $\exists r' \in V$. It follows immediately from clause (d) in the claim that

$$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P))} \models nprem(\sigma'(r)).$$

Furthermore, by clause (c) and lemma 5.5.5:

$$\bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P))} \models pprem(\sigma'(r)).$$

Hence, there is some $r' \in V$. It follows from clause (c) in claim 1 that $\longrightarrow_{Pos(Red^{\alpha}(P))} \models pprem(\sigma'(r))$ and therefore, we have that $\longrightarrow_{Pos(Red^{\alpha}(P))} \models pprem(r') = prem(r')$. Thus $\longrightarrow_{Pos(Red^{\alpha}(P))} \models conc(r') = conc(\sigma'(r)) = v \xrightarrow{a} \sigma'(t)$ and $u' = \sigma(t) R_P \sigma'(t) = v'$.

2. For reasons of symmetry it is enough to show that:

   If uR_Pv and $\longrightarrow_{True(Red^{\alpha}(P))} \models u \xrightarrow{a} u'$,
   then $\exists v' \in T(\Sigma)$ such that $\longrightarrow_P \models v \xrightarrow{a} v'$ and $u' R_P v'$.

As $\longrightarrow_{True(Red^{\alpha}(P))} \models u \xrightarrow{a} u'$, $u \xrightarrow{a} u'$ can be proved from $True(Red^{\alpha}(P))$. We use induction on the depth of this proof. As uR_Pv we can distinguish two cases:

- $u \leftrightarrows_P v$. As by lemma 5.5.5 and theorem 5.5.7 $\longrightarrow_{True(Red^{\alpha}(P))} \subseteq \longrightarrow_P$, $P \models u \xrightarrow{a} u'$. So, $\exists v' \in T(\Sigma)$ such that $\longrightarrow_P \models v \xrightarrow{a} v'$ and $u' \leftrightarrows_P v'$. Hence, $u' R_P v'$.
- For some $f \in F$, $u = f(u_1, ..., u_{rank(f)})$, $v = f(v_1, ..., v_{rank(f)})$ and $u_i R_P v_i$ for $1 \leq i \leq rank(f)$. In this case the final (ground) rule $r \in True(Red^{\alpha}(R))$ of the proof of $u \xrightarrow{a} u'$ from $True(Red^{\alpha}(P))$ is also present in $Red^{\alpha}(R)$ and has no negative premises.

$$Red^{\alpha}(R) = \begin{cases} Reduce(R_{ground}, \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P))}, & \\ \qquad\qquad \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P))}) & \text{if } \alpha > 0, \\ R_{ground} & \text{if } \alpha = 0. \end{cases}$$

Thus there is a rule $r' \in R$ and a substitution $\sigma : V \to T(\Sigma)$ such that $\sigma(r')$ is reduced to r. This means that $conc(r) = conc(\sigma(r'))$ and

$prem(r) \subseteq pprem(\sigma(r'))$. Moreover, all negative premises of $\sigma(r')$ and all premises in $pprem(\sigma(r')) - pprem(r)$, which are removed, are redundant:

$$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^\beta(P))} \models pprem(\sigma(r')) - pprem(r),$$

$$\bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \models nprem(\sigma(r')).$$

As P is in *ntyft*-format, r' is of the form

$$\frac{\{t_k \xrightarrow{a_k} y_k \mid k \in K\} \cup \{t_l \xrightarrow{a_l} \!\!\!\!/ \mid l \in L\}}{f(x_1, ..., x_{rank(f)}) \xrightarrow{a} t}$$

and $\sigma(x_i) = u_i$ $(1 \leq i \leq rank(f))$, hence $\sigma(f(x_1, ..., x_{rank(f)})) = u$, and $\sigma(t) = u'$. As P is pure, $\{x_1, ..., x_{rank(f)}\} \cup \{y_k \mid k \in K\} = Var(r')$.

Claim 2. There is a closed substitution σ' such that for all $x \in Var(r')$:

(a) $\sigma(x) R_P \sigma'(x)$,
(b) if $x = x_i$ then $\sigma'(x) = v_i$,
(c) if $x = y_k$ $(k \in K)$ then $\sigma'(t_k) \xrightarrow{a_k} \sigma'(y_k) \in \longrightarrow_P$,
(d) for all $l \in L$: $\longrightarrow_P \models \sigma'(t_l) \xrightarrow{a_l} \!\!\!\!/$.

Proof of claim 2. We prove the first three points of the claim by giving a construction of $\sigma'(x)$ for every $x \in Var(r')$, using induction on the degree of x in the VDG of $pprem(r')$.

For $x \in \{x_1, ..., x_{rank(f)}\}$, $\sigma'(x_i) = v_i$ is prescribed. Also $\sigma(x_i) = u_i R_P v_i = \sigma'(x_i)$ is satisfied.

For $x = y_k$ $(k \in K)$, we have $t_k \xrightarrow{a_k} y_k \in pprem(r')$. For all $y \in Var(t_k)$, $n_{VDG}(y) < n_{VDG}(x)$, so by induction $\sigma(y) R_P \sigma'(y)$. As R_P is a congruence, $\sigma(t_k) R_P \sigma'(t_k)$. Two cases arise.

i. $\sigma(t_k \xrightarrow{a_k} y_k) \in pprem(r)$. Then there is a proof of $\sigma(t_k \xrightarrow{a_k} y_k)$ from $True(Red^\alpha(P))$ that is less deep than the proof of $u \xrightarrow{a} u'$. As $\sigma(t_k \xrightarrow{a_k} y_k) \in \longrightarrow_{True(Red^\alpha(P))}$ and $\sigma(t_k) R_P \sigma'(t_k)$, it follows by induction that $\exists w \in T(\Sigma)$: $\sigma'(t_k) \xrightarrow{a_k} w \in \longrightarrow_P$ and $\sigma(y_k) R_P w$.

ii. $\sigma(t_k \xrightarrow{a_k} y_k) \notin pprem(r)$. Hence $\exists \beta < \alpha$: $\longrightarrow_{True(Red^\beta(P))} \models \sigma(t_k \xrightarrow{a_k} y_k)$. As also $\sigma(t_k) R_P \sigma'(t_k)$, it follows by induction that $\exists w \in T(\Sigma) : \sigma'(t_k) \xrightarrow{a_k} w \in \longrightarrow_P$ and $\sigma(y_k) R_P w$.

In both cases, we take $\sigma'(y_k) = w$. Note that the first three points of claim 2 are satisfied, which finishes the first part of this proof.

We are left to show that $\longrightarrow_P \models \sigma'(t_l) \stackrel{a_l}{\not\longrightarrow}$ $(l \in L)$. As $\sigma(t_l) R_P \sigma'(t_l)$, it follows from point 1 of this lemma that $\sigma(t_l) \leftrightarrows_{P \Rightarrow Pos(Red^\alpha(P))} \sigma'(t_l)$. In order to obtain a contradiction, assume that $\longrightarrow_P \models \sigma'(t_l) \stackrel{a_l}{\longrightarrow} s'$ for some s'. Then $\longrightarrow_{Pos(Red^\alpha(P))} \models \sigma(t_l) \stackrel{a_l}{\longrightarrow} s$ for some s. So by lemma 5.5.5 for all $\beta < \alpha$: $\longrightarrow_{Pos(Red^\beta(P))} \models \sigma(t_l) \stackrel{a_l}{\longrightarrow} s$. This cannot be the case, as $\bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^\beta(P))} \models \sigma(t_l) \stackrel{a_l}{\not\longrightarrow}$. □

From claim 2 it follows that there is a substitution σ' such that $\longrightarrow_P \models prem(\sigma'(r'))$. Hence $\longrightarrow_P \models conc(\sigma'(r')) = v \stackrel{a}{\longrightarrow} \sigma'(t)$. Finally, as for all $x \in Var(r')$: $\sigma(x) R_P \sigma'(x)$, $u' = \sigma(t) R_P \sigma'(t) = v'$.

□

Lemma 5.8.10. *Let P be a pure TSS in ntyft-format that is positive after reduction. Then $R_P = \leftrightarrows_P$.*

Proof. As P is positive after reduction for some ordinal α, $Red^\alpha(P)$ is positive. It follows using corollary 5.5.8 that:

$$\longrightarrow_P = \longrightarrow_{Red^\alpha(P)} = \longrightarrow_{Pos(Red^\alpha(P))} .$$

Now, it follows using lemma 5.8.9, the introduction of definition 5.7.2 and the definition of R_P that:

$$R_P \subseteq \leftrightarrows_{P \Rightarrow Pos(Red^\alpha(P))} = \leftrightarrows_P \subseteq R_P.$$

□

Theorem 5.8.11 *(Congruence theorem).* *Let P be a well-founded TSS in ntyft/ntyxt-format that is positive after reduction. Then $\leftrightarrows_P$ is a congruence.*

Proof. Assume $P = (\Sigma, A, R)$. According to lemma 5.8.6 there is a pure TSS $P' = (\Sigma, A, R')$ in *ntyft*-format that is positive after reduction such that $\longrightarrow_P = \longrightarrow_{P'}$. Hence, $\leftrightarrows_P = \leftrightarrows_{P'}$. By lemma 5.8.10 $\leftrightarrows_{P'} = R_{P'}$. As $R_{P'}$ is a congruence w.r.t. Σ, $\leftrightarrows_P$ is also a congruence w.r.t. Σ. □

The next example shows that the requirement in the congruence theorem 5.8.11 that the TSS P must be positive after reduction is really needed. We give a TSS in *ntyft/ntyxt*-format that has a unique stable transition relation but that is *not* positive after reduction and for which bisimulation is *not* a congruence.

Example 5.8.12. Let $P = (\Sigma, A, R)$ be a TSS where Σ contains constants c_1 and c_2 and a unary function f. The actions in A are a, b_1, b_2 and the rules are the following:

$$\text{E1:}\quad c_1 \xrightarrow{a} c_1 \qquad\qquad \text{E2:}\quad c_2 \xrightarrow{a} c_2$$

$$\text{E3:}\quad \frac{x \xrightarrow{a} y \quad f(x) \overset{b_1}{\not\longrightarrow} \quad f(c_1) \overset{b_2}{\not\longrightarrow}}{f(x) \xrightarrow{b_2} c_2} \qquad \text{E4:}\quad \frac{x \xrightarrow{a} y \quad f(x) \overset{b_2}{\not\longrightarrow} \quad f(c_2) \overset{b_1}{\not\longrightarrow}}{f(x) \xrightarrow{b_1} c_1}.$$

Note that P is pure and in *ntyft*-format. $Red^1(P)$ is a TSS with the following rules:

$$\text{E1}' :\quad c_1 \xrightarrow{a} c_1 \qquad\qquad \text{E2}' :\quad c_2 \xrightarrow{a} c_2$$

$$\text{E3}' :\quad \frac{f(c_1) \overset{b_1}{\not\longrightarrow} \quad f(c_1) \overset{b_2}{\not\longrightarrow}}{f(c_1) \xrightarrow{b_2} c_2} \qquad \text{E3}'' :\quad \frac{f(c_2) \overset{b_1}{\not\longrightarrow} \quad f(c_1) \overset{b_2}{\not\longrightarrow}}{f(c_2) \xrightarrow{b_2} c_2}$$

$$\text{E4}' :\quad \frac{f(c_1) \overset{b_2}{\not\longrightarrow} \quad f(c_2) \overset{b_1}{\not\longrightarrow}}{f(c_1) \xrightarrow{b_1} c_1} \qquad \text{E4}'' :\quad \frac{f(c_2) \overset{b_2}{\not\longrightarrow} \quad f(c_2) \overset{b_1}{\not\longrightarrow}}{f(c_2) \xrightarrow{b_1} c_1}.$$

Further reduction of P is not possible. However, we observe that both in E3′ and E4″ the conclusion denies the second premise. Therefore, a transition relation that is stable for P must deny the first premise of E3′ and of E4″, i.e. it must contain $f(c_1) \xrightarrow{b_1} t_1$ and $f(c_2) \xrightarrow{b_2} t_2$ for some t_1 and t_2. The only candidates that might be provable are $f(c_1) \xrightarrow{b_1} c_1$ and $f(c_2) \xrightarrow{b_2} c_2$. Indeed they are provable from E3″ and E4′ (as blocking E3′ and E4″ implies $f(c_1) \overset{b_2}{\not\longrightarrow}$ and $f(c_2) \overset{b_1}{\not\longrightarrow}$), so $\{c_1 \xrightarrow{a} c_1,\ c_2 \xrightarrow{a} c_2,\ f(c_1) \xrightarrow{b_1} c_1,\ f(c_2) \xrightarrow{b_2} c_2\}$ is the unique transition relation that is stable for P. Now it is obvious that $c_1 \leftrightarrows_P c_2$, but not $f(c_1) \leftrightarrows_P f(c_2)$, so $\leftrightarrows_P$ is not a congruence.

5.9 Conservative extensions of TSS's

It can be useful to enrich a given language with additional language constructs (like in our running example, where $\mathrm{BPA}_{\delta\epsilon\tau}$ is enriched with the priority and unless operator). For these new constructs operational rules are devised which are added to the operational semantics of the basic language. In this section we study how an operational semantics can be extended and especially how we can guarantee that transitions between terms in the basic language are not effected by the extension.

In this section we assume that the operational semantics of the basic language is given by a TSS P_0. All extensions, i.e. the added signature, label set and operational rules are given in a TSS P_1. The extension of P_0 with P_1 is written

as $P_0 \oplus P_1$ [14]. Due to the symmetric nature – we could as well extend P_1 with P_0 – this is called the *sum* of P_0 and P_1.

Definition 5.9.1. Let $\Sigma_i = (F_i, rank_i)$ $(i = 0, 1)$ be two signatures such that for all $f \in F_0 \cap F_1$: $rank_0(f) = rank_1(f)$. The *sum* of Σ_0 and Σ_1, notation $\Sigma_0 \oplus \Sigma_1$, is the signature:

$$\Sigma_0 \oplus \Sigma_1 = (F_0 \cup F_1, \lambda f.\text{if } f \in F_0 \text{ then } rank_0(f) \text{ else } rank_1(f)).$$

Definition 5.9.2. Let $P_i = (\Sigma_i, A_i, R_i)$ $(i = 0, 1)$ be two TSS's with $\Sigma_0 \oplus \Sigma_1$ defined. The *sum* of P_0 and P_1, notation $P_0 \oplus P_1$, is the TSS:

$$P_0 \oplus P_1 = (\Sigma_0 \oplus \Sigma_1, A_0 \cup A_1, R_0 \cup R_1).$$

If P_0 is extended with P_1 such that 'the properties' of P_0 are maintained, $P_0 \oplus P_1$ is said to be a *conservative extension* of P_0. With 'properties' of P_0 we mean transitions that can be performed by terms over the signature of P_0. To be more precise:

Definition 5.9.3. Let $P_i = (\Sigma_i, A_i, R_i)$ $(i = 0, 1)$ be two TSS's such that P_0 has associated transition relation $\longrightarrow_{P_0}$. Let $P_0 \oplus P_1$ with associated transition relation $\longrightarrow_{P_0 \oplus P_1}$ be defined. We say that $P_0 \oplus P_1$ is a *conservative extension* of P_0 and that P_1 *can be added conservatively* to P_0 if

$$\longrightarrow_{P_0 \oplus P_1} \cap (T(\Sigma_0) \times (A_0 \cup A_1) \times T(\Sigma_0 \oplus \Sigma_1)) = \longrightarrow_{P_0} .$$

An alternative definition has been given in [14]. Adapting that definition to our terminology, it says that $P = P_0 \oplus P_1 = (\Sigma, A, R)$ with associated transition relation $\longrightarrow_P$ is a conservative extension of $P_0 = (\Sigma_0, A_0, R_0)$ if for all $t \in T(\Sigma_0)$, $a \in A$ and $t' \in T(\Sigma)$:

$$\longrightarrow_P \models t \xrightarrow{a} t' \quad \Leftrightarrow \quad \longrightarrow_{P_0} \models t \xrightarrow{a} t'.$$

where $\longrightarrow_{P_0}$ is associated with P_0.

We now head for a theorem that gives conditions under which P_1 can be added conservatively to P_0. It turns out that this is the case if P_0 is pure and each rule in P_1 contains a function name in the source of its conclusion that does not appear in the signature of P_0. This theorem considerably extends the results in the previous chapter in which a comparable theorem was proved for TSS's in *ntyft/ntyxt*-format. If our result is restricted to this format, both results coincide, except that here, we deal with TSS's that are positive after reduction while in chapter 4 only stratified TSS's were considered.

Lemma 5.9.4. *Let $\Sigma_0 = (F_0, rank_0)$ be a signature. Let $P_0 = (\Sigma_0, A_0, R_0)$ be a pure TSS and let $P_1 = (\Sigma_1, A_1, R_1)$ be a TSS such that $P_0 \oplus P_1$ is defined and for each rule $r \in R_1$: $source(conc(r)) \notin \mathbb{T}(\Sigma_0)$. Then, for each ordinal α:*

$$\longrightarrow_{Pos(Red^\alpha(P_0 \oplus P_1))} \cap (T(\Sigma_0) \times (A_0 \cup A_1) \times T(\Sigma_0 \oplus \Sigma_1)) = \longrightarrow_{Pos(Red^\alpha(P_0))} \quad (5.1)$$

$$\longrightarrow_{True(Red^\alpha(P_0 \oplus P_1))} \cap (T(\Sigma_0) \times (A_0 \cup A_1) \times T(\Sigma_0 \oplus \Sigma_1)) = \longrightarrow_{True(Red^\alpha(P_0))} \quad (5.2)$$

Proof. We prove clauses (5.1) and (5.2) by simultaneous induction on α.

(5.1) $\subseteq$ For this case it is sufficient to show the following:

$$Pos(Red^{\alpha}(P_0 \oplus P_1)) \vdash t \xrightarrow{a} t' \text{ and } t \in T(\Sigma_0) \text{ implies}$$

$$Pos(Red^{\alpha}(P_0)) \vdash t \xrightarrow{a} t',\ a \in A_0 \text{ and } t' \in T(\Sigma_0).$$

So assume that $Pos(Red^{\alpha}(P_0 \oplus P_1)) \vdash t \xrightarrow{a} t'$ and $t \in T(\Sigma_0)$. We use induction on the depth of this proof. Let the last rule of this proof be $r \in Pos(Red^{\alpha}(R_0 \oplus R_1))$. Then $conc(r) = t \xrightarrow{a} t'$. Hence, as $t \in T(\Sigma_0)$ and all rules in R_1 contain a function name $f \notin \Sigma_0$ in the source of their conclusions, r is derived from a rule $\sigma(r')$ with $r' \in R_0$. So $a \in A_0$.

Claim 1. For all $x \in Var(r')$: $\sigma(x) \in T(\Sigma_0)$.

Proof of claim 1. As r' is pure, it is well-founded, so $pprem(r')$ has a variable dependency graph VDG. We prove the claim by induction on $n_{VDG}(x)$. Consider some x with $n_{VDG}(x) = \gamma$ and assume the claim holds for all x' such that $n_{VDG}(x') < \gamma$. As r' does not contain free variables, one of the following two cases must hold:

1. $x \in Var(source(conc(r'))$. As $\sigma(source(conc(r')) \in T(\Sigma_0)$, $\sigma(x) \in T(\Sigma_0)$.
2. $x \in Var(u')$ and $u \xrightarrow{a} u' \in pprem(r')$. For all $x' \in Var(u)$: $n_{VDG}(x') < n_{VDG}(x)$ and therefore $\sigma(x') \in T(\Sigma_0)$. Hence, $\sigma(u) \in T(\Sigma_0)$. Distinguish the following two cases:
   (a)
   $$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0 \oplus P_1))} \models \sigma(u) \xrightarrow{a} \sigma(u').$$
   Then by (5.2),
   $$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0))} \models \sigma(u) \xrightarrow{a} \sigma(u')$$
   and this means that $\sigma(u') \in T(\Sigma_0)$. Therefore, as $x \in Var(u')$, $\sigma(x) \in T(\Sigma_0)$.
   (b)
   $$\bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0 \oplus P_1))} \not\models \sigma(u) \xrightarrow{a} \sigma(u').$$
   Then $\sigma(u) \xrightarrow{a} \sigma(u') \in pprem(r)$ and therefore,
   $$Pos(Red^{\alpha}(P_0 \oplus P_1)) \vdash \sigma(u) \xrightarrow{a} \sigma(u').$$

By induction (on the proof tree) it follows that:

$$Pos(Red^{\alpha}(P_0)) \vdash \sigma(u) \xrightarrow{a} \sigma(u')$$

and $\sigma(u') \in T(\Sigma_0)$. Hence, $\sigma(x) \in T(\Sigma_0)$.

□

As r is derived from reducing $\sigma(r')$ we have the following:

$$\begin{array}{l} \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P_0 \oplus P_1))} \models pprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0 \oplus P_1))} \models nprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0 \oplus P_1))} \models pprem(\sigma(r')) - prem(r). \end{array}$$

As by claim 1 $\sigma : Var(r') \to T(\Sigma_0)$ it follows using the outermost induction hypothesis that:

$$\begin{array}{l} \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P_0))} \models pprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0))} \models nprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0))} \models pprem(\sigma(r')) - prem(r). \end{array}$$

Or in other words $r \in Pos(Red^{\alpha}(R_0))$. By induction on the proof tree and claim 1 it follows that $Pos(Red^{\alpha}(P_0)) \vdash prem(r)$ and therefore

$$Pos(Red^{\alpha}(P_0)) \vdash t \xrightarrow{a} t' = \sigma(conc(r')) \in Tr(\Sigma_0, A_0).$$

(5.1) $\supseteq$ For this case it is sufficient to prove (using induction on the proof tree for $Pos(Red^{\alpha}(P_0)) \vdash t \xrightarrow{a} t'$) that:

$$Pos(Red^{\alpha}(P_0)) \vdash t \xrightarrow{a} t' \quad \Rightarrow \quad Pos(Red^{\alpha}(P_0 \oplus P_1)) \vdash t \xrightarrow{a} t'.$$

So assume $r \in Pos(Red^{\alpha}(R_0))$ is the last rule used in the proof for $t \xrightarrow{a} t'$. Hence there is a rule $r' \in R_0$ and a substitution $\sigma : Var(r') \to T(\Sigma_0)$ with $conc(\sigma(r')) = conc(r)$, $prem(r) \subseteq pprem(\sigma(r'))$. Moreover:

$$\begin{array}{l} \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P_0))} \models pprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0))} \models nprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0))} \models pprem(\sigma(r')) - prem(r). \end{array}$$

As for each premise $\psi \in prem(\sigma(r'))$, $source(\psi) \in T(\Sigma_0)$, we have by induction:

$$\begin{array}{l} \bigcap_{\beta<\alpha} \longrightarrow_{Pos(Red^{\beta}(P_0 \oplus P_1))} \models pprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0 \oplus P_1))} \models nprem(\sigma(r')), \\ \bigcup_{\beta<\alpha} \longrightarrow_{True(Red^{\beta}(P_0 \oplus P_1))} \models pprem(\sigma(r')) - prem(r). \end{array}$$

Hence, $r \in Pos(Red^{\alpha}(R_0 \oplus R_1))$. As $Pos(Red^{\alpha}(P_0)) \vdash t \xrightarrow{a} t'$, it holds that $Pos(Red^{\alpha}(P_0)) \vdash \psi$ for each $\psi \in prem(r)$. By induction $Pos(Red^{\alpha}(P_0 \oplus P_1)) \vdash \psi$ and hence, $Pos(Red^{\alpha}(P_0 \oplus P_1)) \vdash t \xrightarrow{a} t'$.

(5.2) This case can be shown in the same way as (5.1).

□

Theorem 5.9.5 *(Conservativity).* *Let $\Sigma_0 = (F_0, rank_0)$ be a signature. Let $P_0 = (\Sigma_0, A_0, R_0)$ be a pure TSS and let $P_1 = (\Sigma_1, A_1, R_1)$ be a TSS such that each rule $r \in R_1$ contains at least one function name $f \notin F_0$ in the source of its conclusion. Furthermore, assume that $P_0 \oplus P_1$ exists and is positive after reduction. Then $P_0 \oplus P_1$ is a conservative extension of P_0.*

Proof. As $P_0 \oplus P_1$ is positive after reduction, there is some ordinal α such that $Red^\alpha(P_0 \oplus P_1)$ is a positive TSS. Hence, $P_0 \oplus P_1$ has an associated transition relation $\longrightarrow_{P_0 \oplus P_1}$. Let $A = A_0 \cup A_1$ and $\Sigma = \Sigma_0 \oplus \Sigma_1$. By lemma 5.9.4 we have:

$$\begin{array}{l} \longrightarrow_{Pos(Red^\alpha(P_0))} = \\ \longrightarrow_{Pos(Red^\alpha(P_0 \oplus P_1))} \cap (T(\Sigma_0) \times A \times T(\Sigma)) = \\ \longrightarrow_{True(Red^\alpha(P_0 \oplus P_1))} \cap (T(\Sigma_0) \times A \times T(\Sigma)) = \\ \longrightarrow_{True(Red^\alpha(P_0))} . \end{array}$$

Hence by remark 5.5.2, $Red^{\alpha+1}(P_0)$ is a positive TSS. Therefore P_0 also has an associated transition relation $\longrightarrow_{P_0}$. Moreover, using corollary 5.5.8 and lemma 5.9.4 we have:

$$\begin{array}{l} \longrightarrow_{P_0} = \\ \longrightarrow_{True(Red^{\alpha+1}(P_0))} = \\ \longrightarrow_{True(Red^{\alpha+1}(P_0 \oplus P_1))} \cap (T(\Sigma_0) \times A \times T(\Sigma)) = \\ \longrightarrow_{P_0 \oplus P_1} \cap (T(\Sigma_0) \times A \times T(\Sigma)). \end{array}$$

□

Remark 5.9.6. From the alternative definition of conservativity it is immediately obvious that if $P_0 \oplus P_1$ is a conservative extension of $P_0 = (\Sigma_0, A_0, R_0)$ then for all $t, u \in T(\Sigma_0)$: $t \leftrightarrows_{P_0} u \;\Leftrightarrow\; t \leftrightarrows_{P_0 \oplus P_1} u$.

Example 5.9.7. We can apply the conservativity theorem to show that the priority operator and the unless operator form a conservative extension of $\mathrm{BPA}_{\delta\epsilon\tau}$. We can also conservatively add the parallel operator which is characterised by the following rules

$$10.1 \quad \frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \qquad\qquad 10.2 \quad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$$

to $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities. In fact in almost all cases the addition of new operators to an existing TSS turns out to be conservative.

$x + (y + z) = (x + y) + z$	A1	$a\tau = a$	T1
$x + y = y + x$	A2	$\tau x + x = \tau x$	T2
$x + x = x$	A3	$a(\tau x + y) = a(\tau x + y) + ax$	T3
$(x + y)z = xz + yz$	A4		
$(xy)z = x(yz)$	A5		
$x + \delta = x$	A6	$\theta(\epsilon) = \epsilon$	THE
$\delta x = \delta$	A7	$\theta(\delta) = \delta$	THD
$\epsilon x = x$	A8	$\theta(ax) = a\theta(x)$	TH1
$x\epsilon = x$	A9	$\theta(x + y) = \theta(x) \lhd y + \theta(y) \lhd x$	TH2
$\epsilon \lhd x = \epsilon$	PE1	$\tau_I(\epsilon) = \epsilon$	TIE
$x \lhd \epsilon = x$	PE2	$\tau_I(\delta) = \delta$	TID
$\delta \lhd x = \delta$	PD1	$\tau_I(a) = a$ if $a \notin I$	TI1
$x \lhd \delta = x$	PD2	$\tau_I(a) = \tau$ if $a \in I$	TI2
$ax \lhd by = \delta$ if $(a < b)$	P1	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI3
$ax \lhd cy = ax$ if $\neg(a < c)$	P2	$\tau_I(xy) = \tau_I(x)\tau_I(y)$	TI4
$ax \lhd \tau y = ax \lhd y$ if $\neg(a < \tau)$	P3		
$x \lhd (y + z) = (x \lhd y) \lhd z$	P4		
$(x + y) \lhd z = x \lhd z + y \lhd z$	P5		

Table 5.2: The axiom set $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau}$ ($a, b \in Act_\tau$ and $c \in Act$)

5.10 An axiomatisation of priorities with abstraction

This last section is devoted to our running example. We consider an instance $P_\theta = (\Sigma_\theta, A_\theta, R_\theta)$ of $\mathrm{BPA}_{\delta\epsilon\tau}$ with priorities such that for all $(X \Leftarrow t_X) \in E$: $\tau_I(\cdot)$ does not occur in t_X and for all $a \in Act$ it does not hold that $\tau < a$. By theorem 5.6.6 P_θ has an associated transition relation $\longrightarrow_{P_\theta}$.

In table 5.2 we list the axiom set $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau}$ for strong bisimulation equivalence induced by P_θ. This axiom system consists of a straightforward assembly of existing axioms [1, 17], adding only the axiom P3 showing the interaction between $\lhd$ and τ. Nevertheless, as far as we know, this straightforward compilation has not been justified in bisimulation semantics. Only in [26] τ and θ have been combined using an isomorphic embedding.

This section is added to show how an axiom system can be proved sound and complete with respect to an operational semantics, even if this semantics is defined using negative premises. We give all essential lemmas and theorems but only some insightful parts of the proofs. Most proofs apply induction on proof trees (standard for positive TSS's) of the 'stripped' TSS. This leads to a more general observation: induction on proof trees derived from a 'stripped' TSS is a powerful proof tool for TSS's with negative premises.

Definition 5.10.1. Let $\Sigma = (F, rank)$ be a signature and let Eq be a set of axioms over Σ. Let $R_{Eq} \subseteq T(\Sigma) \times T(\Sigma)$ be the smallest congruence relation satisfying that $tR_{Eq}u$ if $t = u$ is a ground instance of an axiom in Eq. For terms $t, u \in T(\Sigma)$, we say that Eq *proves* $t = u$, notation $Eq \vdash t = u$, if $tR_{Eq}u$.

The following lemma says how behaviour of a complex term can be explained in terms of necessary behaviour of its components. This lemma is first used in [25] to prove the soundness of the axioms. Due to the rules R9.2 and R9.3, the proof of this lemma is lengthy.

Lemma 5.10.2 *(Structuring lemma).* *Let $t, u, v \in T(\Sigma_\theta)$ and $a \in A_\theta$.*
If $t + u \xrightarrow{a} v$ then one of the following must hold:

1. $t \xrightarrow{a} v$,
2. $u \xrightarrow{a} v$.

If $t \cdot u \xrightarrow{a} v$ then one of the following must hold:

1. $t \xrightarrow{a} t'$, $v \equiv t' \cdot u$ *and* $a \not\equiv \surd$ *for some* $t' \in T(\Sigma_\theta)$,
2. $t \xrightarrow{\surd} t'$ *and* $u \xrightarrow{a} v$ *for some* $t' \in T(\Sigma_\theta)$,
3. $t \xrightarrow{a} t'$, $t' \xrightarrow{\surd} t''$, $u \xrightarrow{\tau} v$ *and* $a \not\equiv \surd$ *for some* $t', t'' \in T(\Sigma_\theta)$.

If $\theta(t) \xrightarrow{a} u$ then one of the following must hold:

1. $t \xrightarrow{a} t'$, $u \equiv \theta(t')$, $a \not\equiv \surd$ *and* $\forall b > a\ t \not\xrightarrow{b}$ *for some* $t' \in T(\Sigma_\theta)$,
2. $t \xrightarrow{\tau} t'$, $t' \xrightarrow{a} t''$, $u \equiv \theta(t'')$, $a \not\equiv \surd$ *and* $\forall b > a\ t' \not\xrightarrow{b}$ *for some* $t', t'' \in T(\Sigma_\theta)$,
3. $t \xrightarrow{\surd} t'$, $u \equiv \theta(t')$ *and* $a \equiv \surd$ *for some* $t' \in T(\Sigma_\theta)$.

If $t \triangleleft u \xrightarrow{a} v$ then one of the following must hold:

1. $t \xrightarrow{a} v$, $a \not\equiv \surd$ *and* $\forall b > a\ u \not\xrightarrow{b}$,
2. $t \xrightarrow{\tau} t'$, $t' \xrightarrow{a} v$ *and* $a \not\equiv \surd$ *for some* $t' \in T(\Sigma_\theta)$,
3. $t \xrightarrow{\surd} v$ *and* $a \equiv \surd$.

If $\tau_I(t) \xrightarrow{a} u$ then one of the following must hold:

1. $t \xrightarrow{a_1} t_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} t_n \xrightarrow{a} t_{n+1} \xrightarrow{a_{n+2}} \ldots \xrightarrow{a_m} t_m$, $a \notin I$ *and* $u \equiv \tau_I(t_m)$ *for some* $a_1, .., a_n, a_{n+2}, .., a_m \in I$, $t_1, .., t_m \in T(\Sigma_\theta)$, $n \geq 0$ *and* $m \geq 1$.
2. $t \xrightarrow{a_1} t_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} t_n$, $a \equiv \tau$ *and* $u \equiv \tau_I(t_n)$ *for some* $a_1, .., a_n \in I$, $t_1, .., t_n \in T(\Sigma_\theta)$ *and* $n \geq 1$.

Proof. As an illustration, we give the proof for $\theta(t) \stackrel{a}{\longrightarrow} u$ in case $a \not\equiv \surd$. All other proofs can be given in the same way.

If $\theta(t) \stackrel{a}{\longrightarrow} u$ then this is equivalent to saying that $\mathit{Strip}(P_\theta, \longrightarrow_{P_\theta}) \vdash \theta(t) \stackrel{a}{\longrightarrow} u$. We show with induction on the proof tree that $\mathit{Strip}(P_\theta, \longrightarrow_{P_\theta}) \vdash \theta(t) \stackrel{a}{\longrightarrow} u$ implies that one of the following holds:

1. $t \stackrel{a}{\longrightarrow} t'$, $u \equiv \theta(t')$ and $\forall b > a\ t \stackrel{b}{\not\longrightarrow}$ for some $t' \in T(\Sigma_\theta)$,
2. $t \stackrel{\tau}{\longrightarrow} t'$, $t' \stackrel{a}{\longrightarrow} t''$, $u \equiv \theta(t'')$ and $\forall b > a\ t' \stackrel{b}{\not\longrightarrow}$ for some $t', t'' \in T(\Sigma_\theta)$.

Suppose $\mathit{Strip}(P_\theta, \longrightarrow_{P_\theta}) \vdash \theta(t) \stackrel{a}{\longrightarrow} u$. The last rule that is used in this proof must either be R9.2, R9.3 or a stripped version of R5.1. Suppose a simplified version of rule R5.1 has been used. In this case the premises of R5.1, $t \stackrel{a}{\longrightarrow} t'$ and $\forall b > a\ t \stackrel{b}{\not\longrightarrow}$, hold in $\longrightarrow_{P_\theta}$. Furthermore, $u \equiv \theta(t')$. So case 1 of θ in the structuring lemma must hold.

If rule R9.2 has been used, we know that $\mathit{Strip}(P_\theta, \longrightarrow_{P_\theta}) \vdash \theta(t) \stackrel{\tau}{\longrightarrow} u'$ and $\mathit{Strip}(P_\theta, \longrightarrow_{P_\theta}) \vdash u' \stackrel{a}{\longrightarrow} u$. By induction one of the following four cases must hold:

1. $t \stackrel{\tau}{\longrightarrow} t'$, $t' \stackrel{a}{\longrightarrow} t''$, $\forall b > a\ t' \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t'')$,
2. $t \stackrel{\tau}{\longrightarrow} t'$, $t' \stackrel{\tau}{\longrightarrow} t''$, $t'' \stackrel{a}{\longrightarrow} t'''$, $\forall b > a\ t'' \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t''')$,
3. $t \stackrel{\tau}{\longrightarrow} t'$, $t' \stackrel{\tau}{\longrightarrow} t''$, $t'' \stackrel{a}{\longrightarrow} t'''$, $\forall b > a\ t'' \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t''')$,
4. $t \stackrel{\tau}{\longrightarrow} t'$, $t' \stackrel{\tau}{\longrightarrow} t''$, $t'' \stackrel{\tau}{\longrightarrow} t'''$, $t''' \stackrel{a}{\longrightarrow} t''''$, $\forall b > a\ t''' \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t'''')$.

In all cases it must hold that for some v and v':

$$t \stackrel{\tau}{\longrightarrow} v,\ v \stackrel{a}{\longrightarrow} v',\ \forall b > a\ v \stackrel{b}{\not\longrightarrow} \text{ and } u \equiv \theta(v').$$

Suppose rule R9.3 has been used as last step in the proof. As the premises of R9.3 are derivable, we have:

$$\mathit{Strip}(P_\theta, \longrightarrow_{P_\theta}) \vdash \theta(t) \stackrel{a}{\longrightarrow} u',\ \mathit{Strip}(P_\theta, \longrightarrow_{P_\theta}) \vdash u' \stackrel{\tau}{\longrightarrow} u.$$

By induction one of the following four cases must hold.

1. $t \stackrel{a}{\longrightarrow} t'$, $t' \stackrel{\tau}{\longrightarrow} t''$, $\forall b > a\ t \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t'')$,
2. $t \stackrel{a}{\longrightarrow} t'$, $t' \stackrel{\tau}{\longrightarrow} t''$, $t'' \stackrel{\tau}{\longrightarrow} t'''$, $\forall b > a\ t \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t''')$,
3. $t \stackrel{\tau}{\longrightarrow} t'$, $t' \stackrel{a}{\longrightarrow} t''$, $t'' \stackrel{\tau}{\longrightarrow} t'''$, $\forall b > a\ t' \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t''')$,
4. $t \stackrel{\tau}{\longrightarrow} t'$, $t' \stackrel{a}{\longrightarrow} t''$, $t'' \stackrel{\tau}{\longrightarrow} t'''$, $t''' \stackrel{\tau}{\longrightarrow} t''''$, $\forall b > a\ t' \stackrel{b}{\not\longrightarrow}$ and $u \equiv \theta(t'''')$.

From cases 1 or 2 it follows that (for appropriate $v \in T(\Sigma_\theta)$):

$$t \xrightarrow{a} v,\ \forall b > a\ t \overset{b}{\not\longrightarrow} \text{ and } u \equiv \theta(v)$$

which is case 1 for θ in the structuring lemma. From cases 3 or 4 it follows that (for appropriate $v, v' \in T(\Sigma_\theta)$):

$$t \xrightarrow{\tau} v \xrightarrow{a} v',\ \forall b > a\ v \overset{b}{\not\longrightarrow} \text{ and } u \equiv \theta(v')$$

which is case 2 for θ in the structuring lemma. □

With the structuring lemma it is rather straightforward, but unpleasantly lengthy, to prove the soundness of the axioms.

Theorem 5.10.3 *(Soundness of* $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau}$*).* *Let* $t, u \in T(\Sigma_\theta)$*:*

$$\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash t = u \quad \Rightarrow \quad t \leftrightarrows_{P_\theta} u.$$

Proof. We must show that $R_{\mathrm{BPA}^{\theta}_{\delta\epsilon\tau}} \subseteq \leftrightarrows_{P_\theta}$. As $R_{\mathrm{BPA}^{\theta}_{\delta\epsilon\tau}}$ is the smallest congruence relation containing (t, u) if $t = u$ is an instance of an axiom in $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau}$, and as by theorem 5.8.11 $\leftrightarrows_{P_\theta}$ is also a congruence relation, it is sufficient to show that

$$t = u \text{ is an instance of an axiom in } \mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \quad \Rightarrow \quad t \leftrightarrows_{P_\theta} u.$$

Suppose $t = u$ is an instance of an axiom. We will only consider axiom P3. All other axioms can be dealt with in the same way. Hence, $t \equiv at' \triangleleft \tau u'$, $u \equiv at' \triangleleft u'$ $(t', u' \in T(\Sigma_\theta))$ and $\neg(a < \tau)$. In order to show that $at' \triangleleft \tau u' \leftrightarrows_{P_\theta} at' \triangleleft u'$, it suffices to show that if $at' \triangleleft \tau u' \xrightarrow{b} v$, $(b \in A_\theta)$ then $at' \triangleleft u' \xrightarrow{b} v$ and vice versa, $at' \triangleleft u' \xrightarrow{b} v$ implies $at' \triangleleft \tau u' \xrightarrow{b} v$. So suppose $at' \triangleleft \tau u' \xrightarrow{b} v$. By the structuring lemma one of the following cases must hold:

1. $at' \xrightarrow{b} v$, $b \not\equiv \surd$ and $\forall c > b\ \tau u' \overset{c}{\not\longrightarrow}$,
2. $at' \xrightarrow{\tau} t''$, $t'' \xrightarrow{b} v$ and $b \not\equiv \surd$ for some $t'' \in T(\Sigma_\theta)$,
3. $at' \xrightarrow{\surd} v$ and $b \equiv \surd$.

Note that case 3 is impossible. So either case 1 or case 2 must hold. If case 2 holds, it is immediately clear that $at' \triangleleft u' \xrightarrow{\tau} t''$ and $t'' \xrightarrow{b} v$. Therefore, $at' \triangleleft u' \xrightarrow{b} v$. If case 1 holds, then $\forall c > b\ u' \overset{c}{\not\longrightarrow}$. If this were not the case, i.e. $\exists c > b\ u' \xrightarrow{c} u''$, then $\tau u' \xrightarrow{c} u''$ contradicting that $\forall c > b\ \tau u' \overset{c}{\not\longrightarrow}$. Hence $at' \triangleleft u' \xrightarrow{b} v$.

The other implication can be proved likewise. □

We now show completeness of the axioms. This is done in three stages. First the class of basic terms is introduced. This class is a subset of all closed Σ_θ-terms, but it is still powerful enough to denote all recursion free processes. This is in fact shown in lemma 5.10.6.

Then operational characteristics are linked to the syntactic forms of terms using the operational soundness and completeness lemmas. In the last lemma all results are gathered together and completeness is shown.

Definition 5.10.4. The set of *basic terms* is the smallest subset of $T(\Sigma_\theta)$ satisfying:

- δ and ϵ are basic terms,
- if t is a basic term, then at $(a \in Act_\tau)$ is a basic term,
- if t, t' are basic terms, then $t + t'$ is a basic term.

Note that $a\epsilon$ and $a\epsilon + b\delta$ are basic terms but a and $(a + b)c$ are not.

Lemma 5.10.5. *Let t, t' be basic terms. Then there is a basic term u such that:*

1. $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t \square t' = u$ *($\square = +, \cdot, \lhd$),*
2. $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash \square(t) = u$ *($\square = \tau_I, \theta$).*

Proof. As an example we show the proof for $\lhd$. For a basic term t define $\#t$ as the number of function names in t. Define the depth of a term $t \lhd t'$ with t, t' basic terms by (ω is the first infinite ordinal):

$$D(t \lhd t') = \omega \cdot \#t' + \#t.$$

We prove this case with induction on $D(t \lhd t')$. Distinguish the following cases:

$t = \epsilon, \delta$ Apply PE1 or PD1.

$t' = \epsilon, \delta$ Apply PE2 or PD2.

$t = au_1, t' = bu_2, (b \not\equiv \tau)$ Apply P1 or P2.

$t = au_1, t' = \tau u_2$ Apply P1 if $a < \tau$. If $\neg(a < \tau)$ then $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash au_1 \lhd \tau u_2 \overset{P3}{=} au_1 \lhd u_2$. As $D(au_1 \lhd u_2) < D(au_1 \lhd \tau u_2)$, it follows with induction that $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash au_1 \lhd u_2 = v$ for some basic term v.

$t' = u_1 + u_2$ We have that $t \lhd t' \equiv t \lhd (u_1 + u_2) \overset{P4}{=} (t \lhd u_1) \lhd u_2$. As $D(t \lhd u_1) < D(t \lhd (u_1 + u_2))$ it follows that $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t \lhd u_1 = v$ for some basic term v. As $D(v \lhd u_2) < D(t \lhd (u_1 + u_2))$ it follows that $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash (t \lhd u_1) \lhd u_2 = v \lhd u_2 = v'$ for some basic term v'.

$t = u_1 + u_2$ It follows that $t \lhd t' \equiv (u_1 + u_2) \lhd t' \stackrel{P5}{=} u_1 \lhd t' + u_2 \lhd t'$. As $D(u_1 \lhd t') < D((u_1 + u_2) \lhd t')$ and $D(u_2 \lhd t') < D((u_1 + u_2) \lhd t')$, there are basic terms v, v' such that $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash u_1 \lhd t' = v$ and $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash u_2 \lhd t' = v'$. Hence, $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash t \lhd t' = v + v'$.

□

Lemma 5.10.6. *Let $t \in T(\Sigma_\theta)$ be a recursion free term. Then there is a basic term u such that:*

$$\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash t = u.$$

Proof. Apply induction on the structure of t. If $t \equiv \epsilon, \delta, a (\in Act_\tau)$ then the basic terms are respectively: ϵ, δ and $a\epsilon$. If $t \equiv t_1 \Box t_2$ ($\Box \equiv +, \cdot, \lhd$), it follows with induction that t_1 and t_2 are provably equal to basic terms u_1, u_2. Then lemma 5.10.5 yields $\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash u_1 \Box u_2 = u$ with u a basic term. For the unary operators θ and τ_I a similar argument can be applied. □

The following notation is an abbreviation that turns out to be useful.

Notation 5.10.7 *(Summand inclusion).* We write $t \subseteq t'$ for $t + t' = t'$.

The following lemmas relate summand inclusion to the operational rules in table 5.1. They state that if a process t can perform an a-step ($t \stackrel{a}{\longrightarrow} t'$) then it is provable that at' is a summand of t. A weak variant of the converse also holds.

Lemma 5.10.8 *(Operational soundness).* *Let $t, t' \in T(\Sigma_\theta)$ be recursion free terms and let $a \in Act_\tau$:*

$$\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash a \cdot t' \subseteq t \quad \Rightarrow \quad \exists t'' : \; t \stackrel{a}{\longrightarrow} t'' \text{ and } t'' \leftrightarrows_{P_\theta} t',$$

$$\mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash \epsilon \subseteq t \quad \Rightarrow \quad \exists t' : \; t \stackrel{\surd}{\longrightarrow} t'.$$

Proof. Directly using the soundness theorem 5.10.3. □

Lemma 5.10.9 *(Operational completeness).* *Let $t, t' \in T(\Sigma_\theta)$ be recursion-free and $\theta, \lhd$-free terms and let $a \in Act_\tau$:*

$$t \stackrel{a}{\longrightarrow} t' \quad \Rightarrow \quad \mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash at' \subseteq t,$$

$$t \stackrel{\surd}{\longrightarrow} t' \quad \Rightarrow \quad \mathrm{BPA}^{\theta}_{\delta\epsilon\tau} \vdash \epsilon \subseteq t.$$

Proof. By induction on the proof of $t \stackrel{a}{\longrightarrow} t'$ and $t \stackrel{\surd}{\longrightarrow} t'$ from $Strip(P_\theta, \longrightarrow_{P_\theta})$.
□

Lemma 5.10.10. *Let t be a basic term. If $t \xrightarrow{a} t'$ ($a \in Act_\tau$), then $t' \equiv \epsilon \cdot u$ or $t' \equiv \tau \cdot u$ for some basic term u. Moreover, t' contains at most as many function names as t.*

Proof. Use induction on the proof of $t \xrightarrow{a} t'$ from $Strip(P_\theta, \longrightarrow_{P_\theta})$. □

Notation 5.10.11. Let $t, u \in T(\Sigma_\theta)$ be recursion free. $t \rightrightarrows_{P_\theta} u$ stands for: $t \xrightarrow{a} t'$ implies $\exists u'$ $u \xrightarrow{a} u'$ and $t' \leftrightarrows_{P_\theta} u'$. Note that this condition resembles clause 1 in the definition of bisimulation.

Lemma 5.10.12. *Let t and u be basic terms over $\mathrm{BPA}^\theta_{\delta\epsilon\tau}$. Then:*

1. *If $t \rightrightarrows_{P_\theta} u$ then $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t \subseteq u$,*
2. *If $t \leftrightarrows_{P_\theta} u$ then $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t = u$.*

Proof. We use induction on the number of function names in t and u, i.e. $\#t+\#u$. The proof employs the operational soundness and completeness lemmas.
Basis. First 1 is proved. Suppose that $t \equiv \epsilon$ and $u \in T(\Sigma_\theta)$. We have that

$$\epsilon \rightrightarrows_{P_\theta} u \;\Rightarrow\; u \xrightarrow{\surd} u' \;\Rightarrow\; \mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash \epsilon \subseteq u.$$

Suppose $t \equiv \delta$. This case is trivial using axiom A6. In case 2 $t \leftrightarrows_{P_\theta} u$ implies $t \rightrightarrows_{P_\theta} u$ and $u \rightrightarrows_{P_\theta} t$, so it follows by 1 that $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t \subseteq u$ and $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash u \subseteq t$. Hence $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t = t + u = u + t = u$.
Induction. First consider 1. Suppose $t \equiv (t_1 + t_2) \rightrightarrows_{P_\theta} u$. This implies that $t_1 \rightrightarrows_{P_\theta} u$ and $t_2 \rightrightarrows_{P_\theta} u$. Using 1 inductively yields: $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t_1 \subseteq u$ and $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t_2 \subseteq u$. Now using axiom A1 leads to $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t_1 + t_2 \subseteq u$.

Now suppose that $t \equiv at_1 \rightrightarrows_{P_\theta} u$. Note that $\#t_1 < \#t$. There is a t_2 (e.g. ϵt_1) such that $t \xrightarrow{a} t_2 \leftrightarrows_{P_\theta} t_1$. As $t \rightrightarrows_{P_\theta} u$, it follows that there is a u_1 such that $u \xrightarrow{a} u_1$, $t_1 \leftrightarrows_{P_\theta} t_2 \leftrightarrows_{P_\theta} u_1$. By lemma 5.10.10 u_1 is a basic term and $\#u_1 \leq \#u$. With the induction hypothesis conclude that $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t_1 = u_1$. By operational completeness it follows that $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash au_1 \subseteq u$. Therefore, $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash at_1 \subseteq u$.

In case 2 $t \leftrightarrows_{P_\theta} u$ implies $t \rightrightarrows_{P_\theta} u$ and $u \rightrightarrows_{P_\theta} t$, so it follows by 1 that $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t \subseteq u$ and $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash u \subseteq t$. Hence $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t = t + u = u + t = u$. □

Theorem 5.10.13 *(Completeness of $\mathrm{BPA}^\theta_{\delta\epsilon\tau}$). Let $t, u \in T(\Sigma_\theta)$ be recursion free. It holds that:*

$$t \leftrightarrows_{P_\theta} u \quad\Rightarrow\quad \mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t = u.$$

Proof. Suppose $t \leftrightarrows_{P_\theta} u$. Then there are basic terms t' and u' that are provably equivalent to t and u. With soundness it follows that $t' \leftrightarrows_{P_\theta} u'$. An application of lemma 5.10.12 yields $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t' = u'$ and thus $\mathrm{BPA}^\theta_{\delta\epsilon\tau} \vdash t = u$. □

5.11 Appendix: the relation between TSS's and logic programs

Throughout this paper techniques from logic programming are applied to TSS's. This raises the question whether TSS's can be viewed as logic programs. It appears that there exists indeed a straightforward translation from TSS's to logic programs.

Definition 5.11.1. Let $P = (\Sigma, A, R)$ be a TSS. We define the translation $\mathcal{L}$ as:

for every positive literal $t \xrightarrow{a} t'$, $\mathcal{L}(t \xrightarrow{a} t') = transition(t, a, t')$,
for every negative literal $t \not\xrightarrow{a}$, $\mathcal{L}(t \not\xrightarrow{a}) = \neg possible(t, a)$,
for every rule $r \in R$: $\mathcal{L}(r) = \mathcal{L}(conc(r)) \leftarrow \mathcal{L}(prem(r))$[1],

and finally

$$\mathcal{L}(P) = \mathcal{L}(R)^1 \cup \{possible(T, A) \leftarrow transition(T, A, U)\}$$

where T, A and U are variables.

For an introduction in logic programming we refer to [16]. We must point out some small differences between the two formalisms.

First of all, logic programs are usually untyped, whereas a TSS $P = (\Sigma, A, R)$ has clearly two types, namely terms (from $T(\Sigma)$) and labels (from A). Thus the translation $\mathcal{L}(P)$ must also be treated as a typed program, its Herbrand base being

$$\begin{aligned} HB_P \quad = \quad & \{transition(t, a, t') \mid t, t' \in T(\Sigma),\ a \in A\} \cup \\ & \{possible(t, a) \mid t \in T(\Sigma),\ a \in A\}. \end{aligned}$$

Secondly, a traditional logic program consists of a finite set of finite clauses. A TSS may have an infinite number of rules and each rule may have infinitely many premises. The main reason for this is that in TSS's only variables ranging over terms are used, and no variables ranging over labels. Thus instead of one rule like

$$\frac{x \xrightarrow{z} x'}{x + y \xrightarrow{z} x'},$$

this rule must be incorporated for every action z separately. Usually rule schemes with meta-variables ranging over A are given, like in this case rule R3.1 of the running example. Translating a TSS yields a possibly infinite set of possibly infinite clauses. Of course having an infinite number of clauses is not a problem: the set of ground instances of clauses from a traditional logic program is normally infinite as well. Having infinitely many premises seems harmless too.

In order to formulate the intended correspondencies between the TSS P and the logic program $\mathcal{L}(P)$, we also need a translation on the semantical level, i.e. between transitions relations and (well-typed) Herbrand interpretations.

[1] As usual $\mathcal{L}(X)$ abbreviates $\{\mathcal{L}(x) \mid x \in X\}$

Definition 5.11.2. Let $\longrightarrow$ be a transition relation.

$$\begin{array}{rcl}\mathcal{M}(\longrightarrow) & = & \{transition(t,a,t') \mid t \xrightarrow{a} t' \in \longrightarrow\} \cup \\ & & \{possible(t,a) \mid \exists t' :\ t \xrightarrow{a} t' \in \longrightarrow\}.\end{array}$$

According to this definition only interpretations M satisfying for all t and a:

$$possible(t,a) \in M \quad \text{iff} \quad \exists t' :\ transition(t,a,t') \in M$$

are translations of a transition relation. The clause

$$possible(T,A) \leftarrow transition(T,A,U)$$

is obviously incorporated in the translation of every TSS to enforce this property. As long as only supported models of the resulting logic programs are considered, the addition of this clause in indeed sufficient. The following example shows that a weaker choice of semantics (in this case minimal models) can produce certain anomalous models.

Example 5.11.3. Consider the TSS P with one constant c and one unary function f, one action a and the following rules:

$$\frac{c \xrightarrow{a} /}{c \xrightarrow{a} c} \qquad\qquad \frac{c \xrightarrow{a} x}{c \xrightarrow{a} f(x)}.$$

For all $n \geq 0$, the transition relation $\{c \xrightarrow{a} f^i(c) \mid i \geq n\}$ is a model of P; P has no other models. As n increases, the model decreases (w.r.t. $\subseteq$), thus P has no minimal model. Now consider

$$\begin{array}{rcl}\mathcal{L}(P) & = & \{transition(c,a,c) \leftarrow \neg possible(c,a) \\ & & \ transition(c,a,f(X)) \leftarrow transition(c,a,X) \\ & & \ possible(T,A) \leftarrow transition(T,A,U)\}.\end{array}$$

The corresponding models are (for all $n \geq 0$):

$$\{transition(c,a,f^i(c)) \mid i \geq n\} \cup \{possible(c,a)\}.$$

But $\mathcal{L}(P)$ has one more model, namely just $\{possible(c,a)\}$, which is the least model of $\mathcal{L}(P)$, but not supported by $\mathcal{L}(P)$.

As we concentrate on the stable and well-founded model semantics, which generate only supported models, anomalous models will no more arise.

In the rest of this section we establish the relationships between TSS's and their translations into logic programs. For the definitions regarding logic programming we refer to the literature. As these definitions are always similar to the definitions regarding TSS's as presented in this paper, it is straightforward to prove the following propositions.

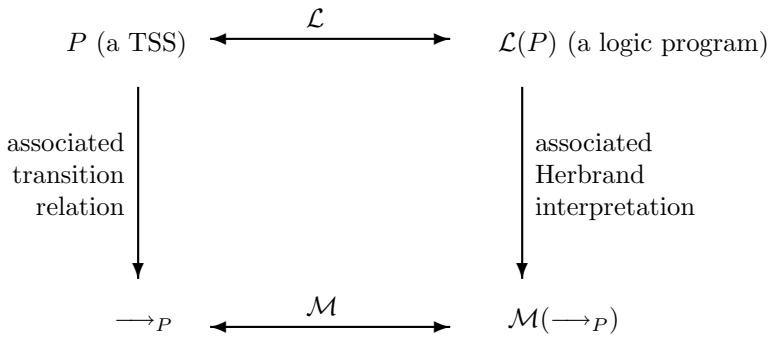


Figure 5.2: The relation between TSS's and logic programs

Proposition 5.11.4. Let P be a TSS.

- P is positive iff $\mathcal{L}(P)$ is positive.
- P is stratified iff $\mathcal{L}(P)$ is locally stratified (see [21]).

For a positive logic program P, M_P denotes its least Herbrand model.

Proposition 5.11.5. Let P be a TSS and $\longrightarrow$ be a transition relation.
$\longrightarrow$ is stable for P iff $\mathcal{M}(\longrightarrow)$ is a stable model of $\mathcal{L}(P)$ (see [11]).
In particular, if P is positive, then $\mathcal{M}(\longrightarrow_P) = M_{\mathcal{L}(P)}$ and
if P is stratified, then $\mathcal{M}(\longrightarrow_P)$ is the unique perfect model of $\mathcal{L}(P)$ (see [21]).

Two slightly different, but equivalent, definitions of well-founded models for logic programs have been given ([10] and W_P in [24] versus [22] and V_P in [24]). Here we follow [22].

Definition 5.11.6 *(well-founded model).* Let P be a logic program.

- A 3-valued interpretation for P is a pair $I =< T, F >$, where T and F are subsets of the Herbrand base HB_P (but not necessarily $T \cap F = \emptyset$).
- Let A be a ground atom. Then $< T, F > \models A$ iff $A \in T$ and $< T, F > \models \neg A$ iff $A \in F$.
- $T_P(I) = \{A \in HB_P \mid$ there exists a clause $A \leftarrow L_1, ..., L_n \in ground(P)$ such that $I \models L_1$ and ... and $I \models L_n\}$.

  $F_P(I) = \{A \in HB_P \mid$ for every clause $A \leftarrow L_1, ..., L_n \in ground(P)$: $I \models \neg L_1$ or ... or $I \models \neg L_n\}$. (If L is a negative literal $\neg B$, then $\neg L$ denotes B.)

  $T_P(I)$ defines the ground atoms that are immediately true given P and I, $F_P(I)$ defines the ground atoms that are immediately false.

– Let $T, F \subseteq HB_P$ and let I be a 3-valued interpretation for P.

$\mathcal{T}_I(T) = T_P(I \cup < T, \emptyset >)$.
$\mathcal{F}_I(F) = F_P(I \cup < \emptyset, F >)$.
$\mathcal{I}_P(I) = I \ \cup \ < \bigcup_{n<\omega} \mathcal{T}_I^n(\emptyset), \ \bigcap_{n<\omega} \mathcal{F}_I^n(HB_P) >$.
(Note: $\cup$ denotes pointwise union.)

$\mathcal{I}_P(I)$ defines the ground atoms that are certainly true respectively false given P and I.

– For a limit ordinal α: $I_\alpha = \bigcup_{\beta<\alpha} I_\beta$ (in particular: $I_0 =< \emptyset, \emptyset >$).
For a successor ordinal $\alpha + 1$: $I_{\alpha+1} = \mathcal{I}_P(I_\alpha)$.

– Let δ be the smallest countable ordinal such that $I_\delta = \mathcal{I}(I_\delta)$. Then I_δ is the well-founded (partial) model of P. If I_δ is 2-valued, i.e. $I_\delta =< T, F >$ is a partitioning of HB_P, then I_δ is the well-founded (complete) model of P.

An alternative definition of the well-founded model, based on the reduction of logic programs, can also be given.

Definition 5.11.7. Let P be a logic program and I a 3-valued interpretation for P. Then:

$$Reduce(P, I) = \bigcup_{C \in ground(P)} Reduce(C, I), \text{ where}$$
$$Reduce(A \leftarrow S, I) = \begin{cases} \emptyset & \text{if for some literal } L \in S: \ I \models \neg L \\ \{A \leftarrow S'\} & \text{otherwise, where } S' = \{L \in S \mid I \not\models L\}. \end{cases}$$

Furthermore:

$$True(P) = \{A \leftarrow S \in P \mid S \text{ contains only positive literals}\} \text{ and}$$
$$Pos(P) = \{A \leftarrow S' \mid \text{there is a clause } A \leftarrow S \in P \text{ such that } S' \text{ is the set of positive literals in } S\}.$$

Lemma 5.11.8. *Let P be a logic program and I a 3-valued interpretation for P.*

$$\bigcup_{n<\omega} \mathcal{T}_I^n(\emptyset) = M_{True(Reduce(P,I))},$$
$$\bigcap_{n<\omega} \mathcal{F}_I^n(HB_P) = HB_P - M_{Pos(Reduce(P,I))}.$$

Thus an alternative definition of the well-founded (partial) model of a logic program is obtained by replacing

$\bigcup_{n<\omega} \mathcal{T}_I^n(\emptyset)$ by $M_{True(Reduce(P,I))}$ and
$\bigcap_{n<\omega} \mathcal{F}_I^n(HB_P)$ by $HB_P - M_{Pos(Reduce(P,I))}$

in definition 5.11.6. The proof of lemma 5.11.8 is beyond the scope of this paper.

Using this alternative definition, it is straightforward to link the reduction of a TSS P and the sequence of interpretations leading to the well-founded (partial) model of $\mathcal{L}(P)$.

Lemma 5.11.9. *Let $P = (\Sigma, A, R)$ be a TSS and let $\longrightarrow_{true}, \longrightarrow_{pos} \subseteq Tr(\Sigma, A)$. Then:*

- $\mathcal{L}(True(P)) = True(\mathcal{L}(P))$,
- $\mathcal{L}(Pos(P)) = Pos(\mathcal{L}(P))$,
- $\mathcal{L}(Reduce(P, \longrightarrow_{true}, \longrightarrow_{pos})) = Reduce(\mathcal{L}(P), < \mathcal{M}(\longrightarrow_{true}), \mathrm{HB}_P - \mathcal{M}(\longrightarrow_{pos}) >)$.

Theorem 5.11.10. *Let P be a TSS. Let for all ordinals α, I_α be defined w.r.t. $\mathcal{L}(P)$ as in definition 5.11.6. Then:*

$$\mathcal{L}(Red^\alpha(P)) = Reduce(\mathcal{L}(P), I_\alpha),$$

$$I_\alpha = < \bigcup_{\beta<\alpha} \mathcal{M}(\longrightarrow_{True(Red^\beta(P))}), \bigcup_{\beta<\alpha} \mathrm{HB}_P - \mathcal{M}(\longrightarrow_{Pos(Red^\beta(P))}) > .$$

Proof. Straightforward. □

Corollary 5.11.11. *Let P be a TSS. If $\mathcal{L}(P)$ has a well-founded complete model I_α, then $Red^\alpha(P)$ is a positive TSS and $I_\alpha = \mathcal{M}(\longrightarrow_{Red^\alpha(P)})$. If $Red^\alpha(P)$ is a positive TSS then $\mathcal{L}(P)$ has a well-founded complete model $I_{\alpha+1} = \mathcal{M}(\longrightarrow_{Red^\alpha(P)})$.*

The change from α to $\alpha + 1$ in the second implication is caused by the fact that it is possible that at the end of the iteration first the interpretation I_α becomes 2-valued (making $Reduce(\mathcal{L}(P), I_\alpha)$ positive), but also that a partial I_α results in a positive $Reduce(\mathcal{L}(P), I_\alpha)$, in which case only $I_{\alpha+1}$ is 2-valued.

Apart from its theoretical merits, the translation of TSS's into logic programs has also more practical implications. For logic programs interpreters and compilers are available. Thus in order to find out whether a term t can perform an a-step according to $\longrightarrow_P$, the TSS P is translated into the logic program $\mathcal{L}(P)$, and the query $\leftarrow transition(t, a, X)$ is presented to it.

For positive TSS's this poses only one problem: the depth-first strategy of most programming systems tends to result in non-termination of the program without finding all solutions. The rules R9.2 and R9.3 of the running example are typically rules leading to non-termination. Incorporating certain forms of loop-checking ([6, 27]) might partly solve the problem, but as even for positive TSS's $\longrightarrow_P$ need not be recursive, non-termination can never be ruled out completely.

In the presence of negation $\longrightarrow_P$ is in general not even recursive enumerable and the execution of the logic program becomes even more involved (but see [22, 24], where 'ideal' mechanisms are proposed for computing the well-founded model, abstracting away from non-termination). Thus the translation into logic programming cannot be expected to produce the associated transition relation as a whole. But in our opinion the interactive use of a logic programming environment for proving that a certain transition holds (or does not hold) is an attractive alternative to generating this proof by hand, especially for larger TSS's.

On the bright side is that for pure TSS's (see definition 5.8.3) the problem of *floundering* (the necessity to resolve a non-ground negative literal, see e.g. [16]) does not occur for queries of the form $\leftarrow transition(t, a, X)$ (with $t \in T(\Sigma)$ and $a \in A$). This can be shown by *annotating* the program (in the sense of [9]) by $transition(\downarrow, \downarrow, \uparrow)$ and $possible(\downarrow, \downarrow)$, meaning that the first and second argument of both predicates are considered to be input, and the third argument of *transition* is output. (Due to the fact that TSS's have no variables ranging over labels, the annotations of the second (label) arguments are inessential.)

Proposition 5.11.12. Let $P = (\Sigma, A, R)$ be a TSS. Let $t \in T(\Sigma)$ and $a \in A$. If P is pure then $\mathcal{L}(P) \cup \{\leftarrow transition(t, a, X)\}$ is well-formed (see [9]) w.r.t. the above annotation.

The well-formedness of a logic program and a query implies that during the computation every predicate is called with ground terms on its input arguments. In particular, every call $\neg possible(t, a)$ will be ground. In a more general setting, this annotation gives insight in the data-flow of the act of proving transitions from pure TSS's.

References

[1] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fund. Inf.*, XI(2):127–168, 1986.

[2] E. Best and M. Koutny. Partial order semantics of priority systems. Technical Report 6/90, Universität Hildesheim, Institut für Informatik, 1990.

[3] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription. In *Proceedings IEEE Conference on Logic in Computer Science* (LICS87), New York, pages 89–97, 1987.

[4] N. Bidoit and C. Froidevaux. Negation by default and non stratifiable logic programs. Technical Report 437, L.R.I. France, 1988.

[5] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: preliminary report. In *Conference Record of the* 15^{th} *ACM Symposium on Principles of Programming Languages,* San Diego, California, pages 229–239, 1988.

[6] R.N. Bol, K.R. Apt, and J.W. Klop. An analysis of loop checking mechanisms for logic programs. Technical Report CS-R8947, CWI, Amsterdam, 1989. To appear in *Theoretical Computer Science*.

[7] J. Camilleri. An operational semantics for OCCAM. *International Journal of Parallel Programming*, 18(5):149–167, October 1989.

[8] R. Cleaveland and M. Hennessy. Priorities in process algebra. In *Proceedings third Annual Symposium on Logic in Computer Science* (LICS), Edinburgh, pages 193–202, 1988.

[9] P. Dembinski and J. Małuszynski. And-parallelism with intelligent backtracking for annotated logic programs. In *Symposion on Logic Programming,* Boston, pages 29–38, 1985.

[10] A. van Gelder, K. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the Seventh Symposium on Principles of Database Systems*, pages 221–230. ACM SIGACT-SIGMOD, 1988.

[11] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. MIT press, 1988.

[12] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings STACS 87,* LNCS 247, pages 336–347. Springer-Verlag, 1987.

[13] R.J. van Glabbeek. The linear time - branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings Concur90,* Amsterdam, LNCS 458, pages 278–297. Springer Verlag, 1990.

[14] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. Technical Report CS-R8845, CWI, Amsterdam, 1988. An extended abstract appeared in G. Ausiello, M. Dezani-Ciancaglini and, S. Ronchi Della Rocca, editors, *Proceedings ICALP 89*, Stresa, LNCS 372, pages 423–438. Springer-Verlag, 1989.

[15] M. Hennessy and G.D. Plotkin. Full abstraction for a simple programming language. In J. Bečvář, editor, *Proceedings Eighth Symposium on Mathematical Foundations of Computer Science* (MFCS), LNCS 74, pages 108–120. Springer-Verlag, 1979.

[16] J.W. Lloyd. *Foundations of Logic Programming, Second Edition.* Springer-Verlag, 1987.

[17] R. Milner. *A Calculus of Communicating Systems.* LNCS 92. Springer-Verlag, 1980.

[18] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings Fifth GI Conference,* LNCS 104, pages 167–183. Springer-Verlag, 1981.

[19] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[20] H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Logic Programming Symposium*, pages 1106–1120. MIT press, 1988.

[21] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers Inc., Los Altos, California, 1987.

[22] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the Eighth Symposium on Principles of Database Systems*, pages 11–21. ACM SIGACT-SIGMOD, 1989.

[23] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[24] K. Ross. A procedural semantics for well founded negation in logic programs. In *Proceedings of the Eighth Symposium on Principles of Database Systems*, pages 22–33. ACM SIGACT-SIGMOD, 1989.

[25] F.W. Vaandrager. Specificatie en verificatie van communicatieprotocollen met procesalgebra. Unpublished, in Dutch.

[26] F.W. Vaandrager. *Algebraic Techniques for Concurrency and their Application.* PhD thesis, University of Amsterdam, 1990.

[27] L. Vieille. Recursive query processing: the power of logic. *Theoretical Computer Science*, 69(1):1–53, 1989.

6

ACP with Real-Time Steps

(Jan Friso Groote)

An extension of ACP (Algebra of Communicating Processes) is proposed to describe and verify real time systems using a discrete time scale. This language is called $\mathrm{ACP}^t_{\tau\epsilon}$. $\mathrm{ACP}_{\tau\epsilon}$ is the timeless variant of $\mathrm{ACP}^t_{\tau\epsilon}$. $\mathrm{ACP}^t_{\tau\epsilon}$ unifies real time, abstraction, parallelism and communication in one algebraic framework.

We give a set of axioms describing the properties of $\mathrm{ACP}^t_{\tau\epsilon}$, together with a corresponding operational semantics. Furthermore, we define an interpretation of $\mathrm{ACP}_{\tau\epsilon}$ in $\mathrm{ACP}^t_{\tau\epsilon}$, thereby fixing an intuition behind time in $\mathrm{ACP}_{\tau\epsilon}$. Some examples show the use of $\mathrm{ACP}^t_{\tau\epsilon}$.

6.1 Introduction

Distributed computer systems working on a real time basis find an increasing number of applications. Therefore, it is worthwhile to develop formal techniques in order to construct reliable and correct real time computer systems. We call any system a real time system if its interactions with the environment must satisfy certain time constraints. These constraints can be of various kinds, e.g. maximal reaction times, minimal delay times etc.

The importance of real time is illustrated by the huge amount of papers that have been published in recent years. We mention just a few [1, 12, 19, 26, 29], most of them introducing real time into process algebra in different ways. It is our feeling that most of these approaches are complicated, which makes it difficult to apply these. Therefore, we start from a simple idea, namely that the proceeding of some fixed amount of time is represented by an action, which we denote by t and which we call a *time step*, and elaborate on it in the well developed setting of ACP [3, 4, 5] obtaining $\mathrm{ACP}^t_{\tau\epsilon}$. The same basic idea underlies the work in [18, 23, 24, 27] but the exact interpretation of t differs in all of these. Technically, our approach is closest to the work of Richier, Sifakis and Voiron [27].

The idea turns out to work well, both for the development of theory and in examples. Because we can use the techniques available in ACP we have parallelism

and abstraction almost for free in our framework. We present an operational model in Plotkin style together with a complete axiom system for weak bisimulation semantics [22]. Weak bisimulation is arbitrarily chosen. The theory could as well be developed using for instance branching bisimulation [16], ready trace semantics, failure semantics etc. (see [14]). We introduce several delay operators and we illustrate their use by an example. Finally, we show how $\mathrm{ACP}_{\tau\epsilon}$ processes can be interpreted in $\mathrm{ACP}^t_{\tau\epsilon}$. This interpretation makes the notion of time underlying $\mathrm{ACP}_{\tau\epsilon}$ explicit.

6.2 The language and its axioms

We first present $\mathrm{ACP}_{\tau\epsilon}$ which is a slight modification of ACP_τ [5]. $\mathrm{ACP}_{\tau\epsilon}$ does not contain explicit constructs for time. As usual it has two parameters, a set Act and a function γ. The possibly infinite set Act contains *atomic actions* representing the different basic activities of these processes. The function $\gamma : Act \times Act \to Act$ is called the *communication function*. It is a partial, associative and commutative function that defines how actions in Act can synchronise in order to obtain interactions between processes. The signature of $\mathrm{ACP}_{\tau\epsilon}$ is given in table 6.1, except for the constant t which is not part of it.

The elementary identities that a process algebra over the signature of $\mathrm{ACP}_{\tau\epsilon}$ should satisfy are those listed in table 6.2 (again except those referring to t). In this table a, b range over $Act_\delta = Act \cup \{\delta\}$ and x, y, z are variables. The axioms differ from the axioms given in [5, 6] because we have an ϵ that allows us to formulate the equations a little bit more efficiently. Moreover, we have adapted the axioms for the communication merge in combination with internal actions. Especially we have replaced the axiom $\tau x \mid y = x \mid y$ by EM11, EM12 and EM13. This has no influence on the main operators, as is shown by the operational model. The effect of this change is that an interesting relation between $\mathrm{ACP}_{\tau\epsilon}$ with and without time steps holds (see theorem 6.6.6). At the end of this article the example is given that shows why $\tau x \mid y = x \mid y$ does not hold.

Equalities between process terms can be calculated using the normal inference rules of equational logic. For completeness' sake, these rules are summarised in table 6.3. O_1 stands for ∂_H or τ_I and O_2 may be replaced by $+, \cdot, \|, \lfloor\!\lfloor$ or $|$. A set T of axioms proves an equation $p = q$, notation $T \vdash p = q$, if there is an equational logic proof tree with root $p = q$ from the axioms in T in the usual sense.

The extension $\mathrm{ACP}_{\tau\epsilon}$ with time steps is called $\mathrm{ACP}^t_{\tau\epsilon}$. In $\mathrm{ACP}^t_{\tau\epsilon}$ actions are considered to be pointwise in time, i.e. their execution does not take time. Especially, τ is time-less. It is also assumed that the execution of the operators is time-less, for instance the execution of the sequential composition operator is instantaneous.

Time is modelled by a constant t, the *time step*, that represents the progress of exactly one time unit and not more than that. Intuitively we think of t as a fixed physical amount of time, such as for instance a millisecond or a minute. We

$\Sigma(\mathrm{ACP}^t_{\tau\epsilon})$:	constants	a	for any atomic action $a \in Act$
		δ	inaction
		τ	silent action
		ϵ	empty process
		t	time step
	unary operators:	∂_H	encapsulation, for any $H \subseteq Act$
		τ_I	hiding, for any $I \subseteq Act$
	binary operators:	$+$	alternative composition
		$\cdot$	sequential composition
		$\parallel$	parallel composition
		$\lfloor\!\lfloor$	left-merge
		$\mid$	communication-merge

Table 6.1: The signature of $\mathrm{ACP}^t_{\tau\epsilon}$

also assume that only t allows time to go on. A consequence of this view is that inaction (δ) cannot let time go on. Therefore, it can be called *time-deadlock* or *time-stop*. This means that time can be blocked, which is impossible in reality. However, we think that blocking time is rather useful for analysis of real time systems. For instance, when two systems are composed in parallel in such a way that one system *must* synchronise at times the other is not available, analysis will reveal that time cannot proceed from a certain point onwards without violating the behaviour of one of the subsystems. In this case a *time-stop* indicates the time inconsistency in the composition. It is of course clear that only systems without time-stop can be implemented.

Example 6.2.1. We will try to give the reader some idea about the properties of the language $\mathrm{ACP}^t_{\tau\epsilon}$. Consider the $\mathrm{ACP}^t_{\tau\epsilon}$-expressions:

1. $a \cdot t \cdot b + c \cdot t \cdot t \cdot d$,
2. $a \cdot b$,
3. $a^\omega = a \cdot a \cdot a \cdot \ldots$.

The first process must immediately perform either an a or a c action. If it starts with an a action, then after exactly one time unit it must perform a b. If it initially selects a c action, then two time units later, it must do a d. The second process must perform an a, instantaneously followed by a b; a and b happen at the same time, only b depends causally on a. The last process illustrates an imperfection in the current approach. Here a^ω is the process that can do an infinite number of a actions in no time. Later we will give constructs that enable us to define a^ω precisely. Intuitively, such processes are not very appealing but we do not feel that they will yield any problem when using our algebra. However, if this turns out not be the case, we can easily get rid of them by a time guardedness constraint on processes (cf. [25]).

The set of axioms of $\mathrm{ACP}^t_{\tau\epsilon}$ in table 6.2 characterises the basic properties of the function symbols in the signature. The axioms about time only state that time must proceed synchronously in parallel processes (TIM1–TIM5).

One can ask why there is no axiom $tx + ty = t(x + y)$ (*Time Determinism* (DT)) which is explicitly present in [18, 23] and implicitly in [24]. The reason lies in the interaction with the sequential composition operator. Consider the term

$$(t\,t + t)\,t\,t\,t\,a.$$

It would be somewhat strange if at time 3 it is *not* determined whether a will happen at time 4 or at time 5 because the tail of the process seems completely deterministic. But with DT and A4 we can derive:

$$\begin{aligned}&(t\,t + t)\,t\,t\,t\,a =\\ &t\,t\,t\,t\,t\,a + t\,t\,t\,t\,a =\\ &t\,t\,t\,(t\,t\,a + t\,a).\end{aligned}$$

This last term suggests that at time 3 the choice still has to be made. If we want to avoid such examples, the moment of choice for t actions is important. If one insists on having DT then one must drop A4. But A4 is so fundamental for sequential composition, that without A4, it is better to drop the sequential composition operator altogether. This approach is chosen in [18, 23, 24] where action prefix operators are used. As we want to stay in the realm of ACP, we retain the sequential composition operator and do not have DT.

Note that with time nondeterminism, time deterministic processes can easily be described. But with time determinism, time nondeterministic processes cannot be described in a straightforward way. Consider a process where at time 1 the choice is determined between two actions a and b that should happen at time 2. This can be described by $t(ta + tb)$. With time determinism this process is equal to $t\,t(a + b)$ where the choice is made at time 2. In [1] internal actions must be used explicitly to indicate such an early choice.

An additional advantage of time nondeterminism is that the theory stays closer to existing theory. This means that it is in general straightforward to combine real-time steps with other features. For instance, the current setting can easily be extended with priorities (see the article in chapter 5 of this thesis). Such an extension may be a good basis for relating the present work to other real-time step proposals.

As usual in ACP, we introduce a *guarded recursive specification* as a set of guarded recursive equations, used to specify infinite processes.

Definition 6.2.2. Let p be an open term over the signature $\mathrm{ACP}^t_{\tau\epsilon}$. An occurrence of a variable x in p is *guarded* if it occurs in a subterm $a \cdot p'$ of p ($a \in A_{\delta t}$) and p does not contain the τ_I operator. p is *guarded* if all variables occur guarded in p.

$x + (y + z) = (x + y) + z$	A1	$a\tau = a$	T1a
$x + y = y + x$	A2	$\tau\tau = \tau$	T1b
$x + x = x$	A3	$t\tau = t$	T1t
$(x + y)z = xz + yz$	A4	$\tau \cdot x + x = \tau \cdot x$	T2
$(xy)z = x(yz)$	A5	$a(\tau x + y) = a(\tau x + y) + ax$	T3
$x + \delta = x$	A6	$t(\tau x + y) = t(\tau x + y) + tx$	T3t
$\delta x = \delta$	A7		
$\epsilon x = x$	A8	$a \mid b = \gamma(a, b)$ if $\gamma(a, b)$ defined	
$x\epsilon = x$	A9	$a \mid b = \delta$ if $\gamma(a, b)$ undefined	
$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$	EM1	$\tau x \lfloor\!\lfloor y = \tau(x \parallel y)$	EM10
$\epsilon \lfloor\!\lfloor x = \delta$	EM2	$tx \lfloor\!\lfloor y = \delta$	TIM1
$ax \lfloor\!\lfloor y = a(x \parallel y)$	EM3	$\epsilon \mid tx = \delta$	TIM2
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	EM4	$\epsilon \mid \tau x = \epsilon \mid x$	EM11
$x \mid y = y \mid x$	EM5	$\tau x \mid ay = x \mid ay$	EM12
$\epsilon \mid \epsilon = \epsilon$	EM6	$\tau x \mid \tau y = \tau(x \parallel y)$	EM13
$\epsilon \mid ax = \delta$	EM7	$tx \mid ty = t(x \parallel y)$	TIM3
$ax \mid by = (a \mid b)(x \parallel y)$	EM8	$tx \mid ay = \delta$	TIM4
$(x + y) \mid z = x \mid z + y \mid z$	EM9	$tx \mid \tau y = tx \mid y$	TIM5
$\partial_H(\tau) = \tau$	DT	$\tau_I(\tau) = \tau$	TI1
$\partial_H(t) = t$	TID	$\tau_I(t) = t$	TIT
$\partial_H(\epsilon) = \epsilon$	DE	$\tau_I(\epsilon) = \epsilon$	TE
$\partial_H(a) = a$ if $a \notin H$	D1	$\tau_I(a) = a$ if $a \notin I$	TI2
$\partial_H(a) = \delta$ if $a \in H$	D2	$\tau_I(a) = \tau$ if $a \in I$	TI3
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI4
$\partial_H(xy) = \partial_H(x)\partial_H(y)$	D4	$\tau_I(xy) = \tau_I(x)\tau_I(y)$	TI5

Table 6.2: Axioms for $\mathrm{ACP}^t_{\tau\epsilon}$

$x = x$ $\quad \dfrac{x = y}{y = x}$ $\quad \dfrac{x = y \;\; y = z}{x = z}$ $\quad p = q$ if $p = q$ is an axiom	
$\dfrac{x = y}{O_1(x) = O_1(y)}$	$\dfrac{x_1 = y_1 \;\; x_2 = y_2}{O_2(x_1, x_2) = O_2(y_1, y_2)}$

Table 6.3: Rules of equational logic

Definition 6.2.3. A set E of equations of the form $\{x_i = p_i | i \in I\}$ over the signature $\mathrm{ACP}^t_{\tau\epsilon}$, with I an index set, is called a *recursive specification over* $\mathrm{ACP}^t_{\tau\epsilon}$ if for every variable x in p_i $(i \in I)$, $x \equiv x_j$ for some $j \in I$. E is called a *guarded recursive specification over* $\mathrm{ACP}^t_{\tau\epsilon}$ if E is a recursive specification and if every p_i $(i \in I)$ is guarded.

Notation 6.2.4. Let E be a guarded recursive specification in which a variable x occurs. The (unique) solution for x in E is written as $\langle x|E\rangle$.

The construct $\langle x|E\rangle$ may occur as a process term. It will be treated as a constant. So the process term $a \cdot a \cdot \langle x|x = a\,x\rangle$ is a valid term, representing of course a^ω. Any term not containing these constants is called *recursion free*. The fact that $\langle x|E\rangle$ is the solution of x in E is expressed by the following axiom:

$$\text{REC: } \langle x|E\rangle = \langle p_x|E\rangle \quad \text{if } x = p_x \in E.$$

$\langle p_x|E\rangle$ is an abbreviation for the term p_x where every occurrence of a variable Y in p_x is replaced by $\langle Y|E\rangle$. If REC is used in a proof, then this is written as $\ldots + \mathrm{REC} \vdash p = q$.

The principle RSP (*Recursive Specification Principle*) makes it explicit that each guarded recursive equation has exactly one solution. Using the notation for the solution of a guarded recursive specification, we can state RSP as follows (see [15]):

Principle 6.2.5. *Let E be a guarded recursive specification containing the variable x.*

$$\textit{RSP: } \frac{E}{x = \langle x|E\rangle}$$

This rule must be read as follows. Suppose we can prove the equations in $\sigma(E)$ with σ a substitution. Here $\sigma(E)$ is an abbreviation for $\{\sigma(x) = \sigma(p_x) | x = p_x \in E\}$. This means that we can show that $\sigma(x)$ is a solution for x in E. Then, RSP says that $\sigma(x)$ is equal to the unique solution $\langle x|E\rangle$ for x in E. Hence, each solution for x in E is the same.

Inference rules generated by RSP can occur in proofs, besides the normal rules of equational logic. If this is the case, this is written as $\ldots + \mathrm{RSP} \vdash p = q$.

Example 6.2.6. A typical real time device is a stopwatch. It can be used to measure time durations. Suppose the stopwatch has as visible actions *go, stop, reset* and $read_n$ for all time values $n \in \mathbb{N}$ with their obvious meanings. The stopwatch is given by the following infinite system of guarded recursive equations. Here $Stopwatch(i)$ and $\overline{Stopwatch(i)}$ are variables for every $i \in \mathbb{N}$. The process that we are interested in is $\langle Stopwatch(0)|E\rangle$ where E is defined by:

$$\begin{aligned} E = \{Stopwatch(i) \quad = \quad & stop \cdot \overline{Stopwatch(i)} + \\ & read_i \cdot Stopwatch(i) + \\ & t \cdot Stopwatch(i+1), \end{aligned}$$

$$\begin{aligned}\overline{Stopwatch(i)} &= go \cdot Stopwatch(i) + \\ &\quad reset \cdot \overline{Stopwatch(0)} + \\ &\quad read_i \cdot \overline{Stopwatch(i)} + \\ &\quad t \cdot \overline{Stopwatch(i)} | i \in \mathbb{N}\}.\end{aligned}$$

The stopwatch can either be measuring time (represented by $Stopwatch(i)$), in which case its internal counter is incremented every time unit, or its internal counter can be stopped, which is indicated by the barred variant of the stopwatch. Note that it is not possible that the action *go* and *reset* take place when the internal counter is counting, while *stop* cannot happen when the internal counter is stopped. The counter contains also other internal design choices. One of them is that *go* and *stop* can succeed each other infinitely fast, which may be hard to build in a real system.

6.3 Delays

In this section we introduce several delay processes and study some of their properties. Delay processes can wait an indefinite (but sometimes bounded) amount of time before terminating. They can be used to specify that actions must happen in certain time intervals.

The *(general) delay*, which we denote by Δ, is defined as follows:

$$\Delta = \langle x | x = t \cdot x + \epsilon \rangle.$$

The process Δ can always terminate or wait one more time unit.

The general delay must not be confused with delays that occur elsewhere in the literature. In synchronous calculi [21, 28] Δ (also used as operator and written as $\delta(p)$ or $\Delta(p)$) is the process that can do an arbitrary number of 1 actions before terminating (or performing p). In [3, 8, 9] Δ is used to represent divergence, i.e. the possibility to perform unbounded internal activity.

The general delay immediately suggests a variant, the *bounded delay* Δ_n. The process Δ_n can wait a maximal number of n time units before terminating. It is defined by:

$$\Delta_n = \langle x_n | E \rangle$$

where E is a recursive specification containing the equations ($n \in \mathbb{N}$):

$$\begin{aligned}x_0 &= \epsilon, \\ x_{n+1} &= t \cdot x_n + \epsilon.\end{aligned}$$

Example 6.3.1. We can now specify the following processes

1. $\Delta a \Delta b \Delta$,
2. $\Delta a \Delta_{15} b \Delta$.

The first process can once do an action a and then time can pass. Possibly, after some time action b can happen, but this is not necessary. In the second case action b *must* happen after the occurrence of a within at most 15 time units.

Lemma 6.3.2. *We have the following identities concerning* Δ *and* Δ_n*, provable using the axioms* $\mathrm{ACP}^t_{\tau\epsilon}$*,* REC *and* RSP*:*

1. $\Delta = \Delta + \epsilon$,
2. $\Delta\Delta = \Delta$,
3. $\tau\Delta = \Delta\tau\Delta$,
4. $\Delta_n\Delta = \Delta$.

Proof.

1. $\Delta = t\Delta + \epsilon = t\Delta + \epsilon + \epsilon = \Delta + \epsilon$.
2. Take as an equation: $X = tX + \Delta + \epsilon$. Then $\Delta\Delta$ and Δ are both solutions of this equation.

$$\Delta\Delta = (t\Delta + \epsilon)\Delta = t\Delta\Delta + \Delta = t\Delta\Delta + \Delta + \epsilon$$

$$\Delta = \Delta + \Delta = t\Delta + \epsilon + \Delta$$

   Hence, with RSP, $\Delta\Delta = \Delta$.
3. We show that $\tau\Delta$ and $\Delta\tau\Delta$ are both solutions of $X = tX + \tau\Delta$.

$$\tau\Delta = \Delta + \tau\Delta = t\Delta + \epsilon + \tau\Delta = t\Delta + \tau\Delta = t\tau\Delta + \tau\Delta$$

$$\Delta\tau\Delta = (t\Delta + \epsilon)\tau\Delta = t\Delta\tau\Delta + \tau\Delta$$

4. By induction on n. $(n = 0)$ $\Delta_n\Delta = \epsilon\Delta = \Delta$.
   $(n \geq 0)$ $\Delta_{n+1}\Delta = (t\Delta_n + \epsilon)\Delta = t\Delta_n\Delta + \Delta = t\Delta + \Delta =$
   $t\Delta + \epsilon + \Delta = \Delta + \Delta = \Delta$.

□

We remark that the identity $\Delta_n \cdot \Delta_m = \Delta_{n+m}$ is not derivable. Consider for instance the case where $m = n = 1$. Then $\Delta_m \cdot \Delta_n = t(t + \epsilon) + t + \epsilon$. On the other hand $\Delta_2 = t(t + \epsilon) + \epsilon$. These two processes are not equal as in the former case a t can be done after which termination must take place, while in the latter case this option is absent.

Lemma 6.3.3. *With the axioms in* $\mathrm{ACP}^t_{\tau\epsilon}$ *and* REC *we can show that* Δ *distributes over* $+$*, prefixed actions and the empty process.*

1. $(x+y) \between \Delta = x \between \Delta + y \between \Delta$,
2. $ax \between \Delta = a(x \between \Delta)$ $\qquad (a \in Act_{\tau t\delta})$,
3. $\epsilon \between \Delta = \epsilon$.

Proof.

1. First note that $\Delta \lfloor\!\lfloor x = t\Delta \lfloor\!\lfloor x + \epsilon \lfloor\!\lfloor x = \delta$. Now we have
   $(x+y) \between \Delta = (x+y) \lfloor\!\lfloor \Delta + \Delta \lfloor\!\lfloor (x+y) + (x+y) \mid \Delta =$
   $x \lfloor\!\lfloor \Delta + y \lfloor\!\lfloor \Delta + t\Delta \lfloor\!\lfloor (x+y) + \epsilon \lfloor\!\lfloor (x+y) + x \mid \Delta + y \mid \Delta =$
   $x \lfloor\!\lfloor \Delta + x \mid \Delta + y \lfloor\!\lfloor \Delta + y \mid \Delta =$
   $x \lfloor\!\lfloor \Delta + \Delta \lfloor\!\lfloor x + x \mid \Delta + \Delta \lfloor\!\lfloor y + y \lfloor\!\lfloor \Delta + x \mid \Delta = x \between \Delta + y \between \Delta$.
2. $ax \between \Delta = ax \lfloor\!\lfloor \Delta + \Delta \lfloor\!\lfloor ax + ax \mid t\Delta + ax \mid \epsilon =$
   $a(x \between \Delta) = a(x \between \Delta)$ for $a \in Act$,

   $tx \between \Delta = tx \lfloor\!\lfloor \Delta + \Delta \lfloor\!\lfloor tx + tx \mid t\Delta + tx \mid \epsilon = tx \mid t\Delta = t(x \between \Delta)$,

   $\delta x \between \Delta = \delta \lfloor\!\lfloor \Delta + \Delta \lfloor\!\lfloor \delta + \delta \mid \Delta = \delta \mid t\Delta + \delta \mid \epsilon = \delta = \delta(x \between \Delta)$,

   $\tau x \between \Delta = \tau x \lfloor\!\lfloor \Delta + \Delta \lfloor\!\lfloor \tau x + \Delta \mid \tau x = \tau(x \between \Delta) + t\Delta \mid \tau x + \epsilon \mid \tau x =$
   $\tau(x \between \Delta) + t\Delta \mid x + \epsilon \mid x = \tau(x \between \Delta) + \Delta \mid x \overset{\mathrm{T2,EM1}}{=} \tau(x \between \Delta)$.
3. $\epsilon \between \Delta = \epsilon \lfloor\!\lfloor \Delta + \epsilon \lfloor\!\lfloor \epsilon + t\Delta \lfloor\!\lfloor \epsilon + \epsilon \mid t\Delta + \epsilon \mid \epsilon = \epsilon \mid \epsilon = \epsilon$.

□

Δ can always be delayed or terminated, controlled by its environment. For instance, Δa in $\partial_{\{a,\bar{a}\}}(\Delta a \between \bar{a})$ with $\gamma(a,\bar{a}) = a^*$ behaves as a and therefore, the delay in Δa is forced to terminate immediately. Sometimes, one wants to describe delays that can independently of the context decide to wait or to terminate. The context has to adapt itself in this case. For this purpose *autonomous delays* Γ_0, Γ_1 and Γ_2 are introduced:

$$\begin{aligned} \Gamma_0 &= \langle x | x = \tau t x + \tau \rangle \\ \Gamma_1 &= \langle x | x = \tau t x + \epsilon \rangle \\ \Gamma_2 &= \langle x | x = t x + \tau \rangle \end{aligned}$$

The τ's in Γ_i $(i = 0,1,2)$ indicate that by some internal activity, options to terminate or to proceed can be lost. In Γ_1, for instance, executing an internal τ-step makes immediate termination impossible.

The following theorem gives a number of derivable facts about the autonomous delays.

Lemma 6.3.4. $\mathrm{ACP}^t_{\tau\epsilon} + \mathrm{REC} + \mathrm{RSP} \vdash$

$$\begin{array}{rclcrclcrcl}
\Gamma_0 & = & \Gamma_0 + \tau & & \Gamma_1 & = & \Gamma_1 + \epsilon & & \Gamma_2 & = & \Gamma_2 + \tau \\
\Gamma_0\Gamma_0 & = & \tau\Gamma_0 & & \Gamma_1\Gamma_1 & = & \Gamma_1 & & \Gamma_2\Gamma_2 & = & \tau\Gamma_2 \\
\tau\Gamma_0 & = & \Gamma_0\tau\Gamma_0 & & \tau\Gamma_1 & = & \Gamma_1\tau\Gamma_1 & & \tau\Gamma_2 & = & \Gamma_2\tau\Gamma_2 \\
\Delta \parallel \Gamma_0 & = & \Gamma_0 & & \Delta \parallel \Gamma_1 & = & \Gamma_1 & & \Delta \parallel \Gamma_2 & = & \Gamma_2
\end{array}$$

Proof. The proofs have the same structure as the proof of lemma 6.3.2 □

It is possible to give the finite versions of the autonomous delays also. Here, we only define Γ_0^n as the finite variant of Γ_0 which will be used in the next example.

$$\Gamma_0^n = \langle x_n | E \rangle$$

where E consists of the equations ($i \in \mathbb{N}$):

$$\begin{array}{rcl}
x_0 & = & \epsilon, \\
x_{i+1} & = & \tau t\, x_i + \tau.
\end{array}$$

6.4 Example

In this section a *manufacturing workcell* is presented that shows how time can be used to describe and verify timed systems. In [20] some workcells have been verified in the setting of process algebra without time steps. In our system it is important that certain actions happen just because time has passed. Further, there are actions that cannot be guaranteed to happen at predefined times. In the system a datalink can deliver messages within a certain time bound, but the time the delivery takes is not exactly determined.

The workcell is structured as described in [10]. It consists of two *work stations*, called W_1 and W_2, which produce products, and a *transport system*, TS, that transports products from work station W_1 to W_2. In this case TS is thought of as a simple conveyor belt which can only carry one product at a time. The last component of the workcell is a *workcell controller* (WC) that coordinates all activities in the workcell.

Workcell W_1 needs one time unit to produce a product. Then the transport system takes exactly three time units to transfer this product to workcell W_2. Products that arrive at W_2 can enter W_2 by a small gate that can be opened and closed by W_2. Whenever a product arrives, it should always be open. It is assumed that this gate closes automatically when a product has entered W_2. This is not explicitly specified. The workcell controller is connected to the work stations by two asynchronous point to point datalinks D_1 and D_2 through which it can send coordinating messages to the workcells. D_1 connects WC with W_1 and D_2 connects WC with W_2. Datalink D_2 delivers its messages exactly in one

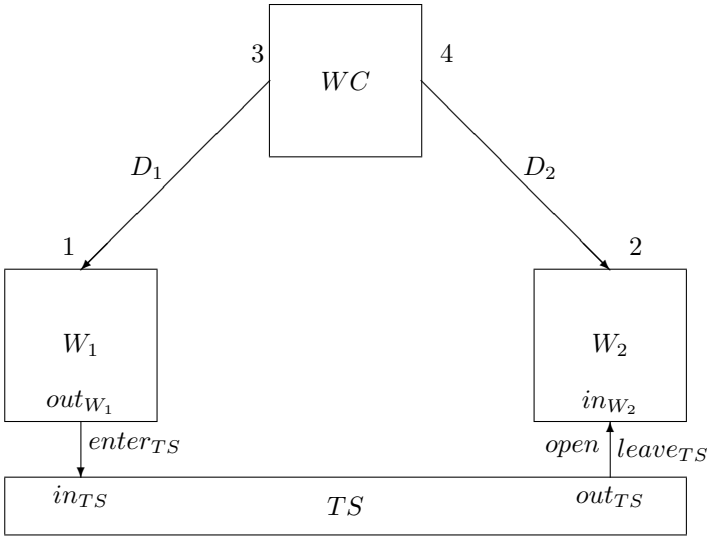


Figure 6.1: A manufacturing workcell

time unit. Datalink D_1 autonomously decides to do it in in 0,1 or 2 time units. The workcell controller first sends a message to W_1 to say that it must send a product to W_2 using the transport system. After some time it sends a message to W_2 saying that it must open its gate because a product is arriving. This is done in such a way that the gate is only open for the smallest possible amount of time. As the control system of this workcell is fairly primitive and there are uncertainties in the delivery time of datalink D_1, the transport system is not used in an optimal way.

The system is given as the solution of the following equations. See also figure 6.1. Variables are denoted by capital letters and actions by lower case letters for readability.

$$S = \partial_H(W_1 \parallel W_2 \parallel WC \parallel TS \parallel D_1 \parallel D_2)$$

with $H = \{r_i, s_i | i = 1,2,3,4\} \cup \{out_{W_1}, in_{W_2}, in_{TS}, out_{TS}\}$ and $\gamma(r_i, s_i) = c_i$, $\gamma(out_{W_1}, in_{TS}) = enter_{TS}$, $\gamma(out_{TS}, in_{W_2}) = leave_{TS}$. The actions r_i, s_i and c_i describe respectively a receive, a send and a resulting communication at port i (see the numbers in figure 6.1). The other action names are self-explanatory.

$$\begin{aligned}
W_1 &= \Delta r_1 t\, out_{W_1} W_1 \\
W_2 &= \Delta r_2 t\, open \Delta in_{W_2} W_2 \\
WC &= s_3 t^2 s_4 t^4 WC \\
TS &= \Delta in_{TS} t^3 out_{TS} TS \\
D_1 &= \Delta r_3 \Gamma_0^2 s_1 D_1
\end{aligned}$$

$$D_2 = \Delta r_4 t\, s_2 D_2$$

The behaviour of the whole system is expected to be the following: The workcell controller sends a message to W_1. After receipt of this message a product is put on the conveyor belt. While it is being transported to W_2 the workcell controller sends a message to W_2 saying that it must open the gate. Then, the product arrives at W_2 and the whole process starts over again. We are especially interested in how long the gate is open before the arrival of a product. This can be studied by hiding all actions in S except *open* and $leave_{TS}$. Hence, we are interested in the behaviour of

$$\tau_I(S)$$

where $I = \{c_i | i = 1, 2, 3, 4\} \cup \{enter_{TS}\}$. We will only write down the result of the verification. The verification itself is straightforward (cf. [3, 15, 20]). $\tau_I(S)$ is a solution for U_1 in the following set of guarded equations.

$$\begin{aligned}
U_1 &= \tau \cdot (\tau \cdot U_3 + \tau \cdot U_4),\\
U_2 &= leave_{TS} \cdot U_1 + \tau \cdot leave_{TS} \cdot U_3 + \tau \cdot leave_{TS} \cdot U_4,\\
U_3 &= t \cdot (\tau t^3 \cdot open \cdot t^2 \cdot U_2 + \tau \cdot t^3 \cdot open \cdot t \cdot leave_{TS} \cdot t \cdot U_1),\\
U_4 &= t^4 \cdot open \cdot leave_{TS} \cdot t^2 \cdot U_1.
\end{aligned}$$

We see that the system does not contain any deadlocks or time-stops and the gate is open for at most two time units. But the behaviour of the system is more complicated than expected. Close inspection of the specification reveals that there is a good reason for this. Consider the case where delivery of a message in D_1 takes two time units. Then the system arrives at U_2 via U_3. In U_2 the situation is that a product is ready to enter W_2 and WC has just issued a new message via D_1 to W_1. Now there are three actions that can happen. The product can enter W_2 or D_1 can decide to deliver the message in 0 or in more than 0 time units. In the last case it is known that the next product will arrive after more than four time units (second option in U_2) or after exactly four time units (third option) when $leave_{TS}$ takes place. Therefore, U_2 cannot be identified with $leave_{TS} \cdot U_1$ and thus all equations are necessary. We need the axiom $abx + aby = a(bx + by)$ $(a, b \in A_{\delta\tau t})$ valid in ready trace semantics [14] to prove $\tau_I(S)$ equal to S_2, defined by the equation:

$$S_2 = \tau(t^4 open\, leave_{TS} t^2 + t^4 open\, t\, leave_{TS} t + t^4 open\, t^2 leave_{TS})\, S_2.$$

The ready trace axiom is not valid in weak bisimulation semantics which will be introduced in the next section, but it respects deadlock behaviour.

We can try to improve the performance of the workcell by letting the workcell controller issue its commands faster. Define a new workcell controller and a workcell S' by:

$$\begin{aligned}
S' &= \partial_H(W_1 \parallel W_2 \parallel WC' \parallel TS \parallel D_1 \parallel D_2),\\
WC' &= s_3 t^2 s_4 t^2 WC'.
\end{aligned}$$

H, γ are the same as above. We are now only interested in the time consistency of WC'. Therefore all actions are hidden. It turns out that $\tau_J(WC')$ ($J = I \cup \{leave_{TS}, open\}$) is the solution for X of the following equations (again the law $abx + aby = a(bx + by)$ is used).

$$\begin{aligned} X &= \tau \cdot t^4 \cdot (X + Y), \\ Y &= \tau \cdot (t \cdot \delta + t^4 \cdot X). \end{aligned}$$

A time-stop occurs in Y. This means that the time constraints of the components of S' were incompatible. And indeed, in S' it is possible that W_1 must put a product on the conveyor belt, while the transport system TS still needs one time unit before it is capable to accept this product from W_1. As in any implementation time cannot be blocked, the new workcell cannot be built.

6.5 An operational semantics for $\mathrm{ACP}^t_{\tau\epsilon}$

We give an operational semantics to $\mathrm{ACP}^t_{\tau\epsilon}$ which corresponds to a way $\mathrm{ACP}^t_{\tau\epsilon}$-terms can be executed. In this way we show directly how $\mathrm{ACP}^t_{\tau\epsilon}$-terms can be seen as processes. The axioms are complete with respect to the operational semantics. Thus, there is a direct correspondence between the two. This means that the axioms indeed capture the idea of processes.

The semantics of an $\mathrm{ACP}^t_{\tau\epsilon}$-term is given using a transition relation $\longrightarrow$ that defines the behaviour of terms. If process p can perform action a then this is denoted by a transition $p \xrightarrow{a} q$. The action a is called the *label* of the transition and the term q represents the resulting behaviour. The transition relation is defined by the *Transition System Specification* (TSS) [17] in table 6.4. The rules R1-R12.3 are inference rules. All transitions between closed $\mathrm{ACP}^t_{\tau\epsilon}$-terms that are derivable using these rules are in the transition relation $\longrightarrow$.

Labels in the transition relation are chosen from $Act_{t\tau\surd} = Act \cup \{t, \tau, \surd\}$. The label $\surd$ denotes termination, t represents the proceeding of a time unit and τ the occurrence of an internal action. In table 6.4 a and b range over $Act_{t\tau\surd}$ and c ranges over Act unless explicitly stated otherwise. H and I are sets of actions not containing $\surd, t, \tau$ and x, y, z are variables.

Only some rules are explained here. For the others we refer to [13, 17] in particular for rules 12.2 and 12.3. Rule 7.2 is introduced to allow $p \mid q$ to terminate if both p and q can terminate, and to make it possible that if τ's can occur as initial steps in both p and q, then $p \mid q$ can perform an initial τ-step. This is introduced for the proof of lemma 6.6.6. Note that this only influences the behaviour of the communication merge, which is just an auxiliary operator. Rule 5.4 indicates that t-actions in both sides of the parallel operator can communicate.

The behaviour of a term is now given by the transitions it can perform, whereby we consider strong bisimulation equivalent process terms as equal. Strong bisimulation equivalence is chosen as it is the coarsest relation that does not alter the operational behaviour of a transition system. Due to the rules 12.1-12.3 this captures exactly what is known as weak bisimulation.

R1: $\epsilon \xrightarrow{\surd} \delta$

R2: $a \xrightarrow{a} \epsilon$

R3.1: $\dfrac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$ R3.2: $\dfrac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$

R4.1: $\dfrac{x \xrightarrow{a} x'}{xy \xrightarrow{a} x'y}$ if $a \in Act_{t\tau}$ R4.2: $\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{a} y'}{xy \xrightarrow{a} y'}$

R5.1: $\dfrac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$ if $a \in Act_{\tau}$ R5.2: $\dfrac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$ if $a \in Act_{\tau}$

R5.3: $\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \parallel y \xrightarrow{c} x' \parallel y'}$ if $\gamma(a,b) = c$ R5.4: $\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x' \parallel y'}$ if $a = t, \surd$

R6: $\dfrac{x \xrightarrow{a} x'}{x \between y \xrightarrow{a} x' \parallel y}$ if $a \in Act_{\tau}$

R7.1: $\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{x \mid y \xrightarrow{c} x' \parallel y'}$ if $\gamma(a,b) = c$ R7.2: $\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{x \mid y \xrightarrow{a} x' \parallel y'}$ if $a = t, \tau, \surd$

R9: $\dfrac{x \xrightarrow{a} x'}{\partial_H(x) \xrightarrow{a} \partial_H(x')}$ if $a \notin H$

R10.1 $\dfrac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{a} \tau_I(x')}$ if $a \notin I$ R10.2: $\dfrac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')}$ if $a \in I$

R11: $\dfrac{\langle p_x | E\rangle \xrightarrow{a} y}{\langle x | E\rangle \xrightarrow{a} y}$ if $x = p_x \in E$

R12.1: $a \xrightarrow{a} \tau$ if $a \in Act_{t\tau}$ R12.2: $\dfrac{x \xrightarrow{\tau} y \quad y \xrightarrow{a} z}{x \xrightarrow{a} z}$

R12.3: $\dfrac{x \xrightarrow{a} y \quad y \xrightarrow{\tau} z}{x \xrightarrow{a} z}$

Table 6.4: The operational rules for $\mathrm{ACP}^t_{\tau\epsilon}$

Definition 6.5.1. A relation $R \subseteq$ between $\mathrm{ACP}^t_{\tau\epsilon}$-terms is called a *bisimulation relation* if it satisfies the *transfer property*, i.e.:

1. if $p\,R\,q$ and $p \xrightarrow{a} p'$ for $a \in Act_{t\tau\surd}$ then $\exists q'$ such that $q \xrightarrow{a} q'$ and $p'\,R\,q'$,
2. if $p\,R\,q$ and $q \xrightarrow{a} q'$ for $a \in Act_{t\tau\surd}$ then $\exists p'$ such that $p \xrightarrow{a} p'$ and $p'\,R\,q'$.

We say that two terms p and q are bisimilar, notation $p \leftrightarroweq q$, if there exists a bisimulation relation containing the pair (p, q).

The transition system specification in table 6.4 is in *tyft/tyxt*-format [17]. This immediately implies that $\leftrightarroweq$ is a congruence relation.

It is standard to prove soundness and completeness of the axioms. Therefore, we omit all the proofs (cf. [3] and section 5.10 where it is outlined how to prove soundness and completeness).

Lemma 6.5.2. *(Soundness of* $\mathrm{ACP}^t_{\tau\epsilon}$*) Let* p *and* q *be closed* $\mathrm{ACP}^t_{\tau\epsilon}$*-expressions.*

$$\mathrm{ACP}^t_{\tau\epsilon} + \mathrm{RSP} + \mathrm{REC} \vdash p = q \Rightarrow p \leftrightarroweq q.$$

Theorem 6.5.3. *(Completeness) Let* p *and* q *be recursion free closed* $\mathrm{ACP}^t_{\tau\epsilon}$*-expressions. It holds that:*

$$p \leftrightarroweq q \quad \Leftrightarrow \quad \mathrm{ACP}^t_{\tau\epsilon} \vdash p = q$$

6.6 Relating $\mathrm{ACP}_{\tau\epsilon}$ to $\mathrm{ACP}^t_{\tau\epsilon}$

Real time specifications contain more information than specifications in $\mathrm{ACP}_{\tau\epsilon}$. One is not always interested in the timing information and therefore, one sometimes likes to work in the setting of $\mathrm{ACP}_{\tau\epsilon}$. However, it is interesting to know what an $\mathrm{ACP}_{\tau\epsilon}$ process means in terms of the timing of $\mathrm{ACP}^t_{\tau\epsilon}$, for instance when specifications where time does not play a role are combined with specifications where time is important. Here we give a translation of $\mathrm{ACP}_{\tau\epsilon}$ processes into $\mathrm{ACP}^t_{\tau\epsilon}$, thereby fixing an intuition about time in $\mathrm{ACP}_{\tau\epsilon}$.

The sequential composition in $\mathrm{ACP}_{\tau\epsilon}$ describes that its left argument must happen before its right argument with an arbitrary delay in between. We use Δ to model this delay. Moreover, a general delay is put in front and after every process to indicate that it may start after an undefined amount of time and that time can proceed after the process has been terminated. E.g. a is translated to $\Delta a \Delta$ and $a \cdot b$ is translated to $\Delta a \Delta b \Delta$. Note that somehow this intuition conflicts with the intuition adopted when testing processes [11] where the delay between actions that are not blocked, is bounded.

For more complex processes the translation is slightly more complicated. The translation of $a + b$ is $\Delta(a\Delta + b\Delta)$ as the choice between a and b is externally determinable. In order to achieve this, we first define a function T that translates a non-timed process to a timed process without initial delay. The translation is

finished by putting an initial Δ in front of the translated process, e.g. p translates to $\Delta \cdot T(p)$.

The function T is defined by:

Definition 6.6.1. The *real time translation* T from $\mathrm{ACP}_{\tau\epsilon}$-expressions to closed $\mathrm{ACP}^t_{\tau\epsilon}$-expressions is defined by:

$$\begin{aligned}
&T(a) = a\Delta \ (a \in Act_\tau),\\
&T(\delta) = \delta,\\
&T(\epsilon) = \epsilon,\\
&T(p + q) = T(p) + T(q),\\
&T(p \cdot q) = T(p) \cdot T(q),\\
&T(p \parallel q) = T(p \between q + q \between p + p \mid q),\\
&T(p \between q) = T(p) \between \Delta \cdot T(q),\\
&T(p \mid q) = T(p) \mid T(q),\\
&T(\partial_H(p)) = \partial_H(T(p)),\\
&T(\tau_I(p)) = \tau_I(T(p)),\\
&T(\langle x|E\rangle) = \langle x|T(E)\rangle,\\
&T(x) = x.
\end{aligned}$$

For a guarded recursive specification E, $T(E)$ is defined as:

$$T(E) = \{x = T(p_x) | x = p_x \in E\}$$

The following lemmas are used to prove that identities derivable in $\mathrm{ACP}_{\tau\epsilon}$ are also valid and derivable after translation into $\mathrm{ACP}^t_{\tau\epsilon}$.

Definition 6.6.2. Let σ be a substitution, mapping variables to $\mathrm{ACP}^t_{\tau\epsilon}$-terms. We define the substitution σ_T by:

$$\sigma_T(x) = T(\sigma(x)).$$

Lemma 6.6.3. *Let p be an $\mathrm{ACP}^t_{\tau\epsilon}$-expression and let σ be a substitution. Then:*

$$T(\sigma(p)) = \sigma_T(T(p)).$$

Proof. Use induction on the structure of p. □

Lemma 6.6.4. *We have the following fact:*

$$\mathrm{ACP}^t_{\tau\epsilon} + \mathrm{REC} + \mathrm{RSP} \vdash \Delta \cdot T(p) \parallel \Delta \cdot T(q) = \Delta T(p \parallel q)$$

Proof. It is shown that both sides satisfy the equation $X = tX + T(p \parallel q)$.
$\Delta T(p) \parallel \Delta T(q) =$
$\Delta T(p) \between \Delta T(q) + \Delta T(q) \between \Delta T(p) + \Delta T(p) \mid \Delta T(q) =$
$T(p) \between \Delta T(q) + T(q) \between \Delta T(p) + t(\Delta T(p) \parallel \Delta T(q)) + T(p) \mid T(q) =$
$T(p \parallel q) + t(\Delta T(p) \parallel \Delta T(q))$

$\Delta T(p \parallel q) = t\Delta T(p \parallel q) + T(p \parallel q)$ □

Lemma 6.6.5. *Let* p, p' *be* $\mathrm{ACP}_{\tau\epsilon}$*-expressions. Then:*

$$\mathrm{ACP}_{\tau\epsilon} + \mathrm{REC} + \mathrm{RSP} \vdash p = p' \;\Rightarrow\; \mathrm{ACP}^t_{\tau\epsilon} + \mathrm{REC} + \mathrm{RSP} \vdash T(p) = T(p').$$

Proof. This lemma is proved with induction on the depth of the proof of $p = p'$. As a basic case $p = p'$ can be an instantiation of REC or an axiom $\mathrm{ACP}_{\tau\epsilon}$ or it can be an equational inference rule without premisses. Checking the axioms A1,..,A9 is completely trivial and therefore left out. Checking the reflexivity rule $(p = p)$ of equational reasoning is also straightforward.

T1 $T(a\tau) = T(a)T(\tau) = a\Delta\tau\Delta = a\tau\Delta = a\Delta = T(a)$,

T2 $T(\tau + \epsilon) = T(\tau) + T(\epsilon) = \tau\Delta + \epsilon = (\tau + \epsilon)(\Delta + \epsilon) + \epsilon =$
$\tau(\Delta + \epsilon) + \Delta + \epsilon + \epsilon = \tau\Delta = T(\tau)$,

T3 $T(a(\tau p + q)) = a\Delta(\tau\Delta T(p) + T(q)) =$
$a(t\Delta(\tau\Delta T(p) + T(q)) + \tau\Delta T(p) + T(q)) =$
$T(a(\tau p + q)) + a\Delta T(p) = T(a(\tau p + q) + ap)$,

EM1 trivial,

EM2 $T(\epsilon \between p) = T(\epsilon) \between \Delta T(p) = \epsilon \between \Delta T(p) = \delta = T(\delta)$,

EM3 $T(ap \between q) = a\Delta T(p) \between \Delta T(q) = a(\Delta T(p) \parallel \Delta T(q)) =$
$a(\Delta T(p) \parallel \Delta T(q)) = a\Delta T(p \parallel q) = T(a(p \parallel q))$,

EM4 $T((p + q) \between r) = (T(p) + T(q)) \between \Delta T(r) =$
$T(p) \between \Delta T(r) + T(q) \between \Delta T(r) = T(p \between r + q \between r)$,

EM5 trivial,

EM6 $T(\epsilon \mid \epsilon) = \epsilon \mid \epsilon = \epsilon = T(\epsilon)$,

EM7 $T(\epsilon \mid ap) = \epsilon \mid T(ap) = \delta = T(\delta)$,

EM8 $T(ap \mid bq) = a\Delta T(p) \mid b\Delta T(q) = (a \mid b)(\Delta T(p) \parallel \Delta T(q)) =$
$(a \mid b)\Delta T(p \parallel q) = T(\gamma(a, b)(p \parallel q))$ if $\gamma(a, b)$ defined. Otherwise, $\delta = T(\delta)$ results.,

EM9 trivial,

EM10 $T(\tau p \between q) = \tau\Delta T(p) \between \Delta T(q) = \tau(\Delta T(p) \parallel \Delta T(q)) = \tau\Delta T(p \parallel q) =$
$T(\tau(p \parallel q))$,

EM11 $T(\epsilon \mid \tau p) = \epsilon \mid \tau\Delta T(p) = \epsilon \mid (t\Delta T(p) + T(p)) = \epsilon \mid T(p) = T(\epsilon \mid p)$,

EM12 $T(\tau p \mid aq) = \tau\Delta T(p) \mid a\Delta T(q) = \Delta T(p) \mid a\Delta T(q) =$
$T(p) \mid T(aq) + tT(p) \mid a\Delta T(q) = T(p) \mid T(aq) = T(p \mid aq)$,

EM13 $T(\tau p \mid \tau q) = \tau\Delta T(p) \mid \tau\Delta T(q) = \tau(\Delta T(p) \mid \Delta T(q)) =$
$\tau\Delta T(p \parallel q) = T(\tau(p \parallel q))$.

Checking D1-D4,DT,DE,TE,TI1-TI5 goes in the same way.,

REC $T(\langle x|E\rangle) = \langle x|T(E)\rangle = \langle T(p_x)|T(E)\rangle = T(\langle p_x|E\rangle)$.

Now we check the inference rules. As they all go in the same way we only consider RSP and the congruence rule for the parallel operator.

Suppose RSP is the last inference rule used to conclude that $\sigma(x) = \langle x|E\rangle$. We must show that we can find a proof for $T(\sigma(x)) = T(\langle x|E\rangle)$, which is by definition equal to $\sigma_T(x) = \langle x|T(E)\rangle$. By induction there is a proof for $T(\sigma(E))$ which is, using lemma 6.6.3, the same as $\sigma_T(T(E))$. By applying RSP it follows that $\sigma_T(x) = \langle x|T(E)\rangle$, which is exactly what we had to prove.

Suppose the last (instantiated) inference rule used is

$$\frac{p_1 = q_1 \quad p_2 = q_2}{p_1 \parallel p_2 = q_1 \parallel q_2}.$$

By induction we have a proof for $T(p_1) = T(q_1)$ and $T(p_2) = T(q_2)$. From this we can derive $T(p_1) \between \Delta T(p_2) + T(p_2) \between \Delta T(p_1) + T(p_1) \mid T(p_2) = T(q_1) \between \Delta T(q_2) + T(q_2) \between \Delta T(q_1) + T(q_1) \mid T(q_2)$. This is exactly equal to $T(p_1 \parallel p_2) = T(q_1 \parallel q_2)$, which we had to prove. □

The following lemma says that the transition from ACP to ACP with time steps preserves derivable facts.

Theorem 6.6.6. *Let p, p' be $\mathrm{ACP}_{\tau\epsilon}$-expressions. Then:*

$$\mathrm{ACP}_{\tau\epsilon} + \mathrm{REC} + \mathrm{RSP} \vdash p = p' \;\Rightarrow\; \mathrm{ACP}^t_{\tau\epsilon} + \mathrm{REC} + \mathrm{RSP} \vdash \Delta \cdot T(p) = \Delta \cdot T(p').$$

Proof. Direct by lemma 6.6.5. □

We are now able to explain why theorem 6.6.6 requires an unusual approach to the axioms for the communication merge together with τ. The reason can be found in the T-translation of the term $(\tau a \mid \tau b)$ which is $(\tau\Delta a\Delta \mid \tau\Delta b\Delta)$. If we adopt the traditional law for τ and $\mid$, i.e. $\tau x \mid y = x \mid y$ [6], which holds when dropping τ in rule 7.2 in table 6.4, we would be able to prove $(\tau a \mid \tau b) = a \mid b$. But $T(a \mid b) \neq T(\tau a \mid \tau b)$, as the right hand side can do a t step while the left hand side cannot. So, without modification of the axioms for the communication merge, theorem 6.6.6 would not hold.

References

[1] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[2] J.C.M. Baeten and R.J. van Glabbeek. Abstraction and empty process in process algebra. *Fundamenta Informaticae*, XII:221–242, 1989.

[3] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[4] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

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

[6] J.A. Bergstra and J.W. Klop. Process algebra: specification and verification in bisimulation semantics. In M. Hazewinkel, J.K. Lenstra, and L.G.L.T. Meertens, editors, *Mathematics and Computer Science II*, CWI Monograph 4, pages 61–94. North-Holland, Amsterdam, 1986.

[7] J.A. Bergstra and J.W. Klop. A complete inference system for regular processes with silent moves. In F.R. Drake and J.K. Truss, editors, *Proceedings Logic Colloquium 1986*, pages 21–81, Hull, 1988. North-Holland. First appeared as: Report CS-R8420, CWI, Amsterdam, 1984.

[8] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Failure semantics with fair abstraction. Report CS-R8609, CWI, Amsterdam, 1986.

[9] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Failures without chaos: a new process semantics for fair abstraction. In M. Wirsing, editor, *Formal Description of Programming Concepts – III, Proceedings of the* 3^{th} *IFIP WG 2.2 working conference,* Ebberup 1986, pages 77–103, Amsterdam, 1987. North-Holland.

[10] F. Biemans and P. Blonk. On the formal specification and verification of CIM architectures using LOTOS. *Computers in Industry*, 7(6):491–504, 1986.

[11] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal, editor, *Proceedings of the Eighth International Conference on Protocol Specification, Testing and Verification*. North-Holland, 1987.

[12] R.T. Gerth and A. Boucher. A timed failures model for extended communicating processes. In Th. Ottmann, editor, *Proceedings* 14^{th} *ICALP,* Karlsruhe, volume 267 of *Lecture Notes in Computer Science*, pages 95–114. Springer-Verlag, 1987.

[13] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 336–347. Springer-Verlag, 1987.

[14] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[15] R.J. van Glabbeek and F.W. Vaandrager. Modular specifications in process algebra – with curious queues (extended abstract). In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications, Workshop Passau 1987*, volume 394 of *Lecture Notes in Computer Science*, pages 465–506. Springer-Verlag, 1989.

[16] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989.

[17] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings* 16^{th} *ICALP,* Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation.*

[18] M. Hennessy and T. Regan. A temporal process algebra. Report 2/90, Computer Science Department, University of Sussex, 1990.

[19] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79:210–256, 1988.

[20] S. Mauw. Process algebra as a tool for the specification and verification of CIM–architectures. In J.C.M. Baeten, editor, *Applications of process algebra*, Cambridge Tracts in Theoretical Computer Science 17, pages 53–80. Cambridge University Press, 1990.

[21] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[22] R. Milner. *Communication and concurrency.* Prentice Hall International, 1989.

[23] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 1990.

[24] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: An algebra for timed processes. Technical Report RT-C16, IMAG, Laboratoire de Génie informatique, Grenoble, 1990. Also appeared in M. Broy and C.B. Jones,

editors, *Proceedings IFIP Working Conference on Programming Concepts and Methods*, Sea of Gallilea, Israel. North-Holland, 1990.

[25] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and applications. Report RT-C26, IMAG, Laboratoire de Génie informatique, Grenoble, 1990.

[26] R. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[27] J.L. Richier, J. Sifakis, and J. Voiron. Une algèbre des processus temporisés, 1987. In A. Arnold, editor, *Actes du deuxième colloque* C^3, Angouléme, 1987.

[28] W.P. Weijland. *Synchrony and asynchrony in process algebra.* PhD thesis, University of Amsterdam, 1989.

[29] W. Yi. Real-time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR'90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer-Verlag, 1990.

7

The Syntax and Semantics of μCRL

(Jan Friso Groote & Alban Ponse)

A simple specification language based on CRL (*Common Representation Language*) and therefore called μCRL (*micro* CRL) is proposed. It has been developed to study processes with data. So the language contains only basic constructs with an easy semantics. To obtain executability, *effective* μCRL has been defined. In effective μCRL equivalence between closed *data-terms* is decidable and the operational behaviour is finitely branching and computable. This makes effective μCRL a good platform for tooling activities.

7.1 Introduction

In telecommunication applications the necessity of the use of formal methods has been observed several times. For that purpose several specification languages have been developed (SDL [6], LOTOS [15], PSF [18] and CRL [22]). These languages are designed to optimise their usability for specification purposes. This means that they contain non-fundamental constructs to enhance the expressibility of the language.

In this paper we define a language called μCRL (*micro* CRL, where CRL stands for Common Representation Language [22]) as it consists of the essence of CRL. It has been developed under the assumption that an extensive study of the basic constructs of specification languages will yield fundamental insights that are hard to obtain via the languages mentioned above. These insights may assist further development of these languages. So our language is indeed very small although its definition still requires quite some pages. But, as μCRL only contains core constructs, it may not be so well suited as an actual specification language.

We believe that an advantage of our 'simple' approach is that when in the future several constructs that are not included in the language will be well understood and will have a concise and natural semantics, we can add them to the language without a time and manpower consuming redesign of existing but not optimally devised features.

The language μCRL consists of data and processes. The data part contains equational specifications: one can declare sorts and functions working upon these

sorts, and describe the meaning of these functions by equational axioms. The process part contains processes described in the style of CCS [19], CSP [12] or ACP [2, 3], where the particular process syntax has been taken from ACP. It basically consists of a set of uninterpreted actions that may be parameterised by data. These actions can represent various kinds of activities, depending on the usage of the language. There are sequential, alternative and parallel composition operators. Furthermore, recursive processes are specified in a simple way.

An important feature is executability. To obtain this, we define *effective* μCRL. In effective μCRL it is required that the equations specifying data constitute a semi-complete term rewriting system. This implies that data equivalence is decidable. Moreover, the specification of recursive processes must be guarded and sums over data sorts must be finite. This guarantees that the operational behaviour of every effective μCRL specification is finitely branching and computable. We believe that effective μCRL is an excellent base for building tools.

7.2 The syntax of μCRL

In this section we present the syntax of μCRL. It contains two major components, namely data specified by a many sorted term rewriting system and processes which are based on process algebra [3]. The syntax is defined in the BNF formalism. Syntactical categories are written in italics and we use a '.' to end each BNF clause. In reasoning about the syntax of μCRL we use the symbol $\equiv$ to denote syntactic equivalence.

7.2.1 Names

We assume the existence of a set $\mathcal{N}$ of *names* that are used to denote sorts, variables, functions, processes and labels of actions. The *names* in $\mathcal{N}$ are sequences over an alphabet not containing

$$\bot, +, \|, \between, |, \lhd, \rhd, \cdot, \delta, \tau, \partial, \rho, \Sigma, \surd, \times, \rightarrow, :, =, (,), \{, \}, `, `, \text{ a space and a newline.}$$

The space and the newline serve as separators between names and are used to lay out specifications. The symbol $\bot$ is used in the description of the semantics and the other symbols have special functions. Moreover, $\mathcal{N}$ does not contain the reserved keywords **sort**, **proc**, **var**, **act**, **func**, **comm**, **rew** and **from**.

7.2.2 Lists

In the sequel *X-list*, *×-X-list*, and *space-X-list* for any syntactical category X are defined by the following BNF syntax:

$$\begin{aligned} \textit{X-list} &::= X \mid \textit{X-list}, X. \\ \textit{×-X-list} &::= X \mid \textit{×-X-list} \times X. \\ \textit{space-X-list} &::= X \mid \textit{space-X-list}\ X. \end{aligned}$$

Lists are often described by the (informal) use of dots, e.g. $b_1 \times ... \times b_m$ with $m \geq 1$ is a $\times$-*X-list* where $b_1, ..., b_m$ are expressions in the syntactical category X. Note that lists cannot be empty.

7.2.3 Sort specifications

A *sort-specification* consists of a list of *names* representing sorts, preceded by the keyword **sort**.

sort-specification ::= **sort** *space-name-list*.

7.2.4 Function specifications

A *function-specification* consists of a list of function declarations. A *function-declaration* consists of a *name-list* (the names play the role of constant and function names), the sorts of their parameters and their target sort:

function-specification ::= **func** *space-function-declaration-list*.
function-declaration ::= *name-list* : → *name*
| *name-list* : ×-*name-list* → *name*.

7.2.5 Rewrite specifications

A *rewrite-specification* is given by a many sorted term rewriting system. Its syntax is given by the following BNF grammar:

rewrite-specification ::= *variable-declaration-section*
rewrite-rules-section.

In a *variable-declaration-section* all variables that are used in a *rewrite-rules-section* must be declared. In such a declaration, it is also stated what the sort of a variable is. A variable declaration section may be empty.

variable-declaration-section ::= **var** *space-variable-declaration-list*
| .

In a *variable-declaration*, the *name-list* contains the declared variables and the *name* denotes their sort:

variable-declaration ::= *name-list* : *name*.

Data-terms are defined in the standard way. The *name* without brackets in the syntax represents a variable or a constant.

data-term ::= *name*
| *name*(*data-term-list*).

The equations in a *rewrite-rules-section* define the meaning of functions operating on data. The syntax of a *rewrite-rules-section* is given by:

$$\begin{array}{rcl} \textit{rewrite-rules-section} & ::= & \textbf{rew}\ \textit{space-rewrite-rule-list}. \\ \textit{rewrite-rule} & ::= & \textit{name} = \textit{data-term} \\ & | & \textit{name}(\textit{data-term-list}) = \textit{data-term}. \end{array}$$

7.2.6 Process expressions and process specifications

In this section we first define what *process-expressions* look like. Then we define how these expressions can be used to construct *process-specifications*.

Process-expressions are defined via the following syntax explicitly taking care of the precedence among operators:

$$\begin{array}{rcl} \textit{process-expression} & ::= & \textit{parallel-expression} \\ & | & \textit{parallel-expression} + \textit{process-expression}. \\ \\ \textit{parallel-expression} & ::= & \textit{merge-parallel-expression} \\ & | & \textit{comm-parallel-expression} \\ & | & \textit{cond-expression} \\ & | & \textit{cond-expression} \between \textit{cond-expression}. \\ \\ \textit{merge-parallel-expression} & ::= & \textit{cond-expression} \parallel \textit{merge-parallel-expression} \\ & | & \textit{cond-expression} \parallel \textit{cond-expression}. \\ \\ \textit{comm-parallel-expression} & ::= & \textit{cond-expression} \mid \textit{comm-parallel-expression} \\ & | & \textit{cond-expression} \mid \textit{cond-expression}. \\ \\ \textit{cond-expression} & ::= & \textit{dot-expression} \\ & | & \textit{dot-expression} \lhd \textit{data-term} \rhd \textit{dot-expression}. \\ \\ \textit{dot-expression} & ::= & \textit{basic-expression} \\ & | & \textit{basic-expression} \cdot \textit{dot-expression}. \\ \\ \textit{basic-expression} & ::= & \delta \\ & | & \tau \\ & | & \partial(\{\textit{name-list}\}, \textit{process-expression}) \\ & | & \tau(\{\textit{name-list}\}, \textit{process-expression}) \\ & | & \rho(\{\textit{renaming-declaration-list}\}, \textit{process-expression}) \end{array}$$

| Σ(*single-variable-declaration*, *process-expression*)
| *name*
| *name*(*data-term-list*)
| (*process-expression*).

The + is the alternative composition. A *process-expression* $p+q$ behaves exactly as the argument that performs the first step.

The merge or parallel composition operator ($\|$) interleaves the behaviour of both arguments except that some actions in the arguments may communicate, which means that they happen at exactly the same moment and result in a communication action. In a *communication-specification* it can be declared which actions may communicate. The left merge ($\lfloor\!\lfloor$) behaves exactly as the parallel operator, except that its first step must originate from its left argument only. The communication merge ($|$) also behaves as the parallel operator, but now the first action must be a communication between both components. The left merge and the communication merge are added to allow proof theoretic reasoning. It is not expected that they will be used in specifications. In the sequel the syntactical category *parallel-expression* also refers to *merge-parallel-expression* and *comm-parallel-expression*.

The *conditional* construct *dot-expression* $\triangleleft$ *data-term* $\triangleright$ *dot-expression* is an alternative way to write an **if - then - else**-expression and is introduced by Hoare cs. [13] (see also [1]). The *data-term* is supposed to be of the standard sort of the Booleans (**Bool**). The $\triangleleft$-part is executed if the *data-term* evaluates to true (T) and the $\triangleright$-part is executed if the *data-term* evaluates to false (F).

The sequential composition operator '$\cdot$' says that first its left hand side can perform actions, and if it terminates then the second argument continues.

The constant δ describes the process that cannot do anything, especially, it cannot terminate. For instance, the process $\delta \cdot p$ can never perform an action of p. We also expect that δ is not used in specifications, but in reasoning δ is very handy to indicate that at a certain place a deadlock occurs.

The constant τ represents some internal activity that cannot be observed by the environment. It is therefore called the internal action.

The encapsulation operator ∂ is used to prevent actions of which the *name* is mentioned in its first argument from happening. This enables one to force actions into a communication.

The hiding operator, also denoted by a τ, is used to rename actions of which the *name* is mentioned into an internal action.

The renaming operator ρ is more general. It renames the *names* of actions according to the scheme in its first argument. A *renaming-declaration* is given by the following syntax:

renaming-declaration ::= *name* $\rightarrow$ *name*.

The first mentioned *name* is renamed to the second one.

The sum operator is used to declare a variable of a specific sort for use in a *process-expression*. A *single-variable-declaration* is defined by:

$$\textit{single-variable-declaration} ::= \textit{name} : \textit{name}.$$

The scope of the variable is exactly the *process-expression* mentioned in the sum operator. The behaviour of this construct is a choice between the behaviours of *process-expression* in which each value of the sort of the variable has been substituted for the variable.

The constructs *name* and *name*(*data-term-list*) are either process instantiations or actions: *name* refers to a declared process (or to an action) and *data-term-list* contains the arguments of the process identifier (or the action).

The syntax of *process-expressions* says that $\cdot$ binds strongest, the conditional construct binds stronger than the parallel operators which in turn bind stronger than $+$.

A *process-specification* is a list of (parameterised) names, which are used as process identifiers, that are declared together with their bodies.

$$\begin{array}{rcl} \textit{process-specification} & ::= & \textbf{proc}\ \textit{space-process-declaration-list}. \\ \textit{process-declaration} & ::= & \textit{name} = \textit{process-expression} \\ & | & \textit{name}(\textit{single-variable-declaration-list}) = \\ & & \qquad \textit{process-expression}. \end{array}$$

7.2.7 Action specification

In an *action-specification* all actions that are used are declared. Actions may be parameterised by data, and in that case we must declare on which sorts an action depends. An *action-specification* has the following form:

$$\begin{array}{rcl} \textit{action-specification} & ::= & \textbf{act}\ \textit{space-action-declaration-list}. \\ \textit{action-declaration} & ::= & \textit{name} \\ & | & \textit{name-list} : \times\textit{-name-list}. \end{array}$$

7.2.8 Communication specification

A *communication-specification* prescribes how actions may communicate. It only describes communication on the level of *names* of actions, e.g. if it is specified that $in \mid out = com$ then each action $in(t_1, ..., t_k)$ can communicate with $out(t'_1, ..., t'_m)$ to $com(t_1, ..., t_k)$ provided $k = m$ and t_i and t'_i denote the same data-element for $i = 1, ..., k$.

$$\begin{array}{rcl} \textit{communication-specification} & ::= & \textbf{comm}\ \textit{space-communication-declaration-list}. \\ \textit{communication-declaration} & ::= & \textit{name} \mid \textit{name} = \textit{name}. \end{array}$$

In the last rule the $\mid$ is a language symbol and should not be confused with the | used in sets and the BNF-syntax.

7.2.9 Specifications

Specifications are entities in which data, processes, actions etc. can be declared. The syntax of a *specification* is:

$$
\begin{array}{lcl}
\textit{specification} & ::= & \textit{sort-specification} \\
& | & \textit{function-specification} \\
& | & \textit{rewrite-specification} \\
& | & \textit{action-specification} \\
& | & \textit{communication-specification} \\
& | & \textit{process-specification} \\
& | & \textit{specification specification}.
\end{array}
$$

7.2.10 The standard sort Bool

In every *specification* the following function and sort declarations must be included. The reason for this special treatment of the sort **Bool** is that we want to guarantee that true and false as booleans are different. This can only be achieved if the names for true, false and the sort of booleans are predetermined.

$$
\begin{array}{ll}
\textbf{sort} & \textbf{Bool} \\
\textbf{func} & T :\rightarrow \textbf{Bool} \\
 & F :\rightarrow \textbf{Bool}
\end{array}
$$

7.2.11 An example

As an example we give a *specification* of a data transfer process. *Data-elements* of sort D are transferred from *in* to *out*.

$$
\begin{array}{ll}
\textbf{sort} & \textbf{Bool} \\
\textbf{func} & T, F :\rightarrow \textbf{Bool} \\
\textbf{sort} & D \\
\textbf{func} & d1, d2, d3 :\rightarrow D \\
\textbf{act} & in, out : D \\
\textbf{proc} & TR = \sum(x : D, in(x) \cdot out(x) \cdot TR)
\end{array}
$$

7.2.12 The from construct

For a *process-expression* or a *data-term* t, we write t **from** E for a *specification* E where we mean the *process-expression* or *data-term* t as defined in E. Often, it is clear from the context to which *specification* E the item t belongs. In this case we generally write t without explicit reference to E.

7.3 Static semantics

Not every *specification* is necessarily correctly defined. It may be that objects are not declared, that they are declared at several places or are not used in a proper way. In this section we define under which circumstances a *specification* does not have these problems and hence has a correct *static semantics*. Furthermore, we define some functions that will be used in the definition of the semantics of μCRL.

7.3.1 The signature of a specification

The signature of a specification is an important ingredient in defining the static semantics. It consists of a five-tuple of which each component is a set containing all elements of a main syntactical category declared in a *specification* E.

Definition 7.3.1. Let E be a *specification*. The *signature* $Sig(E) = (Sort, Fun, Act, Comm, Proc)$ of E is defined as follows:

- If $E \equiv$ **sort** n_1 ... n_m with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\{n_1, ..., n_m\}, \emptyset, \emptyset, \emptyset, \emptyset)$.
- If $E \equiv$ **func** fd_1 ... fd_m with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, Fun, \emptyset, \emptyset, \emptyset)$, where

$$\begin{array}{rcl} Fun & \stackrel{\text{def}}{=} & \{n_{ij} :\rightarrow S_i \mid fd_i \equiv n_{i1}, ..., n_{il_i} :\rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\} \\ & \cup & \{n_{ij} : S_{i1} \times ... \times S_{ik_i} \rightarrow S_i \mid fd_i \equiv n_{i1}, ..., n_{il_i} : \\ & & \qquad S_{i1} \times ... \times S_{ik_i} \rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\}. \end{array}$$

- If E is a *rewrite-specification*, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.
- If $E \equiv$ **act** ad_1 ... ad_m with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, Act, \emptyset, \emptyset)$, where

$$\begin{array}{rcl} Act & \stackrel{\text{def}}{=} & \{n_i \mid ad_i \equiv n_i, 1 \leq i \leq m\} \\ & \cup & \{n_{ij} : S_{i1} \times ... \times S_{ik_i} \mid \\ & & \qquad ad_i \equiv n_{i1}, ..., n_{il_i} : S_{i1} \times ... \times S_{ik_i}, 1 \leq i \leq m, 1 \leq j \leq l_i\}. \end{array}$$

- If $E \equiv$ **comm** cd_1 ... cd_m with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \{cd_i \mid 1 \leq i \leq m\}, \emptyset)$.
- If $E \equiv$ **proc** pd_1 ... pd_m with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \{pd_i \mid 1 \leq i \leq m\})$.
- If $E \equiv E_1\ E_2$ with $Sig(E_i) = (Sort_i, Fun_i, Act_i, Comm_i, Proc_i)$ for $i = 1, 2$, then

$$\begin{array}{l} Sig(E) \stackrel{\text{def}}{=} (Sort_1 \cup Sort_2, Fun_1 \cup Fun_2, \\ \qquad\qquad Act_1 \cup Act_2, Comm_1 \cup Comm_2, Proc_1 \cup Proc_2). \end{array}$$

Definition 7.3.2. Let $Sig = (Sort, Fun, Act, Comm, Proc)$ be a signature. We write

$Sig.Sort$ for $Sort$,
$Sig.Fun$ for Fun,
$Sig.Act$ for Act,
$Sig.Comm$ for $Comm$,
$Sig.Proc$ for $Proc$.

7.3.2 Variables

Variables play an important role in specifications. The next definition says which *names* can play the role of a variable without confusion with defined constants. Moreover, variables must have an unambiguous and declared sort.

Definition 7.3.3. Let Sig be a signature. A set $\mathcal{V}$ containing elements $\langle x : S \rangle$ with x and S *names*, is called a *set of variables* over Sig iff for each $\langle x : S \rangle \in \mathcal{V}$:

- for each *name* S' and *process-expression* p it holds that $x :\rightarrow S' \notin Sig.Fun$, $x \notin Sig.Act$ and $x = p \notin Sig.Proc$,
- $S \in Sig.Sort$,
- for each *name* S' such that $S' \not\equiv S$ it holds that $\langle x : S' \rangle \notin \mathcal{V}$.

Definition 7.3.4. Let *var-dec* be a *variable-declaration-section*. The function *Vars* is defined by:

$$Vars(\textit{var-dec}) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \textit{var-dec} \text{ is empty,} \\ \{\langle x_{ij} : S_i \rangle \mid 1 \le i \le m, \\ \qquad\qquad 1 \le j \le l_i\} & \text{if for some } m \ge 1 \ \textit{var-dec} \equiv \\ & \textbf{var } x_{11}, ..., x_{1l_1} : \\ & \quad S_1 \ ... \ x_{m1}, ..., x_{ml_m} : S_m. \end{cases}$$

In the following definitions we give functions yielding the sort and the variables in a *data-term* t. If for some reason no answer can be obtained, for instance because an undeclared *name* appears in t, a $\perp$ results. Of course this only works properly if $\perp$ does not occur in *names*.

Definition 7.3.5. Let t be a *data-term* and Sig a signature. Let $\mathcal{V}$ be a set of variables over Sig. We define:

$$sort_{Sig,\mathcal{V}}(t) \stackrel{\text{def}}{=} \begin{cases} S & \text{if } t \equiv x \text{ and } \langle x : S \rangle \in \mathcal{V}, \\ S & \text{if } t \equiv n,\ n :\rightarrow S \in Sig.Fun \text{ and for no} \\ & \quad S' \not\equiv S \ n :\rightarrow S' \in Sig.Fun, \\ S & \text{if } t \equiv n(t_1, ..., t_m), \\ & \quad n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_m) \rightarrow S \in Sig.Fun \\ & \quad \text{and for no } S' \not\equiv S \\ & \quad n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_m) \rightarrow S' \in Sig.Fun, \\ \perp & \text{otherwise.} \end{cases}$$

Definition 7.3.6. Let Sig be a signature, $\mathcal{V}$ a set of variables over Sig and let t be a *data-term*.

$$Var_{Sig,\mathcal{V}}(t) \stackrel{\text{def}}{=} \begin{cases} \{\langle x : S\rangle\} & \text{if } t \equiv x \text{ and } \langle x : S\rangle \in \mathcal{V}, \\ \emptyset & \text{if } t \equiv n \text{ and } n :\rightarrow S \in Sig.Fun, \\ \bigcup_{1\leq i\leq m} Var_{Sig,\mathcal{V}}(t_i) & \text{if } t \equiv n(t_1, ..., t_m), \\ \{\bot\} & \text{otherwise.} \end{cases}$$

We call a *data-term* t *closed* w.r.t. a signature Sig and a set of variables $\mathcal{V}$ iff $Var_{Sig,\mathcal{V}}(t) = \emptyset$. Note that $Var_{Sig,\mathcal{V}}(t) \subseteq \mathcal{V} \cup \{\bot\}$ for any *data-term* t.

7.3.3 Static semantics

A *specification* must be internally consistent. This means that all objects that are used must be declared exactly once and are used such that the sorts are correct. It also means that action, process, constant and variable *names* cannot be confused. Furthermore, it means that communications are specified in a functional way and that it is guaranteed that the rewrite rules satisfy a usual condition that the variables that are used at the right hand side of a equality sign must also occur at the left hand side. Because all these properties can be statically decided, a *specification* that is internally consistent is called SSC (*Statically Semantically Correct*). For a better understanding of the next definition, it may be helpful to read definition 7.3.8 first.

Definition 7.3.7. Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. We define the predicate 'is SSC w.r.t. Sig' inductively over the syntax of a *specification*.

- A *specification* **sort** n_1 ... n_m with $m \geq 1$ is SSC w.r.t. Sig iff all *names* $n_1, ..., n_m$ are pairwise different.
- A *specification* **func** $n_{11}, ..., n_{1l_1} : S_{11} \times ... \times S_{1k_1} \rightarrow S_1$
  $\vdots$
  $n_{m1}, ..., n_{ml_m} : S_{m1} \times ... \times S_{mk_m} \rightarrow S_m$

  with $m \geq 1$, $l_i \geq 1$, $k_i \geq 0$ for $1 \leq i \leq m$ is SSC w.r.t. Sig iff
  - for all $1 \leq i \leq m$ the *names* $n_{i1}, ..., n_{il_i}$ are pairwise different,
  - for all $1 \leq i < j \leq m$ it holds that if $n_{ik} \equiv n_{jk'}$ for some $1 \leq k \leq l_i$ and $1 \leq k' \leq l_j$, then either $k_i \neq k_j$, or $S_{il} \not\equiv S_{jl}$ for some $1 \leq l \leq k_i$,
  - for all $1 \leq i \leq m$ and $1 \leq j \leq k_i$ it holds that $S_{ij} \in Sig.Sort$ and $S_i \in Sig.Sort$.
- A *specification* of the form: *var-dec*
  rew-rul

  where *var-dec* is a *variable-declaration-section* and *rew-rul* is a *rewrite-rules-section* is SSC w.r.t. Sig iff

- *var-dec* is SSC w.r.t. Sig,
- *rew-rul* is SSC w.r.t. Sig and $Vars(\textit{var-dec})$.

⋆ The empty *variable-declaration-section* is SSC w.r.t. Sig.

A *variable-declaration-section* $\textbf{var}\ n_{11}, ..., n_{1k_1} : S_1$
$\vdots$
$n_{m1}, ..., n_{mk_m} : S_m$

with $m \geq 1$, $k_i \geq 1$ for $1 \leq i \leq m$ is SSC w.r.t. Sig iff

- $n_{ij} \not\equiv n_{i'j'}$ whenever $i \neq i'$ or $j \neq j'$ for $1 \leq i \leq m$, $1 \leq i' \leq m$, $1 \leq j \leq k_i$ and $1 \leq j' \leq k_{i'}$,
- the set $Vars(\textbf{var}\ n_{11}, ..., n_{1k_1} : S_1\ ...\ n_{m1}, ..., n_{mk_m} : S_m)$ is a set of variables over Sig.

⋆ A *rewrite-rules-section* $\textbf{rew}\ rw_1\ ...\ rw_m$ with $m \geq 1$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- if $rw_i \equiv n = t$ for some $1 \leq i \leq m$, then
  - $n :\rightarrow sort_{Sig,\emptyset}(t) \in Sig.Fun$,
  - t is SSC w.r.t. Sig and $\emptyset$,
- if $rw_i \equiv n(t_1, ..., t_{k_i}) = t$ for some $1 \leq i \leq m$ and $k_i \geq 1$, then
  - $n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_{k_i}) \rightarrow sort_{Sig,\mathcal{V}}(t) \in Sig.Fun$,
  - t, t_j $(1 \leq j \leq k_i)$ are SSC w.r.t. Sig and $\mathcal{V}$,
  - $Var_{Sig,\mathcal{V}}(t) \subseteq \bigcup_{1 \leq j \leq k_i} Var_{Sig,\mathcal{V}}(t_j)$.

⋆ A *data-term* n with n a *name* is SSC w.r.t. Sig and $\mathcal{V}$ iff $\langle n : S \rangle \in \mathcal{V}$ for some S, or $n :\rightarrow sort_{Sig,\mathcal{V}}(n) \in Sig.Fun$.

A *data-term* $n(t_1, ..., t_m)$ $(m \geq 1)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff $n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_m) \rightarrow sort_{Sig,\mathcal{V}}(n(t_1, ..., t_m)) \in Sig.Fun$ and all t_i $(1 \leq i \leq m)$ are SSC w.r.t. Sig and $\mathcal{V}$.

• A *specification* $\textbf{act}\ ad_1\ ...\ ad_m$ with $m \geq 1$ is SSC w.r.t. Sig iff

- for all $1 \leq i \leq m$ the *action-declaration* ad_i is SSC w.r.t. Sig,
- for all $1 \leq i < j \leq m$ it holds that $Sig(\textbf{act}\ ad_i).Act \cap Sig(\textbf{act}\ ad_j).Act = \emptyset$.

⋆ An *action-declaration* n is SSC w.r.t. Sig iff for each *name* S' it holds that $n :\rightarrow S' \notin Sig.Fun$.

An *action-declaration* $n_1, ..., n_m : S_1 \times ... \times S_k$ with $k, m \geq 1$ is SSC w.r.t. Sig iff

- for all $1 \leq i < j \leq m$ it holds that $n_i \not\equiv n_j$,

- for all $1 \leq i \leq k$ it holds that $S_i \in Sig.Sort$,
- for all $1 \leq i \leq m$ and for each *name* S' it holds that $n_i : S_1 \times ... \times S_k \rightarrow S' \notin Sig.Fun$.

• A *specification* **comm** $n_{11} | n_{12} = n_{13}$ ... $n_{m1} | n_{m2} = n_{m3}$ with $m \geq 1$ is SSC w.r.t. Sig iff

- for each $1 \leq i < j \leq m$ it is not the case that $n_{i1} \equiv n_{j1}$ and $n_{i2} \equiv n_{j2}$, or $n_{i1} \equiv n_{j2}$ and $n_{i2} \equiv n_{j1}$,
- for each $1 \leq i \leq m$ either $n_{i1} \in Sig.Act$ or there is a $k \geq 1$ such that $n_{i1} : S_1 \times ... \times S_k \in Sig.Act$,
- for each $1 \leq i \leq m$, $k \geq 1$ and *names* $S_1, ..., S_k$ it holds that if $n_{i1} : S_1 \times ... \times S_k \in Sig.Act$ then $n_{i2} : S_1 \times ... \times S_k \in Sig.Act$ and $n_{i3} : S_1 \times ... \times S_k \in Sig.Act$,
- for each $1 \leq i \leq m$, $k \geq 1$ and *names* $S_1, ..., S_k$ it holds that if $n_{i2} : S_1 \times ... \times S_k \in Sig.Act$ then $n_{i1} : S_1 \times ... \times S_k \in Sig.Act$ and $n_{i3} : S_1 \times ... \times S_k \in Sig.Act$,
- for each $1 \leq i \leq m$ it holds that if $n_{i1} \in Sig.Act$ then $n_{i2} \in Sig.Act$ and $n_{i3} \in Sig.Act$,
- for each $1 \leq i \leq m$ it holds that if $n_{i2} \in Sig.Act$ then $n_{i1} \in Sig.Act$ and $n_{i3} \in Sig.Act$.

• A *specification* **proc** pd_1 ... pd_m with $m \geq 1$ is SSC w.r.t. Sig iff

- for each $1 \leq i < j \leq m$:
  - if $pd_i \equiv n_i = p_i$ and $pd_j \equiv n_j = p_j$ then $n_i \not\equiv n_j$,
  - if for some $k \geq 1$ it holds that $pd_i \equiv n_i(x_1 : S_1, ..., x_k : S_k) = p_i$ and $pd_j \equiv n_j(x'_1 : S_1, ..., x'_k : S_k) = p_j$ then $n_i \not\equiv n_j$,
  - for all *names* S' it holds that $n_i :\rightarrow S_i \notin Sig.Fun$,
- if $pd_i \equiv n_i = p_i$ $(1 \leq i \leq m)$, then $n_i \notin Sig.Act$ and p_i is SSC w.r.t. Sig and $\emptyset$,
- if $pd_i \equiv n_i(x_{i1} : S_{i1}, ..., x_{ik_i} : S_{ik_i}) = p_i$ $(1 \leq i \leq m)$, then
  - $n_i : S_{i1} \times ... \times S_{ik_i} \notin Sig.Act$,
  - for all *names* S' it holds that $n_i : S_{i1} \times ... \times S_{ik_i} \rightarrow S' \notin Sig.Fun$,
  - the *names* $x_{i1}, ..., x_{ik_i}$ are pairwise different and $\{\langle x_{ij} : S_{ij} \rangle \mid 1 \leq j \leq k_i\}$ is a set of variables over Sig,
  - p_i is SSC w.r.t. Sig and $\{\langle x_{ij} : S_{ij} \rangle \mid 1 \leq j \leq k_i\}$.

⋆ A *process-expression* $p_1 + p_2$, *parallel-expressions* $p_1 \parallel p_2$, $p_1 \lfloor\!\lfloor p_2$, $p_1 \mid p_2$, a *dot-expression* $p_1 \cdot p_2$ are SSC w.r.t. Sig and $\mathcal{V}$ iff

- p_1 is SSC w.r.t. Sig and $\mathcal{V}$,
- p_2 is SSC w.r.t. Sig and $\mathcal{V}$.

A *cond-expression* $p_1 \lhd t \rhd p_2$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- p_1 is SSC w.r.t. Sig and $\mathcal{V}$,
- p_2 is SSC w.r.t. Sig and $\mathcal{V}$,
- t is SSC w.r.t. Sig and $\mathcal{V}$ and $sort_{Sig,\mathcal{V}}(t) = \mathbf{Bool}$.

The *basic-expressions* δ and τ are SSC w.r.t. Sig and $\mathcal{V}$.

The *basic-expressions* $\partial(\{n_1, ..., n_m\}, p)$ and $\tau(\{n_1, ..., n_m\}, p)$ with $m \geq 1$ are SSC w.r.t. Sig and $\mathcal{V}$ iff

- for all $1 \leq i < j \leq m$ $n_i \not\equiv n_j$,
- for $1 \leq i \leq m$ either $n_i \in Sig.Act$ or $n_i : S_1 \times ... \times S_k \in Sig.Act$ for some $k \geq 1$ and *names* $S_1, ..., S_k$,
- p is SSC w.r.t. Sig and $\mathcal{V}$.

The *basic-expression* $\rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, p)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- for $1 \leq i \leq m$ either $n_i \in Sig.Act$ or $n_i : S_1 \times ... \times S_k \in Sig.Act$ for some $k \geq 1$ and *names* $S_1, ..., S_k$,
- for each $1 \leq i < j \leq m$ it holds that $n_i \not\equiv n_j$,
- for $1 \leq i \leq m$, $k \geq 1$ and *names* $S_1, .., S_k$ it holds that if $n_i : S_1 \times ... \times S_k \in Sig.Act$, then also $n'_i : S_1 \times ... \times S_k \in Sig.Act$,
- for $1 \leq i \leq m$ it holds that if $n_i \in Sig.Act$, then also $n'_i \in Sig.Act$,
- p is SSC w.r.t. Sig and $\mathcal{V}$.

A *basic-expression* $\Sigma(x : S, p)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- $\mathcal{V} \backslash \{\langle x : S' \rangle \mid S' \text{ a } name\} \cup \{\langle x : S \rangle\}$ is a set of variables over Sig,
- p is SSC w.r.t. Sig and $\mathcal{V} \backslash \{\langle x : S' \rangle \mid S' \text{ a } name\} \cup \{\langle x : S \rangle\}$.

A *basic-expression* n is SSC w.r.t. Sig and $\mathcal{V}$ iff $n = p \in Sig.Proc$ for some *process-expression* p or $n \in Sig.Act$.

A *basic-expression* $n(t_1, ..., t_m)$ with $m \geq 1$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- $n(x_1 : sort_{Sig,\mathcal{V}}(t_1), ..., x_m : sort_{Sig,\mathcal{V}}(t_m)) = p \in Sig.Proc$ for some *names* $x_1, ..., x_m$ and *process-expression* p, or
  $n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_m) \in Sig.Act$,
- for $1 \leq i \leq m$ the *data-term* t_i is SSC w.r.t. Sig and $\mathcal{V}$.

A *basic-expression* (p) is SSC w.r.t. Sig and $\mathcal{V}$ iff p is SSC w.r.t. Sig and $\mathcal{V}$.

- A *specification* $E_1\ E_2$ is SSC w.r.t. Sig iff

- E_1 and E_2 are SSC w.r.t. Sig,
- $Sig(E_1).Sort \cap Sig(E_2).Sort = \emptyset$,
- if $n : S_1 \times ... \times S_m \to S \in Sig(E_1).Fun$ for some $m \geq 0$ then $n : S_1 \times ... \times S_m \to S' \notin Sig(E_2).Fun$ for any *name* S',
- $Sig(E_1).Act \cap Sig(E_2).Act = \emptyset$,
- if $n_1 \mid n_2 = n_3 \in Sig(E_1).Comm$ then for any *names* n_3' and n_3'' $n_1 \mid n_2 = n_3' \notin Sig(E_2).Comm$ and $n_2 \mid n_1 = n_3'' \notin Sig(E_2).Comm$,
- if $pd_1 \in Sig(E_1).Proc$ and $pd_2 \in Sig(E_2).Proc$, then
  - if $pd_1 \equiv n_1 = p_1$ and $pd_2 \equiv n_2 = p_2$, then $n_1 \not\equiv n_2$,
  - if for some $m \geq 1$ $pd_1 \equiv n_1(x_1 : S_1, ..., x_m : S_m) = p_1$ and $pd_2 \equiv n_2(x_1' : S_1, ..., x_m' : S_m) = p_2$, then $n_1 \not\equiv n_2$.

Definition 7.3.8. Let E be a *specification*. We say that E is SSC iff E is SSC w.r.t. $Sig(E)$.

The following lemma is helpful in checking that the predicate 'is SSC' is correctly defined.

Lemma 7.3.9. *Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. Let t be a data-term that is SSC w.r.t. Sig and $\mathcal{V}$. Then $sort_{Sig,\mathcal{V}}(t) \neq \bot$ and $\bot \notin Var_{Sig,\mathcal{V}}(t)$.*

7.3.4 The communication function

The following definition helps us in guaranteeing that the communication function is commutative and associative. This implies that the merge is also commutative and associative which allows us to write parallel processes without brackets as is done in the syntax (cf. LOTOS [15] where this is not the case).

Definition 7.3.10. Let Sig be a signature. The set $Sig.Comm^*$ is defined by:

$$Sig.Comm^* \stackrel{\text{def}}{=} \{n_1 \mid n_2 = n_3,\ n_2 \mid n_1 = n_3 \quad | \quad n_1 \mid n_2 = n_3 \in Sig.Comm\}.$$

So, in $Sig.Comm^*$ communication is always commutative. We say that a *specification* E is *communication-associative* iff

$$n_1 \mid n_2 = n,\ n \mid n_3 = n' \in Sig(E).Comm^* \Rightarrow \exists n'' :\ n_2 \mid n_3 = n'', n_1 \mid n'' = n' \in Sig(E).Comm^*.$$

With the condition that E is SSC this exactly implies that communication is associative.

7.4 Well-formed μCRL specifications

We define what well-formed specifications are. We only provide well-formed *specifications* with a semantics. Well-formedness is a decidable property.

Definition 7.4.1. Let E be a *specification* that is SSC. We say that E has *no empty sorts* iff for all $S \in Sig(E).Sort$ there is a *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $sort_{Sig(E),\emptyset}(t) \equiv S$.

Definition 7.4.2. Let E be a *specification*. E is called *well-formed* iff

- E is SSC,
- E is communication-associative,
- E has no empty sorts,
- $\mathbf{Bool} \in Sig(E).Sort$,
- $T :\rightarrow \mathbf{Bool} \in Sig(E).Fun$ and
- $F :\rightarrow \mathbf{Bool} \in Sig(E).Fun$.

7.5 Algebraic semantics

In this section we present the semantics of well-formed μCRL specifications. Given a signature Sig we introduce the class of Sig-algebras. Then for a well-formed *specification* E with $Sig(E) = Sig$, we define the subclass of Sig-algebras that form a model for the data part of E and in which the terms T and F of sort **Bool** are interpreted different. Then given such a model, we give an operational semantics for *process-expressions* in E.

7.5.1 Algebras

First we adapt the standard definitions of algebras etc. to μCRL (see e.g. [8] for these definitions).

Definition 7.5.1. Let E be a well-formed *specification*. A $Sig(E)$-*algebra* $\boldsymbol{A}$ is a structure containing

- for each $S \in Sig(E).Sort$ a non-empty domain $D(\boldsymbol{A}, S)$,
- for each $n :\rightarrow S \in Sig(E).Fun$ a constant $C(\boldsymbol{A}, n) \in D(\boldsymbol{A}, S)$,
- for each $n : S_1 \times ... \times S_m \rightarrow S \in Sig(E).Fun$ a function $F(\boldsymbol{A}, n : S_1 \times ... \times S_m)$ from $D(\boldsymbol{A}, S_1) \times ... \times D(\boldsymbol{A}, S_m)$ to $D(\boldsymbol{A}, S)$.

For two elements $a_1 \in D(\boldsymbol{A}, S_1)$ and $a_2 \in D(\boldsymbol{A}, S_2)$, we write $a_1 = a_2$ iff $S_1 \equiv S_2$ and a_1 and a_2 represent exactly the same element.

Definition 7.5.2. Let E be a well-formed *specification* and let $\boldsymbol{A}$ be a $Sig(E)$-algebra. We define the interpretation $[\![\cdot]\!]_{\boldsymbol{A}}$ from *data-terms* that are SSC w.r.t. $Sig(E)$ and $\emptyset$ into the domains of $\boldsymbol{A}$ as follows:

- if $t \equiv n$, then $[\![t]\!]_{\boldsymbol{A}} \stackrel{\text{def}}{=} C(\boldsymbol{A}, n)$,
- if $t \equiv n(t_1, ..., t_m)$ for some $m \geq 1$, then $[\![t]\!]_{\boldsymbol{A}} \stackrel{\text{def}}{=} F(\boldsymbol{A}, n : sort_{Sig(E),\emptyset}(t_1) \times ... \times sort_{Sig(E),\emptyset}(t_m))([\![t_1]\!]_{\boldsymbol{A}}, ..., [\![t_m]\!]_{\boldsymbol{A}})$.

We say that a $Sig(E)$-algebra $\boldsymbol{A}$ is *minimal* iff for each $a \in D(\boldsymbol{A}, S)$ and $S \in Sig(E).Sort$, there is some *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $[\![t]\!]_{\boldsymbol{A}} = a$. For *data-terms* t_1, t_2 that are SSC w.r.t. $Sig(E)$ and $\emptyset$ we write $\boldsymbol{A} \models t_1 = t_2$ iff $[\![t_1]\!]_{\boldsymbol{A}} = [\![t_2]\!]_{\boldsymbol{A}}$.

Definition 7.5.3. Let E be a well-formed *specification* and let $\boldsymbol{A}$ be a minimal $Sig(E)$-algebra. A function r mapping pairs of a sort S and an element from $D(\boldsymbol{A}, S)$ to *data-terms* that are SSC w.r.t. to $Sig(E)$ and $\emptyset$ is called a *representation function* of E and $\boldsymbol{A}$ iff $\boldsymbol{A} \models t = r(sort_{Sig(E),\emptyset}(t), [\![t]\!]_{\boldsymbol{A}})$ for each *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$.

7.5.2 Substitutions

We define substitutions on *data-terms*. These substitutions are immediately extended to *process-expressions* because this is required for the definition of the operational semantics.

Definition 7.5.4. Let E be a well-formed *specification* and $\mathcal{V}$ a set of variables over $Sig(E)$. Let *Term* be the set of *data-terms* that are SSC w.r.t. $Sig(E)$ and $\mathcal{V}$. A *substitution* σ over $Sig(E)$ and $\mathcal{V}$ is a mapping

$$\sigma : \mathcal{V} \to Term$$

such that for each $\langle x : S \rangle \in \mathcal{V}$ it holds that $sort_{Sig(E),\mathcal{V}}(\sigma(\langle x : S \rangle)) = S$. Substitutions are extended to *data-terms* by:

$$\begin{aligned}
&\sigma(x) \stackrel{\text{def}}{=} \sigma(\langle x : S \rangle) \quad \text{if } \langle x : S \rangle \in \mathcal{V} \text{ for some } \textit{name } S,\\
&\sigma(n) \stackrel{\text{def}}{=} n \quad \text{if } n :\rightarrow S \in Sig(E).Fun,\\
&\sigma(n(t_1, ..., t_m)) \stackrel{\text{def}}{=} n(\sigma(t_1), ..., \sigma(t_m)).
\end{aligned}$$

Definition 7.5.5. Let E be a well-formed *specification* and $\mathcal{V}$ a set of variables over $Sig(E)$. Let σ be a substitution over $Sig(E)$ and $\mathcal{V}$. We extend σ to *process-expressions* that are SSC w.r.t. $Sig(E)$ and $\mathcal{V}$ as follows:

- If $p_1 \Box p_2$ is a *process-expression*, a *parallel-expression* or a *dot-expression* ($\Box \in \{+, \|, \between, |, \cdot\}$), then $\sigma(p_1 \Box p_2) \stackrel{\text{def}}{=} \sigma(p_1) \Box \sigma(p_2)$,
- $\sigma(p_1 \lhd t \rhd p_2) \stackrel{\text{def}}{=} \sigma(p_1) \lhd \sigma(t) \rhd \sigma(p_2)$ for a *cond-expression* $p_1 \lhd t \rhd p_2$,

- $\sigma(\delta) \stackrel{\text{def}}{=} \delta$ and $\sigma(\tau) \stackrel{\text{def}}{=} \tau$ for *basic-expressions* δ and τ,
- if $\Box(gl, p)$ is a *basic-expression* ($\Box \in \{\partial, \tau, \rho\}$), then
$$\sigma(\Box(gl, p)) \stackrel{\text{def}}{=} \Box(gl, \sigma(p)),$$
- $\sigma(\Sigma(x : S, p)) \stackrel{\text{def}}{=} \Sigma(x : S, \sigma'(p))$ where σ' is defined by
$$\sigma'(\langle x' : S' \rangle) \stackrel{\text{def}}{=} \begin{cases} \langle x : S \rangle & \text{if } x' \equiv x \\ \sigma(\langle x' : S' \rangle) & \text{otherwise,} \end{cases}$$
for a *basic-expression* $\Sigma(x : S, p)$,
- $\sigma(n(t_1, ..., t_m)) \stackrel{\text{def}}{=} n(\sigma(t_1), ..., \sigma(t_m))$ for a *basic-expression* $n(t_1, ..., t_m)$,
- $\sigma(n) \stackrel{\text{def}}{=} n$ for a *basic-expression* n,
- $\sigma((p)) \stackrel{\text{def}}{=} (\sigma(p))$ for a *basic-expression* (p).

The validity of the following lemma gives us confidence that substitutions are indeed correctly defined.

Lemma 7.5.6. *Let E be a well-formed specification and $\mathcal{V}$ a set of variables over $Sig(E)$. Let σ be a substitution over $Sig(E)$ and $\mathcal{V}$.*

- *For any data-term t that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$, $\sigma(t)$ is also a data-term that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$. Moreover,*
$$sort_{Sig(E),\mathcal{V}}(t) \equiv sort_{Sig(E),\mathcal{V}}(\sigma(t)).$$
- *For any process-expression p that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$, $\sigma(p)$ is a process-expression that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$.*

7.5.3 Boolean preserving models

A $Sig(E)$-algebra $\boldsymbol{A}$ is a model of a well-formed *specification* E iff the equations defining the data in E hold in $\boldsymbol{A}$. Moreover, we say that $\boldsymbol{A}$ is *boolean preserving* iff T and F of sort **Bool** represent exactly the two different elements of $D(\boldsymbol{A}, \mathbf{Bool})$. Note that there are specifications which have no boolean preserving models of E, for instance a specification containing the equation $T = F$. For μCRL we are only interested in the minimal $Sig(E)$-algebras that are boolean preserving.

First we define the function *rewrites* that extracts the rewrite clauses together with declared variables from a *specification*.

Definition 7.5.7. We define the function *rewrites* on a *specification* E inductively as follows:

- If $E \equiv$ *sort-spec* with *sort-spec* a *sort-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.

- If $E \equiv$ *func-spec* with *func-spec* a *function-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv V\ R$ with V a *variable-declaration-section* and R a *rewrite-rules-section* with $R \equiv \textbf{rew}\ rd_1\ ...\ rd_m$ for some $m \geq 1$, then

$$rewrites(E) \stackrel{\text{def}}{=} \{\langle \{rd_i \mid 1 \leq i \leq m\}, Vars(V)\rangle\}.$$

- If $E \equiv$ *act-spec* with *act-spec* an *action-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv$ *comm-spec* with *comm-spec* a *communication-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv$ *proc-spec* with *proc-spec* a *process-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv E_1\ E_2$ where E_1 and E_2 are *specifications*, then $rewrites(E) \stackrel{\text{def}}{=} rewrites(E_1) \cup rewrites(E_2)$.

Definition 7.5.8. Let E be a well-formed *specification*. A $Sig(E)$-algebra $\boldsymbol{A}$ is a *model* of E, notation $\boldsymbol{A} \models_D E$, iff whenever $t = t' \in R$ with $\langle R, \mathcal{V}\rangle \in rewrites(E)$, then for any substitution σ over $Sig(E)$ and $\mathcal{V}$ such that $Var_{Sig(E),\mathcal{V}}(\sigma(t)) = Var_{Sig(E),\mathcal{V}}(\sigma(t')) = \emptyset$ it holds that $\boldsymbol{A} \models \sigma(t) = \sigma(t')$.

We write $\boldsymbol{A} \models_D E$ with a subscript D because the model only concerns the data in E.

Definition 7.5.9. Let E be a well-formed *specification*. A $Sig(E)$-algebra $\boldsymbol{A}$ is called *boolean preserving* w.r.t. E iff

- it is not the case that $\boldsymbol{A} \models T = F$,
- $|D(\boldsymbol{A}, \textbf{Bool})| = 2$, i.e. T and F are exactly the two elements of sort **Bool**.

7.5.4 The process part

In this section we define for each *process-expression* p that is SSC w.r.t. $Sig(E)$ and $\emptyset$, and each minimal model $\boldsymbol{A}$ of E that preserves the booleans and where E is some well-formed *specification*, a meaning in terms of a referential transition system (cf. the operational semantics in [2, 21, 22]).

Definition 7.5.10. A transition system $\mathcal{A}$ is a quadruple $(S, L, \longrightarrow, s)$ where

- S is a set of *states*,
- L is a set of *labels*,

– $\longrightarrow \subseteq S \times L \times S$ is a *transition relation*,

– $s \in S$ is the *initial state*.

Elements $(s', l, s'') \in \longrightarrow$ are generally written as $s' \stackrel{l}{\longrightarrow} s''$.

Definition 7.5.11. Let E be a well-formed *specification*, $\boldsymbol{A}$ be a minimal model of E that is boolean preserving and r be a representation function of E and $\boldsymbol{A}$. Let p be a *process-expression* that is SSC w.r.t. $Sig(E)$ and $\emptyset$. The *meaning* of p **from** E in $\boldsymbol{A}$ with representation function r is the *referential* transition system $\mathcal{A}(\boldsymbol{A}, r, p \textbf{ from } E)$ defined by

$$(S, L, \longrightarrow, s)$$

where

– $S \stackrel{\text{def}}{=} \{q \mid$ where q is a *process-expression* that is SSC w.r.t. $Sig(E)$ and $\emptyset\} \cup \{\surd\}$,

– $L \stackrel{\text{def}}{=} \{n(t_1, ..., t_m) \mid m \geq 0, n \in Sig(E).Act$ and for $1 \leq i \leq m$ it holds that $t_i \equiv r(S_i, a)$ for some $a \in D(\boldsymbol{A}, S_i)$ where $S_i \equiv sort_{Sig(E),\emptyset}(t_i)\} \cup \{\tau, \surd\}$,

– $s \stackrel{\text{def}}{=} p$,

– $\longrightarrow$ is the transition relation that contains exactly all transitions provable using the rules below (see for provability e.g. [9]). Let p, p', q, q' range over the set $S \setminus \{\surd\}$, P is a *process-expression* that is SSC w.r.t. $Sig(E)$ and some set of variables over $Sig(E)$, l ranges over the set L of labels, n, n_1, n_2 are *names*, $m \geq 0$ and $t_1, ..., t_m, u_1, ..., u_m$ are *data-terms* (note that there is no rule for δ):

- $\surd \stackrel{\surd}{\longrightarrow} \delta$.
- $\tau \stackrel{\tau}{\longrightarrow} \surd$.
- $n \stackrel{n()}{\longrightarrow} \surd$ if $n \in Sig(E).Act$,
- $n(u_1, ..., u_m) \stackrel{n(t_1,...,t_m)}{\longrightarrow} \surd$ with $m \geq 1$ if
  - $n : sort_{Sig(E),\emptyset}(u_1) \times ... \times sort_{Sig(E),\emptyset}(u_m) \in Sig(E).Act$,
  - $t_i \equiv r(sort_{Sig(E),\emptyset}(u_i), [\![u_i]\!]_{\boldsymbol{A}})$.
- $\dfrac{p \stackrel{l}{\longrightarrow} p'}{n \stackrel{l}{\longrightarrow} p'}$ if $n = p \in Sig(E).Proc$,
- $\dfrac{p \stackrel{l}{\longrightarrow} \surd}{n \stackrel{l}{\longrightarrow} \surd}$ if $n = p \in Sig(E).Proc$,

- $\dfrac{\sigma(P) \xrightarrow{l} p'}{n(u_1,...,u_m) \xrightarrow{l} p'}$ with $m \geq 1$ if
  - $n(x_1 : S_1,...,x_m : S_m) = P \in Sig(E).Proc$ where $S_i = sort_{Sig(E),\emptyset}(u_i)$ for all $1 \leq i \leq m$,
  - there is a substitution σ over $Sig(E)$ and the set of variables $\{\langle x_1 : S_1\rangle,...,\langle x_m : S_m\rangle\}$ such that $\sigma(\langle x_i : S_i\rangle) \equiv u_i$ for $1 \leq i \leq m$ where $S_i = sort_{Sig(E),\emptyset}(u_i)$,
- $\dfrac{\sigma(P) \xrightarrow{l} \surd}{n(u_1,...,u_m) \xrightarrow{l} \surd}$ with $m \geq 1$ if
  - $n(x_1 : S_1,...,x_m : S_m) = P \in Sig(E).Proc$ where $S_i = sort_{Sig(E),\emptyset}(u_i)$ for all $1 \leq i \leq m$,
  - there is a substitution σ over $Sig(E)$ and the set of variables $\{\langle x_1 : S_1\rangle,...,\langle x_m : S_m\rangle\}$ such that $\sigma(\langle x_i : S_i\rangle) \equiv u_i$ for $1 \leq i \leq m$ where $S_i = sort_{Sig(E),\emptyset}(u_i)$.

- $\dfrac{p \xrightarrow{l} p'}{p + q \xrightarrow{l} p'}$,
- $\dfrac{p \xrightarrow{l} \surd}{p + q \xrightarrow{l} \surd}$,
- $\dfrac{q \xrightarrow{l} q'}{p + q \xrightarrow{l} q'}$,
- $\dfrac{q \xrightarrow{l} \surd}{p + q \xrightarrow{l} \surd}$.

- $\dfrac{p \xrightarrow{l} p'}{p \cdot q \xrightarrow{l} p' \cdot q}$,
- $\dfrac{p \xrightarrow{l} \surd}{p \cdot q \xrightarrow{l} q}$.

- $\dfrac{p \xrightarrow{l} p'}{p \lhd t \rhd q \xrightarrow{l} p'}$ if $\mathcal{A} \models t = T$,
- $\dfrac{p \xrightarrow{l} \surd}{p \lhd t \rhd q \xrightarrow{l} \surd}$ if $\mathcal{A} \models t = T$,
- $\dfrac{q \xrightarrow{l} q'}{p \lhd t \rhd q \xrightarrow{l} q'}$ if $\mathcal{A} \models t = F$,

- $\dfrac{q \overset{l}{\longrightarrow} \surd}{p \lhd t \rhd q \overset{l}{\longrightarrow} \surd}$ if $\boldsymbol{A} \models t = F$.

- $\dfrac{p \overset{l}{\longrightarrow} p'}{p \parallel q \overset{l}{\longrightarrow} p' \parallel q}$,
- $\dfrac{q \overset{l}{\longrightarrow} q'}{p \parallel q \overset{l}{\longrightarrow} p \parallel q'}$,
- $\dfrac{p \overset{l}{\longrightarrow} \surd}{p \parallel q \overset{l}{\longrightarrow} q}$,
- $\dfrac{q \overset{l}{\longrightarrow} \surd}{p \parallel q \overset{l}{\longrightarrow} p}$,
- $\dfrac{p \overset{n_1(t_1,\ldots,t_m)}{\longrightarrow} p' \quad q \overset{n_2(t_1,\ldots,t_m)}{\longrightarrow} q'}{p \parallel q \overset{n(t_1,\ldots,t_m)}{\longrightarrow} p' \parallel q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \overset{n_1(t_1,\ldots,t_m)}{\longrightarrow} \surd \quad q \overset{n_2(t_1,\ldots,t_m)}{\longrightarrow} q'}{p \parallel q \overset{n(t_1,\ldots,t_m)}{\longrightarrow} q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \overset{n_1(t_1,\ldots,t_m)}{\longrightarrow} p' \quad q \overset{n_2(t_1,\ldots,t_m)}{\longrightarrow} \surd}{p \parallel q \overset{n(t_1,\ldots,t_m)}{\longrightarrow} p'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \overset{n_1(t_1,\ldots,t_m)}{\longrightarrow} \surd \quad q \overset{n_2(t_1,\ldots,t_m)}{\longrightarrow} \surd}{p \parallel q \overset{n(t_1,\ldots,t_m)}{\longrightarrow} \surd}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$.

- $\dfrac{p \overset{l}{\longrightarrow} p'}{p \between q \overset{l}{\longrightarrow} p' \parallel q}$,
- $\dfrac{p \overset{l}{\longrightarrow} \surd}{p \between q \overset{l}{\longrightarrow} q}$.

- $\dfrac{p \overset{n_1(t_1,\ldots,t_m)}{\longrightarrow} p' \quad q \overset{n_2(t_1,\ldots,t_m)}{\longrightarrow} q'}{p \mid q \overset{n(t_1,\ldots,t_m)}{\longrightarrow} p' \parallel q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \overset{n_1(t_1,\ldots,t_m)}{\longrightarrow} \surd \quad q \overset{n_2(t_1,\ldots,t_m)}{\longrightarrow} q'}{p \mid q \overset{n(t_1,\ldots,t_m)}{\longrightarrow} q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,

- $\dfrac{p \overset{n_1(t_1,...,t_m)}{\longrightarrow} p' \quad q \overset{n_2(t_1,...,t_m)}{\longrightarrow} \surd}{p \mid q \overset{n(t_1,...,t_m)}{\longrightarrow} p'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \overset{n_1(t_1,...,t_m)}{\longrightarrow} \surd \quad q \overset{n_2(t_1,...,t_m)}{\longrightarrow} \surd}{p \mid q \overset{n(t_1,...,t_m)}{\longrightarrow} \surd}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$.

- $\dfrac{p \overset{l}{\longrightarrow} p'}{\tau(\{n_1,...,n_k\},p) \overset{l}{\longrightarrow} \tau(\{n_1,...,n_k\},p')}$
  if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \leq i \leq k$, or $l \equiv \tau$,
- $\dfrac{p \overset{l}{\longrightarrow} \surd}{\tau(\{n_1,...,n_k\},p) \overset{l}{\longrightarrow} \surd}$
  if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \leq i \leq k$, or $l \equiv \tau$,
- $\dfrac{p \overset{n(t_1,...,t_m)}{\longrightarrow} p'}{\tau(\{n_1,...,n_k\},p) \overset{\tau}{\longrightarrow} \tau(\{n_1,...,n_k\},p')}$ if $n \equiv n_i$ for some $1 \leq i \leq k$,
- $\dfrac{p \overset{n(t_1,...,t_m)}{\longrightarrow} \surd}{\tau(\{n_1,...,n_k\},p) \overset{\tau}{\longrightarrow} \surd}$ if $n \equiv n_i$ for some $1 \leq i \leq k$.

- $\dfrac{p \overset{l}{\longrightarrow} p'}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \overset{l}{\longrightarrow} \rho(\{n_1 \to n_1',...,n_k \to n_k'\},p')}$
  if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \leq i \leq k$, or $l \equiv \tau$,
- $\dfrac{p \overset{l}{\longrightarrow} \surd}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \overset{l}{\longrightarrow} \surd}$
  if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \leq i \leq k$, or $l \equiv \tau$,
- $\dfrac{p \overset{n(t_1,...,t_m)}{\longrightarrow} p'}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \overset{n'(t_1,...,t_m)}{\longrightarrow} \rho(\{n_1 \to n_1',...,n_k \to n_k'\},p')}$
  if $n \equiv n_i$ and $n' \equiv n_i'$ for some $1 \leq i \leq k$,
- $\dfrac{p \overset{n(t_1,...,t_m)}{\longrightarrow} \surd}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \overset{n'(t_1,...,t_m)}{\longrightarrow} \surd}$
  if $n \equiv n_i$ and $n' \equiv n_i'$ for some $1 \leq i \leq k$.

- $\dfrac{p \overset{l}{\longrightarrow} p'}{\partial(\{n_1,...,n_k\},p) \overset{l}{\longrightarrow} \partial(\{n_1,...,n_k\},p')}$

if $l \equiv n(t_1, ..., t_m)$ and $n \not\equiv n_i$ for all $1 \leq i \leq k$, or $l \equiv \tau$,

- $$\frac{p \xrightarrow{l} \surd}{\partial(\{n_1, ..., n_k\}, p) \xrightarrow{l} \surd}$$
if $l \equiv n(t_1, ..., t_m)$ and $n \not\equiv n_i$ for all $1 \leq i \leq k$, or $l \equiv \tau$.

- $$\frac{\sigma(P) \xrightarrow{l} p'}{\Sigma(x : S, P) \xrightarrow{l} p'}$$
where σ is a substitution over $Sig(E)$ and $\{\langle x : S\rangle\}$ such that $\sigma(\langle x : S\rangle) = t$ for some *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$,

- $$\frac{\sigma(P) \xrightarrow{l} \surd}{\Sigma(x : S, P) \xrightarrow{l} \surd}$$
where σ is a substitution over $Sig(E)$ and $\{\langle x : S\rangle\}$ such that $\sigma(\langle x : S\rangle) = t$ for some *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$.

According to the convention in 7.2.12 $\mathcal{A}(\boldsymbol{A}, r, p$ **from** $E)$ is often abbreviated as $\mathcal{A}(\boldsymbol{A}, r, p)$. Note that the rules are not in *tyft/tyxt*-format. This turned out to be technically convenient. Again, the following lemma serves as a justification for our definition.

Lemma 7.5.12. *Let E be a well-formed specification, $\boldsymbol{A}$ be a minimal model of E that is boolean preserving and r a representation function of E and $\boldsymbol{A}$. Consider a process-expression p that is SSC w.r.t. $Sig(E)$ and $\emptyset$ and let $(S, L, \longrightarrow, s) \stackrel{\text{def}}{=} \mathcal{A}(\boldsymbol{A}, r, p)$. If for some sequence of labels $l_1, ..., l_m$ it holds that $p \xrightarrow{l_1} \ ... \ \xrightarrow{l_m} p'$, then either $p' \equiv \surd$ or p' is SSC w.r.t. $Sig(E)$ and $\emptyset$.*

We feel that our operational semantics is somewhat ad hoc; we can easily provide an alternative that is also satisfactory in the sense that for each *process-expression* the generated transition system is strongly bisimilar with that generated by the rules above. Therefore, we generally consider transition systems modulo strong bisimulation equivalence. This means that the operational semantics for μCRL as given in this document has only a *referential* meaning, and any generated transition system is therefore called a *referential transition system.* A consequence of this view is that for the generation of transition systems for a μCRL-*process-expression* an operational semantics generating a smaller number of states can be used.

Definition 7.5.13. Let $\mathcal{A}_1 = (S_1, L_1, \longrightarrow_1, s_1)$ and $\mathcal{A}_2 = (S_2, L_2, \longrightarrow_2, s_2)$ be two transition systems. We say that $\mathcal{A}_1$ and $\mathcal{A}_2$ are bisimilar, notation $\mathcal{A}_1 \leftrightarrows \mathcal{A}_2$, iff there is a relation $R \subseteq S_1 \times S_2$ such that

- $(s_1, s_2) \in R$,
- for each pair $(t_1, t_2) \in R$:

– $t_1 \xrightarrow{a}_1 t_1' \Rightarrow \exists t_2'\ t_2 \xrightarrow{a}_2 t_2'$ and $(t_1', t_2') \in R$,
– $t_2 \xrightarrow{a}_2 t_2' \Rightarrow \exists t_1'\ t_1 \xrightarrow{a}_1 t_1'$ and $(t_1', t_2') \in R$.

Let E be a well-formed *specification*, $\boldsymbol{A}$ a minimal boolean preserving model of E, and r a representation function of E and $\boldsymbol{A}$. For two μCRL-*process-expressions* p and q that are SSC w.r.t. $Sig(E)$ and $\emptyset$, we write

$$p \textbf{ from } E \leftrightarrows_{\boldsymbol{A},r} q \textbf{ from } E$$

iff $\mathcal{A}(\boldsymbol{A}, r, p \textbf{ from } E) \leftrightarrows \mathcal{A}(\boldsymbol{A}, r, q \textbf{ from } E)$.

The following lemma allows us to write $\leftrightarrows_{\boldsymbol{A}}$ instead of $\leftrightarrows_{\boldsymbol{A},r}$. Moreover, it gives us a useful property of bisimulation, i.e. that it is a congruence for all process operators. Note that according to our own convention we do not explicitly say where p and q stem from as they can only come from E.

Lemma 7.5.14. *Let E be a specification, $\boldsymbol{A}$ a minimal, boolean preserving model of E and p, q process-expressions that are SSC w.r.t. E and $\emptyset$.*

- *If $p \leftrightarrows_{\boldsymbol{A},r} q$ for some representation function r of E and $\boldsymbol{A}$, then $p \leftrightarrows_{\boldsymbol{A},r'} q$ for each representation function r' of E and $\boldsymbol{A}$.*
- *For all representation functions of E and $\boldsymbol{A}$, $\leftrightarrows_{\boldsymbol{A},r}$ is a congruence for all μCRL operators working on process-expressions.*

7.6 Effective μCRL-specifications

In order to provide a process language with tools, such as for instance a simulator, it is very important that the language has a computable operational semantics, i.e. it is decidable what the next (finite number of) steps of a process are. This is not at all the case for μCRL. Due to the undecidability of data equivalence, the use of possibly unguarded recursion and infinite sums, the next step relation need not be enumerable. We deal with this situation by restricting μCRL to *effective* μCRL. In effective μCRL data equivalence is decidable, only finite sums are allowed and recursion must be guarded. For effective μCRL the next step relation is indeed decidable.

7.6.1 Semi complete rewriting systems

For the data we require that the rewriting system is semi-complete (= weakly terminating and confluent) [16]. This implies that data equivalence between closed terms is decidable. Moreover, this is (in some sense) not too restrictive: every data type for which data equivalence is decidable, can be specified by a complete (= strongly terminating and confluent) term rewriting system [5]. As a complete term rewriting system is also semi-complete, all decidable data types can be expressed in effective μCRL.

We first define all required rewrite relations.

Definition 7.6.1. Let E be a well-formed *specification*. We define the *elementary rewrite relation* $\longrightarrow_E^e$ by:

$$\begin{aligned}\longrightarrow_E^e \quad \stackrel{\text{def}}{=} \quad & \{\sigma(u) \longrightarrow \sigma(u') \mid \\ & u = u' \in R \text{ with } \langle R, \mathcal{V}\rangle \in rewrites(E), \\ & \sigma \text{ is a substitution over } Sig(E) \text{ and } \mathcal{V} \\ & \qquad\qquad \text{such that } Var_{Sig(E),\mathcal{V}}(\sigma(u)) = \emptyset\}.\end{aligned}$$

The one-step reduction relation $\longrightarrow_E$ is inductively defined by:

- $u \longrightarrow u' \in \longrightarrow_E$ if $u \longrightarrow u' \in \longrightarrow_E^e$.
- $n(t_1, ..., t_m) \longrightarrow n(t'_1, ..., t'_m) \in \longrightarrow_E$ if for some $1 \leq i \leq m$
  - $t_i \longrightarrow t'_i \in \longrightarrow_E$,
  - for $j \neq i$ it holds that $t_j \equiv t'_j$ and $n(t_1, ..., t_m)$ is SSC w.r.t. $Sig(E)$ and $\emptyset$.

The *reduction relation* $\twoheadrightarrow_E$ is the reflexive and transitive closure of $\longrightarrow_E$. We write $t \longrightarrow_E u$ and $t \twoheadrightarrow_E u$ for $t \longrightarrow u \in \longrightarrow_E$ and $t \twoheadrightarrow u \in \twoheadrightarrow_E$, respectively.

The following lemma is meant to reassure ourselves that the definitions of the rewrite relations are correct. Moreover, it gives a basic but useful property.

Lemma 7.6.2. *Let E be a well-formed specification. Let t be a data-term that is SSC w.r.t. $Sig(E)$ and $\emptyset$. If $t \twoheadrightarrow_E t'$, then t' is also SSC w.r.t. $Sig(E)$ and $\emptyset$.*

With these rewrite relations it is easy to define confluence and termination.

Definition 7.6.3. Let E be a well-formed *specification*. E is *data-confluent* iff for *data-terms* t, t' and t'' that are SSC w.r.t. $Sig(E)$ and $\emptyset$ it holds that:

$$\left.\begin{array}{l} t \twoheadrightarrow_E t' \\ t \twoheadrightarrow_E t'' \end{array}\right\} \text{ implies that there is a } \textit{data-term } t''' \text{ such that } \left\{\begin{array}{l} t' \twoheadrightarrow_E t''' \\ t'' \twoheadrightarrow_E t'''. \end{array}\right.$$

A *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ is a *normal form* if for no *data-term* u it holds that $t \longrightarrow_E u$. E is *data-terminating* if for each *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ there is some normal form t'' such that $t \twoheadrightarrow_E t''$. E is *data-semi-complete* if E is data-confluent and data-terminating.

The following lemma states that in μCRL we can find a unique normal form for each *data-term* that can be obtained from a well-formed *specification*.

Lemma 7.6.4. *Let E be a well-formed specification that is data-semi-complete. For any data-term t that is SSC with respect to $Sig(E)$ and $\emptyset$, there is a unique data-term $N_E(t)$ satisfying*

$$t \twoheadrightarrow_E N_E(t) \text{ and } N_E(t) \text{ is a normal form.}$$

$N_E(t)$ is called the normal form of t and there is an algorithm to find $N_E(t)$ for each data-term t that is SSC w.r.t. $Sig(E)$ and $\emptyset$.

Effective μCRL is based on the following algebra of normal forms.

Definition 7.6.5. Let E be a well-formed data-semi-complete *specification*. The $Sig(E)$-algebra $\boldsymbol{A}_{N_E}$ of normal forms is defined by:

- for each name $S \in Sig(E).Sort$ there is a domain $D(\boldsymbol{A}_{N_E}, S) \stackrel{\text{def}}{=} \{N_E(t) \mid sort_{Sig(E),\emptyset}(t) = S$ and t is a *data-term* that is SSC w.r.t. $Sig(E)$ and $\emptyset\}$,
- $C(\boldsymbol{A}_{N_E}, n) \stackrel{\text{def}}{=} N_E(n)$ provided $n :\rightarrow S \in Sig(E).Fun$,
- $F(\boldsymbol{A}_{N_E}, n : S_1 \times ... \times S_m) = f$ where the function f is defined by:

  $$f(t_1, ..., t_m) = N_E(n(t_1, ..., t_m))$$

  with $t_i \in D(\boldsymbol{A}_{N_E}, S_i)$ for $1 \leq i \leq m$ provided $n : S_1 \times ... \times S_m \rightarrow S \in Sig(E).Fun$.

Note that in $\boldsymbol{A}_{N_E}$ it is easy to determine that $T \neq F$. It is however undecidable that the sort **Bool** has at most two elements. We must use the *finite sort tool* of section 7.6.5 to determine this. Often the algebra $\boldsymbol{A}_{N_E}$ is called the *canonical term algebra* of E.

7.6.2 Finite sums

If a μCRL specification contains infinite sums, then the operational behaviour is not finitely branching anymore. Consider for instance the behaviour of the following process:

X	**from**	**sort**	**Bool**
		func	$T, F :\rightarrow$ **Bool**
		sort	Nat
		func	$0 : Nat$
			$succ : Nat \rightarrow Nat$
		act	$a : Nat$
		proc	$X = \sum(x : Nat, a(x))$

The process X can perform an $a(m)$ step for each natural number m. We judge an infinitely branching operational behaviour undesirable and therefore exclude sums over infinite sorts from effective μCRL.

Definition 7.6.6. Let E be a well-formed *specification* and let $\boldsymbol{A}$ be a model of E. We say that E has *finite sums* w.r.t. $\boldsymbol{A}$ iff for each occurrence $\Sigma(x : S, p)$ in E the set $D(\boldsymbol{A}, S)$ is finite.

7.6.3 Guarded recursive specifications

Also unguarded recursion may lead to an infinitely branching operational behaviour. Consider for instance the following example:

$$\begin{array}{lll} X \ \textbf{from} & \textbf{sort} & \textbf{Bool} \\ & \textbf{func} & T, F :\rightarrow \textbf{Bool} \\ & \textbf{act} & a \\ & \textbf{proc} & X = X \cdot a + a \end{array}$$

The *process-expression* $X \cdot a$ can perform an a step to any *process-expression* a^m ($m \geq 1$) where a^m is the sequential composition of m a's. Therefore, we also exclude unguarded recursion from effective μCRL.

In the next definition it is said what a guarded μCRL specification is in very general terms.

Definition 7.6.7. Let E be a well-formed *specification* and $\boldsymbol{A}$ be a model of E that is boolean preserving. Let p be a *process-expression* of the form n or $n(t_1, ..., t_m)$ for some *name* n that is SSC w.r.t. $Sig(E)$ and $\emptyset$. Let q be a *process-expression* that is SSC w.r.t. $Sig(E)$ and $\emptyset$. We say that p is *guarded* w.r.t. $\boldsymbol{A}$ in q iff

- $q \equiv q_1 + q_2$, $q \equiv q_1 \parallel q_2$ or $q \equiv q_1 \mid q_2$, and p is guarded w.r.t. $\boldsymbol{A}$ in q_1 and q_2,
- $q \equiv q_1 \lhd c \rhd q_2$ and either $\boldsymbol{A} \models c = T$ and p is guarded w.r.t. $\boldsymbol{A}$ in q_1, or $\boldsymbol{A} \models c = F$ and p is guarded w.r.t. $\boldsymbol{A}$ in q_2,
- $q \equiv q_1 \cdot q_2$, $q \equiv q_1 \between q_2$, $q \equiv \partial(\{n_1, ..., n_m\}, q_1)$, $q \equiv \tau(\{n_1, ..., n_m\}, q_1)$, $q \equiv \rho(\{n_1 \rightarrow n'_1, ..., n_m \rightarrow n'_m\}, q_1)$ or $q \equiv (q_1)$ and p is guarded w.r.t. $\boldsymbol{A}$ in q_1,
- $q \equiv \Sigma(x : S, q_1)$ and p is guarded w.r.t. $\boldsymbol{A}$ in $\sigma(q_1)$ for any substitution σ over $Sig(E)$ and $\{\langle x : S \rangle\}$,
- $q \equiv \tau$ or $q \equiv \delta$,
- $q \equiv n'$ for a *name* n' and $p \not\equiv n'$ or
- $q \equiv n'(u_1, ..., u_{m'})$ for a *basic-expression* $n'(u_1, ..., u_{m'})$ and $n \not\equiv n'$, $m \neq m'$ or $[\![u_i]\!]_{\boldsymbol{A}} \neq [\![t_i]\!]_{\boldsymbol{A}}$ for some $1 \leq i \leq m$.

If p is not guarded w.r.t. $\boldsymbol{A}$ in q we say that p appears *unguarded* w.r.t. $\boldsymbol{A}$ in q.

Definition 7.6.8. Let E be a well-formed *specification* and $\boldsymbol{A}$ be a model of E that is boolean preserving. The *Process Name Dependency Graph* of E and $\boldsymbol{A}$, notation $PNDG(E, \boldsymbol{A})$, is constructed as follows:

- for each $n = p \in Sig(E).Proc$, n is a node of $PNDG(E, \boldsymbol{A})$,

- for each $n(x_1 : S_1, ..., x_m : S_m) = p \in Sig(E).Proc$ and *data-terms* $t_1, ..., t_m$ that are SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $sort_{Sig(E),\emptyset}(t_i) = S_i$ $(1 \leq i \leq m)$, $n(t_1, ..., t_m)$ is a node of $PNDG(E, \boldsymbol{A})$,

- if n is a node of $PNDG(E, \boldsymbol{A})$ and $n = p \in Sig(E).Proc$, then there is an edge

$$n \longrightarrow q$$

for a node $q \in PNDG(E, \boldsymbol{A})$ iff q is unguarded w.r.t. $\boldsymbol{A}$ in p,

- if $n(x_1 : sort_{Sig(E),\emptyset}(t_1), ..., x_m : sort_{Sig(E),\emptyset}(t_m)) = p \in Sig(E).Proc$ and $n(t_1, ..., t_m)$ is a node of $PNDG(E, \boldsymbol{A})$, then there is an edge

$$n(t_1, ..., t_m) \longrightarrow q$$

for a node $q \in PNDG(E, \boldsymbol{A})$ iff q is unguarded w.r.t. $\boldsymbol{A}$ in $\sigma(p)$ where σ is the substitution over $Sig(E)$ and $\{\langle x_i : sort_{Sig(E),\emptyset}(t_i)\rangle \mid 1 \leq i \leq m\}$ defined by

$$\sigma(\langle x_i : sort_{Sig(E),\emptyset}(t_i)\rangle) = t_i.$$

Definition 7.6.9. Let E be a well-formed *specification* and $\boldsymbol{A}$ be a model of E that is boolean preserving. We say that E is *guarded* w.r.t. $\boldsymbol{A}$ iff $PNDG(E, \boldsymbol{A})$ is well founded, i.e. does not contain an infinite path.

7.6.4 Effective μCRL-specifications

Here we define the operational semantics of effective μCRL by combining all definitions given above.

Definition 7.6.10. Let E be a *specification*. We call E an *effective* μCRL *specification* or for short an *effective specification* iff

- E is well-formed,
- E is data-semi-complete,
- E has finite sums w.r.t. $\boldsymbol{A}_{N_E}$,
- E is guarded w.r.t. $\boldsymbol{A}_{N_E}$.

Definition 7.6.11. Let E be an effective μCRL *specification*. Let p be a *process-expression* that is SSC w.r.t. $Sig(E)$ and $\emptyset$. The behaviour of p is the transition system

$$\mathcal{A}(\boldsymbol{A}_{N_E}, r, p \textbf{ from } E)$$

where the representation function r of E and $\boldsymbol{A}_{N_E}$ is the identity.

In effective μCRL data equivalence is indeed decidable and the operational behaviour is finitely branching and computable:

Theorem 7.6.12. *Let E be an effective μCRL specification and let $(S, L, \longrightarrow, s) = \mathcal{A}(\boldsymbol{A}_{N_E}, r, p)$ for some data-term p that is SSC w.r.t. $Sig(E)$ and $\emptyset$ and let r be the identity. Then*

- *for each pair of data-terms t_1, t_2 that are SSC w.r.t. $Sig(E)$ and $\emptyset$:*

$$t_1 =_E t_2 \text{ is decidable,}$$

- *for each process-expression p' that is SSC w.r.t. $Sig(E)$ and $\emptyset$:*

$$\{\langle a, p'' \rangle \mid p' \xrightarrow{a} p''\}$$

is finite and effectively computable. Moreover, its cardinality is also effectively computable from E and p.

The second point of the previous theorem says that $\mathcal{A}(\boldsymbol{A}_{N_E}, r, p \textbf{ from } E)$ is a *computable* transition system. In a recursion theoretic setting a *computable* transition system is defined as follows: let $\mathcal{A} = (S, L, \longrightarrow, s_0)$ be a transition system with S and L sets of natural numbers and $s_0 \in S$ is represented by 0. We say that $\mathcal{A}$ is a *computable* transition system iff $\longrightarrow$ is represented by a *total* recursive function ϕ that maps each number in S to (a coding of) a finite set of pairs $\{\langle l, s' \rangle \mid s \xrightarrow{l} s'\}$.

7.6.5 Proving μCRL-specifications effective

In general it is not decidable whether a μCRL *specification* is effective. But there are many tools available that can prove the effectiveness for quite large classes of *specifications*. These tools provide, given a specification, a 'yes' or a 'don't know' answer.

Definition 7.6.13. Let $\mathcal{E}$ be the set of all well-formed *specifications*. A *data-semi-completeness tool*, notation DC, a *finite-sort tool*, notation FS, and a *guardedness tool*, notation GD, are all decidable predicates over $\mathcal{E}$, i.e. $DC \subseteq \mathcal{E}, FS \subseteq \mathcal{N} \times \mathcal{E}, GD \subseteq \mathcal{E}$.

A tool is called *sound* if each claim of a certain property it makes about a well-formed *specification* is correct. In the definition of a sound finite-sort tool and a sound guardedness tool we assume that specifications are data-semi-complete because we expect that this is a minimal requirement for these tools to operate.

Definition 7.6.14. A data-semi-completeness tool DC is called *sound* iff for each *specification* E that is well-formed:

if $DC(E)$ holds, then E is data-semi-complete.

A finite-sort tool FS is called *sound* iff for each *name* n and *specification* E that is well-formed and data-semi-complete:

if $FS(n, E)$ holds, then $n \in Sig(E).Sort$ and $D(\mathbb{A}_{N_E}, n)$ is a finite set.

A guardedness tool GD is called *sound* iff for each *specification* E that is well-formed and data-semi-complete:

if $GD(E)$ holds, then E is guarded w.r.t. $\mathbb{A}_{N_E}$.

Sometimes a tool needs auxiliary information per *specification* to perform its task. In this case such a tool may work on a tuple containing a specification and a finite amount of such information. There is no prescribed format for this information, and it may vary from tool to tool. If a tool requires auxiliary information, then the soundness of the tool may not depend on this information. In this case the definition of soundness is modified as follows (the definition is only given for DC, the other cases can be defined likewise):

Definition 7.6.15. A data-semi-completeness tool DC requiring auxiliary information, is called *sound* iff for each well-formed *specification* E and each instance of auxiliary information $\mathcal{I}$:

if $DC(E, \mathcal{I})$ holds, then E is data-semi-complete.

This definition guarantees that even with incorrect auxiliary information DC always produces correct answers. DC has to be *robust*.

Below we describe some techniques for constructing sound tools, except in those cases where techniques are provided in the literature. As time proceeds, more and more powerful techniques will appear. In order to incorporate these technological advancements in μCRL, the techniques mentioned here are only possible candidates for sound tools. They may be replaced by others, as long as these also lead to sound tools.

There are many techniques for proving termination and confluence (see Huet and Oppen [14] and Dershowitz [7] for termination, Newman [20] for confluence if termination has been shown and Klop [16] for an overview). Therefore we will not go into details here.

The problem whether a sort has a finite number of elements [4] is undecidable and as far as we know no general techniques have been developed to prove that a sort has only a finite number of elements in a minimal algebra.

We present a possible approach that can only be applied to a restricted case: let E be a *specification* in $\mathcal{E}$ such that $DC(E)$ for some sound data-semi-completeness tool DC and assume that we are interested in the finiteness of sorts $S_1, ..., S_k$ occurring in E. Let F be the set of all functions specified in E that have as target sort one of the sorts S_i $(1 \leq i \leq k)$. We assume that their parameter sorts also originate from $S_1, ..., S_k$. As auxiliary information we use finite sets $\mathcal{I}_i$ of (closed) *data-terms* that ought to represent all elements of sort S_i.

We compute for each function $f \in F$ (with target sort S_j) and for all arguments in the sets $\mathcal{I}_i$ of appropriate sorts, whether application of f leads to a *data-term* equivalent to one of the elements of $\mathcal{I}_j$. This can be done as we assume that

$DC(E)$ holds. If this is successful, then obviously the sorts $S_1, ..., S_k$ have a finite number of elements.

Also the question whether a *specification* is guarded is undecidable. Still very good results can be obtained when guardedness is checked abstracting from the data parameters of process names. This is done by the following function HV. Its first argument contains the *process-expression* that is being searched for unguarded occurrences of *names* of processes and its second argument guarantees that the bodies of *process-declarations* are not searched twice.

Definition 7.6.16. Let E be a well-formed *specification* and let $\mathcal{V}$ be a set of variables over $Sig(E)$. A *process-type* is an expression $\langle n : S_1 \times ... \times S_m \rangle$ for some $m \geq 0$ with n a *name* and $S_1, ..., S_m$ *names*. The function HV maps pairs of a *process-expression* and a set of *process-types* to sets of *process-types*.

- $HV(\delta, PT) \stackrel{\text{def}}{=} \emptyset$.
- $HV(p_1 + p_2, PT) = HV(p_1 \lhd c \rhd p_2, PT) = HV(p_1 \parallel p_2, PT) = HV(p_1 \mid p_2, PT) \stackrel{\text{def}}{=} HV(p_1, PT) \cup HV(p_2, PT)$.
- $HV(p_1 \cdot p_2, PT) = HV(p_1 \between p_2, PT) = HV(\partial(\{n_1, ..., n_m\}, p_1), PT) = HV(\tau(\{n_1, ..., n_m\}, p_1), PT) = HV(\rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, p_1), PT) =$

  $HV(\Sigma(x : S, p_1), PT) \stackrel{\text{def}}{=} HV(p_1, PT)$.
- $HV(n(t_1, ..., t_m), PT) \stackrel{\text{def}}{=}$
  - $\{\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle\}$ if $\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle \in PT$.
  - $HV(p, PT \cup \{\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle\}) \cup \{\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle\}$ if $\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle \notin PT$ and $n(x_1 : sort_{Sig(E),\mathcal{V}}(t_1), ..., x_m : sort_{Sig(E),\mathcal{V}}(t_m)) = p \in Sig(E).Proc$ for some *names* $x_1, ..., x_m$.
- $HV(n, PT) \stackrel{\text{def}}{=}$
  - $\{\langle n :\rangle\}$ if $\langle n :\rangle \in PT$,
  - $HV(p, PT \cup \{\langle n :\rangle\}) \cup \{\langle n :\rangle\}$ if $\langle n :\rangle \notin PT$ and $n = p \in Sig(E).Proc$.
- $HV((p), PT) \stackrel{\text{def}}{=} HV(p, PT)$.

Theorem 7.6.17. *Let E be a well-formed specification. If for each process-declaration $n(x_1 : S_1, ..., x_m : S_m) = p \in Sig(E).Proc$ it holds that $\langle n : S_1 \times ... \times S_m\rangle \notin HV(p, \emptyset)$ and for each process-declaration $n = p \in Sig(E).Proc$ $n \notin HV(p, \emptyset)$, then E is guarded.*

7.7 Appendix An SDF-syntax for μCRL

We present an SDF-syntax for μCRL [10] which serves two purposes. It provides a syntax that does not employ special characters and, using it as input for the ASF+SDF-system, it yields an interactive editor for μCRL-specifications (see eg. [11]). The ASF+SDF system is also used to provide a well-formedness checker [17].

According to the convention in SDF we write syntactical categories with a capital and keywords with small letters. The first LAYOUT rule says that spaces (' '), tabs (`\t`) and newlines (`\n`) may be used to generate some attractive layout and are not part of the μCRL specification itself. The second LAYOUT rule says that lines starting with a `%`-sign followed by zero or more non-newline characters (`~[\n]*`) followed by a newline (`\n`) must be taken as comments and are therefore also not a part of the μCRL syntax.

In this syntax *names* are arbitrary strings over `a-z`, `A-Z` and `0-9` except that keywords are not *names*. In the context free syntax most items are self-explanatory. The symbol `+` stands for one or more and `*` for zero or more occurrences. For instance { `Name ","`}`+` is a list of one or more *names* separated by commas.

The phrase `right` means that an operator is right-associative and `assoc` means that an operator is associative. The phrase `bracket` says that the defined construct is not an operator, but just a way to disambiguate the construction of a syntax tree. Instead of δ, ∂, τ and ρ we write `delta`, `encap`, `tau`, `hide` and `rename`. These keywords are taken from PSF [18].

The priorities say that '`.`' has highest and `+` has lowest priority on *process-expressions*.

```
exports
  sorts
    Name
    Name-list
    X-name-list
    Space-name-list
    Sort-specification
    Function-specification
    Function-declaration
    Rewrite-specification
    Variable-declaration-section
    Variable-declaration
    Data-term
    Rewrite-rules-section
    Rewrite-rule
    Process-expression
    Renaming-declaration
    Single-variable-declaration
    Process-specification
    Process-declaration
    Action-specification
    Action-declaration
    Communication-specification
    Communication-declaration
```

```
    Specification

lexical syntax
    [ \t\n]                                     -> LAYOUT
    "%" ~[\n]* "\n"                             -> LAYOUT
    [a-zA-Z0-9]*                                -> Name
context-free syntax
    { Name ","}+                                -> Name-list
    { Name "#"}+                                -> X-name-list
      Name+                                     -> Space-name-list
    sort Space-name-list                        -> Sort-specification
    func Function-declaration+                  -> Function-specification
    Name-list ":" X-name-list "->" Name         -> Function-declaration
    Name-list ":" "->" Name                     -> Function-declaration

    Variable-declaration-section
            Rewrite-rules-section               -> Rewrite-specification
    var Variable-declaration+                   -> Variable-declaration-section
                                                -> Variable-declaration-section
    Name-list ":" Name                          -> Variable-declaration
    Name                                        -> Data-term
    Name "(" { Data-term "," }+ ")"             -> Data-term
    rew Rewrite-rule+                           -> Rewrite-rules-section
    Name "(" { Data-term "," }+ ")" "=" Data-term -> Rewrite-rule
    Name "=" Data-term                          -> Rewrite-rule

    Process-expression "+" Process-expression   -> Process-expression right
    Process-expression "||" Process-expression  -> Process-expression right
    Process-expression "||_" Process-expression -> Process-expression
    Process-expression "|"   Process-expression -> Process-expression right
    Process-expression "<|" Data-term "|>"
               Process-expression               -> Process-expression
    Process-expression "." Process-expression   -> Process-expression right
    delta                                       -> Process-expression
    tau                                         -> Process-expression
    encap "(" "{" Name-list "}" ","
                Process-expression ")"          -> Process-expression
    hide "(" "{" Name-list "}" ","
                Process-expression ")"          -> Process-expression
    rename "(" "{" { Renaming-declaration "," }+
              "}" "," Process-expression ")"    -> Process-expression
    sum "(" Single-variable-declaration ","
                  Process-expression ")"        -> Process-expression
    Name "(" { Data-term "," }+ ")"             -> Process-expression
    Name                                        -> Process-expression
    "(" Process-expression ")"                  -> Process-expression bracket

    Name "->" Name                              -> Renaming-declaration
    Name ":" Name                               -> Single-variable-declaration
    proc Process-declaration+                   -> Process-specification
    Name "(" { Single-variable-declaration "," }+ ")"
                "=" Process-expression          -> Process-declaration
    Name "=" Process-expression                 -> Process-declaration

    act Action-declaration+                     -> Action-specification
    Name-list ":" X-name-list                   -> Action-declaration
```

```
    Name                                        -> Action-declaration

    comm Communication-declaration+             -> Communication-specification
    Name "|" Name "=" Name                      -> Communication-declaration

    Sort-specification                          -> Specification
    Function-specification                      -> Specification
    Rewrite-specification                       -> Specification
    Action-specification                        -> Specification
    Communication-specification                 -> Specification
    Process-specification                       -> Specification
    Specification Specification                 -> Specification  assoc

 priorities
    "+" < { "||", "|", "||_"} < "<|" "|>" < "."
```

As an example we provide a μCRL-specification of an alternating bit protocol. This is almost exactly the protocol as described in [2] to which we also refer for an explanation.

```
sort Bool
func T,F:->Bool

sort D
func d1,d2,d3 : -> D

sort error
func e        : -> error

sort bit
func 0,1      : -> bit
     invert   : bit -> bit

rew  invert(1)=0
     invert(0)=1

act  r1,s4    : D
     s2,r2,c2 : D#bit
     s3,r3,c3 : D#bit
     s3,r3,c3 : error
     s5,r5,c5 : bit
     s6,r6,c6 : bit
     s6,r6,c6 : error

comm r2|s2 = c2
     r3|s3 = c3
     r5|s5 = c5
     r6|s6 = c6

proc S            = S(0).S(1).S
     S(n:bit)     = sum(d:D,r1(d).S(d,n))
     S(d:D,n:bit) = s2(d,n).((R6(invert(n))+r6(e)).S(d,n)+r6(n))

     R            = R(1).R(0).R
     R(n:bit)     = (sum(d:D,r3(d,n))+r3(e)).s5(n).R(n)+
                       sum(d:D,r3(d,invert(n)).s4(d).s5(invert(n)))
```

```
K           = sum(d:D,sum(n:bit,r2(d,n).(tau.s3(d,n)+tau.s3(e)))).K
L           = sum(n:bit,r5(n).(tau.s6(n)+tau.s6(e))).L

ABP         = hide({c2,c3,c5,c6},
                   encap({r2,r3,r5,r6,s2,s3,s5,s6},S||R||K||L))
```

References

[1] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. Report P9008, University of Amsterdam, Amsterdam, 1990. To appear in *Proceedings NATO Summer School, Marktoberdorf*, pages 407–419, 1990.

[2] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[3] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[4] J.A. Bergstra and J.V. Tucker. A characterisation of computable data types by means of a finite equational specification method. In J.W. de Bakker and J. van Leeuwen, editors, *Proceedings* 7^{th} *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.

[5] J.A. Bergstra and J.V. Tucker. The completeness of the algebraic specification methods for computable data types. *Information and Control*, 12:186–200, 1982.

[6] CCITT Working Party X/1. *Recommendation Z.100 (SDL)*, 1987.

[7] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65:122–157, 1985.

[8] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[9] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings* 16^{th} *ICALP,* Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation*.

[10] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual –. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[11] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[12] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[13] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[14] G. Huet and D.D. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

[15] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[16] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1991. To appear.

[17] H. Korver, 1991. Private communications.

[18] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[19] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[20] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[21] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II,* Garmisch, pages 199–225, Amsterdam, 1983. North-Holland.

[22] SPECS-semantics. *Definition of MR and CRL Version 2.1*, 1990.

8

Samenvatting: Procesalgebra en Operationele Semantiek

In de procesalgebra wordt het gedrag van systemen bestudeerd. Het begrip systeem kan daarbij ruim worden opgevat. Te denken valt aan mensen, robots en computers. Er wordt verondersteld dat systemen een aantal elementaire activiteiten kunnen verichten. Deze activiteiten heten atomaire acties. Zij worden meestal aangeduid met letters $a, b, c, \ldots$. Het gedrag van een systeem wordt beschreven aan de hand van deze atomaire acties, de 'gevolgd door' operatie $(\cdot)$, de keuze operatie $(+)$ en de parallel operatie $(\|)$. Het proces

$$(a \cdot (b + c)) \parallel (d \cdot e)$$

is opgebouwd uit de atomaire acties a, b, c, d en e en kan eerst een a gevolgd door een b of een c, met daaraan parallel een d gevolgd door een e uitvoeren.

Sommige processen gedragen zich precies zo als andere. Het proces $a + a$ kan kiezen tussen het uitvoeren van twee identieke a's. Omdat de a's gelijk zijn, heeft het maken van de keuze geen zin en gedraagt het proces zich precies zoals het proces a. Om dit aan te geven kunnen we een procesequivalentie definieren waarin a en $a + a$ gelijk zijn. Er zijn veel verschillende procesequivalenties bestudeerd. Deze equivalenties worden op allerlei manieren gekarakteriseerd. Vaak wordt een beperkt aantal gelijkheden (axioma's) gebruikt, waaruit alle andere gelijkheden tussen processen via de eigenschappen van '=' bewezen kunnen worden. Zie tabel 5.2 voor een aantal van zulke axiomastelsels.

Gelijkheden kunnen variabelen bevatten. In $x+y = y+x$ staan de variabelen x en y voor willekeurige processen. Een axiomastelsel heet ω-compleet als ook alle geldige gelijkheden *met variabelen* bewezen kunnen worden. Hoofdstuk 2 van dit proefschrift gaat over de vraag of verschillende axiomastelsels, geformuleerd in het proefschrift van Rob van Glabbeek, ω-compleet zijn. Om in te zien dat deze stelsels inderdaad ω-compleet zijn, wordt in hoofdstuk 2 een tot nu toe onbekende bewijstechniek gepresenteerd en toegepast.

In het proefschrift van Rob van Glabbeek wordt ook een procesequivalentie, met de naam branching bisimulatie, geintroduceerd. Van deze procesequivalentie wordt beweerd dat zij voordelen biedt boven de populaire en al veel langer bestaande zwakke bisimulatie. Iemand kan nu, gegeven twee processen, de vraag stellen of deze processen branching bisimulair zijn. In hoofdstuk 3 wordt een algoritme gegeven om dit in bepaalde gevallen uit te rekenen. Dit algoritme is al in een aantal gevallen toegepast waar andere algoritmes tekort schoten.

In hoofdstuk 6 komt de vraag aan de orde hoe procesalgebra moet worden aangepast om er tijdsafhankelijke processen mee te beschrijven. Er wordt een voorstel gedaan waarbij tijd op een zo simpel mogelijke wijze in de procesalgebra wordt geintroduceerd. Met dit voorstel zijn enkele real-time systemen beschreven en worden enkele real-time fenomenen bestudeerd.

In het laatste hoofdstuk wordt een eenvoudige taal gedefinieerd waarin processen met data worden gecombineerd. Het is de bedoeling dat met deze taal de relatie tussen data en processen op een gestructureerde manier wordt bestudeerd. Hopelijk leidt dit tot inzichten die kunnen bijdragen tot de verdere ontwikkeling van bestaande (en veel omvangrijkere) programmeer- en specificatietalen.

De hoofdstukken 4 en 5 zijn gewijd aan de (gestructureerde) operationele semantiek. Semantiek is het onderdeel van de informatica dat de betekenis van programmeer- en specificatietalen bestudeert. De operationale semantiek is daarvan een belangrijke categorie. Hierin wordt de betekenis van een proces beschreven aan de hand van een stap-functie die dikwijls wordt genoteerd door een $\longrightarrow$. De stap-functie beschrijft de stappen die een proces tijdens uitvoering zet. Als p en p' processen zijn, dan betekent

$$p \xrightarrow{a} p'$$

dat p onder uitvoering van de atomaire actie a een stap zet naar p'. Bijvoorbeeld

$$(a \cdot (b + c)) \parallel (d \cdot e) \xrightarrow{a} (b + c) \parallel (d \cdot e).$$

In de gestructureerde operationele semantiek wordt de stap-functie gedefinieerd met regels van de vorm

$$\frac{q_1 \xrightarrow{b_1} q_1' \quad \ldots \quad q_n \xrightarrow{b_n} q_n'}{p \xrightarrow{a} p'}.$$

Dit betekent dat wanneer de stappen $q_1 \xrightarrow{b_1} q_1' \quad \ldots \quad q_n \xrightarrow{b_n} q_n'$ kunnen plaatsvinden, dan kan $p \xrightarrow{a} p'$ ook plaatsvinden. Zo wordt de keuze operatie gedefinieerd door de twee regels:

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}, \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}.$$

De linker regel zegt dat $x + y$ ervoor kan kiezen zich te gedragen als x en de rechter regel zegt dat $x + y$ zich kan gedragen als y. Tabel 5.1 geeft de regels voor een procesalgebra.

In sommige regels in tabel 5.1 worden ook regels gebruikt met negatieve premissen. Deze zien er als volgt uit:

$$\frac{q_1 \stackrel{b_1}{\longrightarrow} q_1' \;\; ... \;\; q_n \stackrel{b_n}{\longrightarrow} q_n' \qquad r_1 \stackrel{c_1}{\not\longrightarrow} \;\; ... \;\; r_m \stackrel{c_m}{\not\longrightarrow}}{p \stackrel{a}{\longrightarrow} p'}.$$

Deze regel betekent dat indien $q_i \stackrel{b_i}{\longrightarrow} q_i'$ voor alle $1 \leq i \leq n$ kan plaatsvinden en r_j geen c_j-stap ($1 \leq j \leq m$) kan doen, we mogen concluderen dat $p \stackrel{a}{\longrightarrow} p'$ kan plaatsvinden. Dit kan soms tot problemen leiden. Stel namelijk dat we de volgende regel hebben:

$$\frac{p \stackrel{a}{\not\longrightarrow}}{p \stackrel{a}{\longrightarrow} p}.$$

Dit betekent dat p een a-stap kan doen, indien p die a-stap niet kan doen. Dit is met zichzelf in tegenspraak. Een dergelijke regel kan daarom geen stap-functie definieren. In hoofdstuk 4 wordt een methode voorgesteld om aan te tonen dat voor bepaalde verzamelingen regels dergelijke problemen zich niet voordoen. Deze verzamelingen regels karakteriseren op nette wijze een stap-functie. De methode uit hoofdstuk 4 is erg praktisch, maar niet voldoende krachtig voor een aantal interessante gevallen. In hoofdstuk 5 worden aanzienlijk krachtiger methodiek voorgesteld die echter niet zo praktisch is. Combinatie van de methoden uit beide hoofdstukken blijkt echter krachtig en praktisch.

R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* Proefschrift, Vrije Universiteit, Amsterdam, 1990.