# Structural Operational Semantics

Luca Aceto* Wan Fokkink† Chris Verhoef‡

## Contents



---

*BRICS (**B**asic **R**esearch **i**n **C**omputer **S**cience), Centre of the Danish National Research Foundation, Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7-E, DK-9220 Aalborg Ø, Denmark. Email: `luca@cs.auc.dk`. Partially supported by a grant from the Italian CNR, Gruppo Nazionale per l'Informatica Matematica (GNIM).

†CWI, Department of Software Engineering, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Email: `wan@cwi.nl`. Partially supported by a grant from the Nuffield Foundation.

‡University of Amsterdam, Department of Computer Science, Programming Research Group, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands. Email: `x@wins.uva.nl`.

# 1 Introduction

The importance of giving precise semantics to programming and specification languages was recognized since the sixties with the development of the first high-level programming languages (cf., e.g., [30, 206] for some early accounts). The use of operational semantics — i.e., of a semantics that explicitly describes how programs compute in stepwise fashion, and the possible state-transformations they perform — was already advocated by McCarthy in [147], and elaborated upon in references like [142, 143]. Examples of full-blown languages that have been endowed with an operational semantics are Algol 60 [140], PL/I [173], and CSP [178].

Structural operational semantics (SOS) [177] provides a framework to give an operational semantics to programming and specification languages. In particular, because of its intuitive appeal and flexibility, SOS has found considerable application in the study of the semantics of concurrent processes, where, despite successful work by, among others, de Bakker, Zucker, Hennessy, and Abramsky (see, e.g., [1, 31, 117, 120, 122, 125, 150]), the methods of denotational semantics appear to be difficult to apply in general. SOS generates a labelled transition system, whose states are the closed terms over an algebraic signature, and whose transitions between states are obtained inductively from a collection of so-called transition rules of the form $\frac{\text{premises}}{\text{conclusion}}$. A typical example of a transition rule is

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

stipulating that if $t \xrightarrow{a} t'$ holds for certain closed terms $t$ and $t'$, then so does $t \| u \xrightarrow{a} t' \| u$ for each closed term $u$. In general, validity of the premises of a transition rule, under a certain substitution, implies validity of the conclusion of this rule under the same substitution.

Recently, SOS has been successfully applied as a formal tool to establish results that hold for classes of process description languages. This has allowed for the generalization of well-known results in the field of process algebra, and for the development of a meta-theory for process calculi based on the realization that many of the extant results in this field only depend upon general semantic properties of language constructs. The concept of a rule format has played a major role in the development of the meta-theory of process description languages, and several such formats have been proposed in the research literature. A principal aim of this chapter is to give an exposition on existing rule formats. Each of the formats surveyed here comes equipped with a rich body of results that are guaranteed to hold for any process calculus whose SOS is within that format.

Predicates in SOS semantics can be coded as binary relations [111]. Moreover, negative premises can often be expressed positively using predicates [27]. However, in the literature we see more and more that SOS definitions are decorated with predicates and/or negative premises. For example, predicates are used to express matters like (un)successful termination, convergence, divergence [10], enabledness [41], maximal delay, and side conditions [165]. Negative premises are used to describe, e.g., deadlock detection [137], sequencing [55], priorities [24, 65], probabilistic behaviour [139], urgency [58], and various real [136] and discrete time [23, 127, 223] settings. Since predicates and negative premises are so pervasive, and often lead to cleaner semantic descriptions for many features and constructs of interest, we present the theory of SOS in a setting that deals explicitly with these notions as much as possible. We hope that this makes this chapter a useful reference guide to the literature on the use of SOS in process algebra.

The organization of this chapter is as follows. Sect. 2 presents the preliminaries of SOS theory, and contains some standard SOS definitions that serve as running examples. Sect. 3 gives an overview of the different ways to give meaning to SOS definitions. Sect. 4 presents syntactic constraints under which an extension of an SOS definition does not influence some properties of the original SOS definition. Sect. 5 studies a wide range of syntactic formats for SOS definitions that guarantee that the semantics of a term is determined by the semantics of its arguments, and focuses on the connection between SOS semantics and complete proof systems. Sect. 6 describes a formalism to deal with variable binders explicitly. Finally, Sect. 7 pays attention to the automatic generation of fully abstract denotational models of process calculi from their SOS semantics.

**On Terminology: Structural vs Structured Operational Semantics** As mentioned above, in this chapter we shall use the acronym SOS to stand for *Structural Operational Semantics*. The adjective *structural* was used by Plotkin in the title of his seminal set of lecture notes [177] as this approach to giving formal semantics for programming and specification languages places great emphasis on defining the effect of running a program in terms of its structure. Moreover, the term Structural Operational Semantics is the most commonly used in the literature on semantics of programming languages and in various textbooks on this topic (see, e.g., [113, 119, 168]). The form of semantics we describe in this chapter is sometimes also called "Plotkin-style" operational semantics because of the aforementioned influential DAIMI report of Plotkin [177] and several papers in which he used this kind of specification. Some authors (see, e.g., [113]) prefer to use the term *transition semantics* to emphasize that transitions between program states are the main objects of study in this form of semantics. This terminology, albeit more descriptive in this context than "structural" or "Plotkin-style", has the drawback of being applicable to a range of operational semantics—such as those for automata and Petri nets [184]—that are rather different in nature from those that we deal with in this chapter. In [110, 111], Groote and Vaandrager used the acronym SOS to stand for *Structured Operational Semantics*. Their aim was to emphasize that a transition system specification that leads to a transition system for which bisimulation equivalence [171] is not a congruence should not be called *structured*, even though it is possibly compositional on the level of concrete transition systems. We have shunned from adopting their terminology as it is only used in the process algebra literature, and may be construed as suggesting that other forms of operational semantics are unstructured.

**Disclaimer** In this chapter, we focus on the results on the theory of SOS that, we feel, have the most interest from the point of view of process algebra. It is, however, a sign of the maturity of this field that SOS has found applications in many other settings. The original motivation for the development of SOS was to give semantics to programming languages, and the success of this endeavour is witnessed by the growing number of real-life programming languages that have been given *usable* semantic descriptions by means of SOS (see, e.g., [43, 162, 173, 178, 185].) As other applications of SOS, we limit ourselves to mentioning here that:

- the operational approach to type soundness, pioneered in [236], is now the preferred choice over methods based upon denotational semantics;
- the correctness of hardware implementations of real-life programming languages, and of compilation techniques, has been established using SOS [43, 233];
- the fit between reasonable operational extensions for the language PCF [175] and Scott's original lattice model for it has been studied in [46] within the framework of SOS;
- the derivation of proof rules for functional languages from their operational specifications has been investigated in [194], building upon the work in [8] (cf. Sect. 5.4.5).

These are only a few of the many interesting examples of applications of SOS that are not covered in this chapter. We hope that the reader will be tempted to explore them, and possibly to contribute to this fascinating research area.

**Acknowledgments** Our thoughts on the theory of SOS have been shaped by the inspiring work of, and collaborations with, many researchers. We cannot thank them all explicitly here. However, it will be evident to the readers of this chapter that the theory we survey, and the presentation we give of it, would not have been possible without the work of our colleagues. In particular, the ideas and work of Bard Bloom, Robert De Simone, Rob van Glabbeek (on whose work Sect. 3 is heavily based), Jan Friso Groote and Frits Vaandrager have been most influential. We hope that the list of references will prove useful in guiding the interested readers to the original sources for our subject matter. Finally, we thank Davide Marchignoli, Simone Tini and an anonymous referee for their thorough reading of a draft of this chapter.

## 2 Preliminaries

In this section we present the basic notions from process theory that are needed in the remainder of this chapter. The presentation is necessarily brief, and the interested reader is warmly encouraged to consult the references for much more information and motivation on the background material to our subject matter. We hope, however, that the basic definitions and results mentioned in this section will help the reader go through the material presented in this chapter with some ease.

## 2.1 Labelled Transition Systems

We begin by reviewing the model of *labelled transition systems* [134, 177], which are used to express the operational semantics of many process calculi. They consist of binary relations between states, carrying an action label, and predicates on states. Intuitively, $s \xrightarrow{a} s'$ expresses that state $s$ can evolve into state $s'$ by the execution of action $a$, while $sP$ expresses that predicate $P$ holds in state $s$. For convenience of terminology, we refer to both binary relations and predicates on states as *transitions*.

**Definition 2.1 (Labelled transition system)** *A* labelled transition system *(LTS) is a quadruple* $(\mathsf{Proc}, \mathsf{Act}, \left\{\xrightarrow{a} \mid a \in \mathsf{Act}\right\}, \mathsf{Pred})$*, where:*

- $\mathsf{Proc}$ *is a set of* states*, ranged over by* $s$*;*
- $\mathsf{Act}$ *is a set of* actions*, ranged over by* $a, b$*;*
- $\xrightarrow{a} \subseteq \mathsf{Proc} \times \mathsf{Proc}$ *for every* $a \in \mathsf{Act}$*. As usual, we use the more suggestive notation* $s \xrightarrow{a} s'$ *in lieu of* $(s, s') \in \xrightarrow{a}$*, and write* $s \stackrel{a}{\nrightarrow}$ *if* $s \xrightarrow{a} s'$ *for no state* $s'$*;*
- $P \subseteq \mathsf{Proc}$ *for every* $P \in \mathsf{Pred}$*. We write* $sP$ *(resp.* $s\neg P$*) if state* $s$ *satisfies (resp. does not satisfy) predicate* $P$*.*

*Binary relations* $s \xrightarrow{a} s'$ *and unary predicates* $sP$ *in an LTS are called* transitions.

In what follows, we shall sometimes identify an LTS with the set of its transitions. We trust that the meaning will always be clear from the context.

**Definition 2.2 (Finiteness conditions on an LTS)** *An LTS is:*

- finitely branching *if for every state* $s$ *there are only finitely many outgoing transitions* $s \xrightarrow{a} s'$*;*
- regular *if it is finitely branching and each state can reach only finitely many other states;*
- finite *if it is finitely branching and there is no infinite sequence of transitions* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$*.*

**Remark:** The conditions of regularity and finiteness defined above are usually used at the level of *process graphs*, i.e., transition systems with a distinguished initial state from which all other states are reachable in zero or more transitions. In particular, the above definition ensures that an LTS is finite or regular if so are all the process graphs obtained by choosing an arbitrary state as the initial one, removing all the states that are unreachable from it, and restricting the transition relations to the set of reachable states. Note that the notion of regularity defined above is a purely "syntactic" one. For instance, the LTS defined by

$$\{n \xrightarrow{a} n+1 \mid n \in \mathbb{N}\}$$

is not regular according to the above definition, even though it is the unfolding of the regular LTS $0 \xrightarrow{a} 0$. To define more semantic notions of regularity one has to work modulo some notion of behavioural equivalence. (See the following section and elsewhere in this handbook for information on behavioural equivalences over states of LTSs.)

## 2.2 Behavioural Equivalences

LTSs describe the operational behaviour of processes in great detail. In order to abstract away from irrelevant information on the way that processes compute, a wealth of notions of behavioural equivalence (i.e., a relation that is reflexive, transitive, and symmetric) and preorder (i.e., a relation that is reflexive and transitive) over the states of an LTS have been studied in the literature on process theory. A systematic investigation of these notions is presented in [96, 99] (see also [95, Chapter 1] and elsewhere in this handbook), where van Glabbeek presents the linear time/branching time spectrum. This lattice contains all the known behavioural equivalences and preorders over LTSs, ordered by inclusion. We investigate only a fragment of this spectrum, which we now proceed to present for the sake of completeness.

**Definition 2.3 (Simulation, ready simulation, and bisimulation)** *Assume an LTS.*

- *A binary relation $\mathcal{R}$ on states is a* simulation *if whenever $s_1 \mathrel{\mathcal{R}} s_2$:*
  - *if $s_1 \xrightarrow{a} s_1'$, then there is a transition $s_2 \xrightarrow{a} s_2'$ such that $s_1' \mathrel{\mathcal{R}} s_2'$;*
  - *if $s_1 P$, then $s_2 P$.*
- *A binary relation $\mathcal{R}$ on states is a* ready simulation *if it is a simulation with the property that, whenever $s_1 \mathrel{\mathcal{R}} s_2$:*
  - *if $s_1 \stackrel{a}{\nrightarrow}$, then $s_2 \stackrel{a}{\nrightarrow}$;*

- *if $s_1 \neg P$, then $s_2 \neg P$.*

- *A* bisimulation *is a symmetric simulation.*

*We write $s_1 \sqsubseteq_S s_2$ (resp. $s_1 \sqsubseteq_{RS} s_2$) if there is a simulation (resp. a ready simulation) $\mathcal{R}$ with $s_1 \mathrel{\mathcal{R}} s_2$. Two states $s_1, s_2$ are* bisimilar, *written $s_1 \underline{\leftrightarrow} s_2$, if there is a bisimulation relation that relates them. Henceforth the relation $\underline{\leftrightarrow}$ is referred to as* bisimulation equivalence.

Bisimulation equivalence [156, 171] relates two states in an LTS precisely when they have the same branching structure. Simulation (see, e.g., [171]) and ready simulation (also known as $\frac{2}{3}$ bisimulation) [55, 138] relax this requirement to different degrees.

We present seven more preorders, which are induced by yet further ways of abstracting away from the full branching structure of LTSs. They are based on (decorated) versions of traces.

**Definition 2.4 (Trace semantics)** *Given an LTS, a sequence*

$$\varsigma = a_1 \cdots a_n \in \mathsf{Act}^* \ ,$$

*for $n \in \mathbb{N}$, is a trace of state $s_0$ if there exist states $s_1, \ldots, s_n$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ (abbreviated by $s_0 \xrightarrow{\varsigma} s_n$). Moreover, $\varsigma P$ with $\varsigma \in \mathsf{Act}^*$ and $P \in \mathsf{Pred}$ is a trace of state $s$ if there exists a state $s'$ such that $s \xrightarrow{\varsigma} s'P$. We write $s \sqsubseteq_T s'$ if the set of traces of $s$ is included in that of $s'$.*

For a state $s$ we define (here, and in what follows, we use the symbol $\triangleq$ to stand for “equals by definition”):

$$\mathsf{initials}(s) \ \triangleq \ \{a \in \mathsf{Act} \mid \exists s' \in \mathsf{Proc}\,(s \xrightarrow{a} s')\} \ \cup \ \{P \in \mathsf{Pred} \mid sP\} \ .$$

**Definition 2.5 (Decorated trace semantics)** *Assume an LTS.*

- Ready traces. *A sequence $X_0 a_1 X_1 \cdots a_n X_n$ (with $n \in \mathbb{N}$), where $X_i \subseteq \mathsf{Act} \cup \mathsf{Pred}$ and $a_i \in \mathsf{Act}$ for $i = 0, \ldots, n$, is a* ready trace *of state $s_0$ if $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ and $\mathsf{initials}(s_i) = X_i$ for $i = 0, \ldots, n$. We write $s \sqsubseteq_{RT} s'$ if the set of ready traces of $s$ is included in that of $s'$.*

- Failure traces. *For $X$ be a subset of $\mathsf{Act} \cup \mathsf{Pred}$, we define $s_0 \xrightarrow{X} s_1$ if $s_0 = s_1$ and $\mathsf{initials}(s_0) \cap X = \varnothing$. A sequence $\varsigma \in (\mathsf{Act} \cup 2^{(\mathsf{Act} \cup \mathsf{Pred})})^*$ is a* failure trace *of state $s_0$ if $s_0 \xrightarrow{\varsigma} s_1$ for some state $s_1$. We write $s \sqsubseteq_{FT} s'$ if the set of failure traces of $s$ is included in that of $s'$.*

- Readies. *A pair* $(\varsigma, X)$ *with* $\varsigma \in \mathsf{Act}^*$ *and* $X \subseteq \mathsf{Act} \cup \mathsf{Pred}$ *is a* ready *of state* $s_0$ *if* $s_0 \stackrel{\varsigma}{\to} s_1$ *for some state* $s_1$ *with* $\mathsf{initials}(s_1) = X$*. We write* $s \underset{\sim}{\sqsubset}_R s'$ *if the set of readies of* $s$ *is included in that of* $s'$*.*

- Failures. *A pair* $(\varsigma, X)$ *with* $\varsigma \in \mathsf{Act}^*$ *and* $X \subseteq \mathsf{Act} \cup \mathsf{Pred}$ *is a* failure *of state* $s_0$ *if* $s_0 \stackrel{\varsigma}{\to} s_1$ *for some state* $s_1$ *with* $\mathsf{initials}(s_1) \cap X = \varnothing$*. We write* $s \underset{\sim}{\sqsubset}_F s'$ *if the set of failures of* $s$ *is included in that of* $s'$*.*

- Completed traces. $\varsigma \in \mathsf{Act}^*$ *is a* completed trace *of state* $s_0$ *if* $s_0 \stackrel{\varsigma}{\to} s_1$ *for some state* $s_1$ *with* $\mathsf{initials}(s_1) = \varnothing$*. Moreover,* $\varsigma P$ *with* $\varsigma \in \mathsf{Act}^*$ *and* $P \in \mathsf{Pred}$ *is a* completed trace *of state* $s_0$ *if* $s_0 \stackrel{\varsigma}{\to} s_0 P$ *for some state* $s_1$*. We write* $s \underset{\sim}{\sqsubset}_{CT} s'$ *if the set of completed traces of* $s$ *is included in that of* $s'$*.*

- Accepting traces. *Consider an LTS with* $\surd$ *as one of its predicates.* $\varsigma \in \mathsf{Act}^*$ *is an* accepting trace *of state* $s_0$ *if* $s_0 \stackrel{\varsigma}{\to} s_1 \surd$ *for some state* $s_1$*. We write* $s \underset{\sim}{\sqsubset}_{AT} s'$ *if the set of accepting traces of* $s$ *is included in that of* $s'$*.*

The decorated trace semantics defined above take predicates into account. However, most of the uses of these semantics in the literature on process theory occur in settings without predicates. The notion of an accepting trace is standard in formal language theory (see, e.g., [193]), but has not received widespread treatment in the literature on process theory.

For $\Theta \in \{S, RS, CT, RT, FT, F, R, L, T\}$, the relation $\underset{\sim}{\sqsubset}_\Theta$ is a preorder over states in arbitrary LTSs. Its kernel is denoted by $\simeq_\Theta$; i.e., $s \simeq_\Theta s'$ iff $s \underset{\sim}{\sqsubset}_\Theta s'$ and $s' \underset{\sim}{\sqsubset}_\Theta s$. The following result is a standard one in process theory (cf., e.g., [96]).

**Proposition 2.6** *In any LTS,*

$$\begin{tikzcd}
 & & \underset{\sim}{\sqsubset}_S & \underset{\sim}{\sqsubset}_{FT} \arrow[dr] & & & \\
\underline{\leftrightarrow} \arrow[r] & \underset{\sim}{\sqsubset}_{RS} \arrow[r] \arrow[ur] & \underset{\sim}{\sqsubset}_{RT} \arrow[ur] \arrow[dr] & & \underset{\sim}{\sqsubset}_F \arrow[r] & \underset{\sim}{\sqsubset}_{CT} \arrow[r] & \underset{\sim}{\sqsubset}_{AT} \\
 & & & \underset{\sim}{\sqsubset}_R \arrow[ur] & & &
\end{tikzcd}$$

*where a directed edge from one relation to another means that the source of the edge is included in the target. Moreover,*

- $\underset{\sim}{\sqsubset}_S$ *is included in* $\underset{\sim}{\sqsubset}_{AT}$ *and* $\underset{\sim}{\sqsubset}_T$*, and*

- $\underset{\sim}{\sqsubset}_R$ *is included in* $\underset{\sim}{\sqsubset}_T$*.*

*The same inclusions hold for the kernels of the preorders.*

All the inclusions presented in the previous proposition are proper if the LTS under consideration includes, modulo bisimulation equivalence, the finite synchronization trees [153] (see also Sect. 7.1.3) used in the examples presented in [96].

## 2.3 Hennessy-Milner Logic

Modal and temporal logics of reactive programs have found considerable use in the theory and practice of concurrency (see, e.g., [78, 179, 210]). One of the earliest and most influential connections between logics of reactive programs and behavioural relations was given by Hennessy and Milner [124], who introduced a multi-modal logic and showed that it characterized bisimulation equivalence. We limit ourselves to briefly recalling the basic definitions and results on Hennessy-Milner logic. The interested reader is referred to, e.g., [124, 209] for more details and motivation. The following definition is standard, apart from the use of atomic propositions to cater for the presence of predicates in LTSs.

**Definition 2.7 (Hennessy-Milner logic)** *The set of* HML formulae *is given by the BNF grammar [18]*

$$\varphi ::= \ \mathsf{true} \mid P \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle a \rangle \varphi$$

*where $a$ and $P$ range over* $\mathsf{Act}$ *and* $\mathsf{Pred}$*, respectively.*

Given an LTS, the states $s$ that satisfy HML formula $\varphi$, written $s \models \varphi$, are defined inductively by:

$$\begin{array}{rcl} s \models \mathsf{true} & & \\ s \models P & \iff & sP \\ s \models \neg\varphi & \iff & \text{not } s \models \varphi \\ s \models \varphi_1 \wedge \varphi_2 & \iff & s \models \varphi_1 \text{ and } s \models \varphi_2 \\ s \models \langle a \rangle \varphi & \iff & s' \models \varphi \text{ for some } s' \text{ such that } s \xrightarrow{a} s' \ . \end{array}$$

Using negation and conjunction in HML, one can define the other standard boolean connectives. Two states $s, s'$ are considered equivalent with respect to HML, written $s \sim_{\mathrm{HML}} s'$, iff for all HML formulae $\varphi$: $s \models \varphi \iff s' \models \varphi$. The following seminal result is due to Hennessy and Milner [124].

**Theorem 2.8** *The equivalence relations* $\underline{\leftrightarrow}$ *and* $\sim_{\mathrm{HML}}$ *coincide over finitely branching LTSs.*

The restriction to finitely branching LTSs in Thm. 2.8 can be dropped if infinitary conjunctions are allowed in the syntax of HML.

## 2.4 Term Algebras

This section reviews the basic notions of term algebras that will be needed in this chapter. We start from a countably infinite set $\mathsf{Var}$ of variables, ranged over by $x, y, z$.

**Definition 2.9 (Signature)** *A* signature $\Sigma$ *is a set of* function symbols, *disjoint from* $\mathsf{Var}$, *together with an* arity *mapping that assigns a natural number* $ar(f)$ *to each function symbol* $f$. *A function symbol of arity zero is called a* constant, *while function symbols of arity one and two are called* unary *and* binary, *respectively.*

The arity of a function symbol represents its number of arguments.

**Definition 2.10 (Term)** *The set* $\mathbb{T}(\Sigma)$ *of* (open) terms *over a signature* $\Sigma$, *ranged over by* $t, u$, *is the least set such that:*

- *each* $x \in \mathsf{Var}$ *is a term;*
- $f(t_1, \ldots, t_{ar(f)})$ *is a term, if* $f$ *is a function symbol and* $t_1, \ldots, t_{ar(f)}$ *are terms.*

$\mathrm{T}(\Sigma)$ *denotes the set of* closed terms *over* $\Sigma$, i.e., *terms that do not contain variables.*

For a constant $a$, the term $a()$ is abbreviated to $a$. By convention, whenever we write a term-like phrase (e.g., $f(t, u)$), we intend it to be a term (i.e., $f$ is binary).

A *substitution* is a mapping $\sigma : \mathsf{Var} \to \mathbb{T}(\Sigma)$. A substitution is *closed* if it maps each variable to a closed term in $\mathrm{T}(\Sigma)$. A substitution extends to a mapping from terms to terms as usual; the term $\sigma(t)$ is obtained by replacing occurrences of variables $x$ in $t$ by $\sigma(x)$. A *context* $C[x_1, \ldots, x_n]$ denotes an open term in which at most the distinct variables $x_1, \ldots, x_n$ appear. The term $C[t_1, \ldots, t_n]$ is obtained by replacing all occurrences of variables $x_i$ in $C[x_1, \ldots, x_n]$ by $t_i$, for $i = 1, \ldots, n$.

**Definition 2.11 (Congruence)** *Assume a signature $\Sigma$. An equivalence relation (resp. preorder) $\mathcal{R}$ over* $\mathrm{T}(\Sigma)$ *is a* congruence *(resp.* precongruence*) if, for all $f \in \Sigma$,*

$$t_i \mathcal{R} u_i \textit{ for } i = 1, \ldots, ar(f) \textit{ implies } f(t_1, \ldots, t_{ar(f)}) \mathcal{R} f(u_1, \ldots, u_{ar(f)}) \ .$$

## 2.5 Transition System Specifications

In the remainder of this chapter, the set Proc of states will in general consist of the closed terms over some signature. We proceed to introduce the main object of study in the field of SOS, viz. a *transition system specification*, being a collection of inductive proof rules to derive the transitions over the set of closed terms.

**Definition 2.12 (Transition system specification)** *Let $\Sigma$ be a signature, and let $t$ and $t'$ range over $\mathbb{T}(\Sigma)$. A* transition rule *$\rho$ is of the form $H/\alpha$, with $H$ a set of* positive premises *$t \xrightarrow{a} t'$ and $tP$, and of* negative premises *$t \stackrel{a}{\nrightarrow}$ and $t\neg P$. Moreover, the* conclusion *$\alpha$ is of the form $t \xrightarrow{a} t'$ or $tP$. The left-hand side of the conclusion is the* source *of $\rho$, and if the conclusion is of the form $t \xrightarrow{a} t'$, then its right-hand side is the* target *of $\rho$. A transition rule is closed if it does not contain variables.*

*A* transition system specification *(TSS) is a set of transition rules. A TSS is* positive *if its transition rules do not contain negative premises.*

For the sake of clarity, transition rules will often be displayed in the form $\frac{H}{\alpha}$, and the premises of a transition rule will not always be presented using proper set notation. The first systematic study of TSSs may be found in [199], while the first study of TSSs with negative premises appeared in [54].

We proceed to define when a transition is provable from a TSS. The following notion of a proof from [102] generalizes the standard definition (see, e.g., [111]) by allowing for the derivation of closed transition rules. The derivation of a transition $\alpha$ corresponds to the derivation of the closed transition rule $H/\alpha$ with $H = \varnothing$. The case $H \neq \varnothing$ corresponds to the derivation of $\alpha$ under the assumptions in $H$.

**Definition 2.13 (Literal)** Positive literals *are transitions $t \xrightarrow{a} t'$ and $tP$, while* negative literals *are expressions $t \stackrel{a}{\nrightarrow}$ and $t\neg P$, where $t$ and $t'$ range over the collection of closed terms. A* literal *is a positive or negative literal.*

**Definition 2.14 (Proof)** *Let $T$ be a TSS. A* proof *of a closed transition rule $H/\alpha$ from $T$ is a well-founded, upwardly branching tree whose nodes are labelled by literals, where the root is labelled by $\alpha$, and if $K$ is the set of labels of the nodes directly above a node with label $\beta$, then*

1. *either* $K = \varnothing$ *and* $\beta \in H$,

2. *or* $K/\beta$ *is a closed substitution instance of a transition rule in* $T$.

*If a proof of* $H/\alpha$ *from* $T$ *exists, then* $H/\alpha$ *is* provable *from* $T$, *notation* $T \vdash H/\alpha$.

## 2.6 Examples of TSSs

In this section we present some TSSs from the literature, which will serve as running examples in sections to come. Abundant examples of the systematic use of SOS can be found, e.g., in [28, 223] and elsewhere in this handbook. Hartel [115] recently developed a tool environment LATOS for the animation of such TSSs, based on functional programming languages.

### 2.6.1 Basic Process Algebra with Empty Process

The signature of Basic Process Algebra with empty process [229], denoted by $\mathrm{BPA}_\epsilon$, consists of the following operators:

- a set $\mathsf{Act}$ of constants, representing indivisible behaviour;
- a special constant $\epsilon$, called *empty process*, representing successful termination;
- a binary operator $+$, called *alternative composition*, where a term $t_1 + t_2$ represents the process that executes either $t_1$ or $t_2$;
- a binary operator $\cdot$, called *sequential composition*, where a term $t_1 \cdot t_2$ represents the process that executes first $t_1$ and then $t_2$.

So the BNF grammar for $\mathrm{BPA}_\epsilon$ is (with $a \in \mathsf{Act}$):

$$t ::= \; a \mid \epsilon \mid t_1 + t_2 \mid t_1 \cdot t_2 \;\; .$$

The intuition above for the operators in $\mathrm{BPA}_\epsilon$ is formalized by the transition rules in Table 1 from [29], which constitute the TSS for $\mathrm{BPA}_\epsilon$. This TSS defines transitions $t \xrightarrow{a} t'$ to express that term $t$ can evolve into term $t'$ by the execution of action $a \in \mathsf{Act}$, and transitions $t\surd$ to express that term $t$ can terminate successfully. The variables $x$, $x'$, $y$, and $y'$ in the transition rules range over the collection of closed terms, while the $a$ ranges over $\mathsf{Act}$.

$$\frac{}{a \xrightarrow{a} \epsilon} \qquad \frac{}{\epsilon\surd}$$

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

$$\frac{x\surd \quad y\surd}{x \cdot y\surd} \qquad \frac{x\surd \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} \qquad \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$$

Table 1: Transition Rules for $\mathrm{BPA}_\epsilon$.

#### 2.6.2 Priorities

The language $\mathrm{BPA}_{\epsilon\theta}$ is obtained by adding the priority operator $\theta$ from [24] to $\mathrm{BPA}_\epsilon$. This function symbol assumes a partial order $<$ on $\mathsf{Act}$. Intuitively, the process $\theta(t)$ is obtained by eliminating all transitions $s \xrightarrow{a} s'$ from the process $t$ for which there is a transition $s \xrightarrow{b} s''$ with $a < b$. For example, if $a < b$ then $\theta(a+b)$ can execute the action $b$ but not the action $a$. The semantics of the priority operator is captured by the transition rules in Table 2. The TSS for $\mathrm{BPA}_{\epsilon\theta}$ consists of the transition rules in Tables 1 and 2.

$$\frac{x\surd}{\theta(x)\surd} \qquad \frac{x \xrightarrow{a} x' \qquad x \stackrel{b}{\nrightarrow} \text{ for } a < b}{\theta(x) \xrightarrow{a} \theta(x')}$$

Table 2: Transition Rules for the Priority Operator.

#### 2.6.3 Discrete Time

Our final example is the TSS for an extension of $\mathrm{BPA}_\epsilon$ with relative discrete time, denoted by $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ [23]. Time progresses in distinct time steps, where a transition $t \xrightarrow{\sigma} t'$ denotes passing to the next time slice. The syntax of $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ consists of the operators from $\mathrm{BPA}_\epsilon$ together with a unary operator $\sigma_d$ to represent a delay of one time unit. That is, a term $\sigma_d(t)$ can execute all transitions of $t$ delayed by one time step. A term $t + t'$ can evolve into the next time slice if $t$ or $t'$ can evolve into the next time slice. The transition rules dealing with time steps are presented in Table 3. The TSS for $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ consists of the transition rules in Tables 1 and 3.

$$\frac{}{\sigma_d(x) \xrightarrow{\sigma} x} \qquad \frac{x\surd \quad y \xrightarrow{\sigma} y'}{x \cdot y \xrightarrow{\sigma} y'} \qquad \frac{x \xrightarrow{\sigma} x'}{x \cdot y \xrightarrow{\sigma} x' \cdot y}$$

$$\frac{x \xrightarrow{\sigma} x' \quad y \xrightarrow{\sigma} y'}{x + y \xrightarrow{\sigma} x' + y'} \qquad \frac{x \xrightarrow{\sigma} x' \quad y \stackrel{\sigma}{\nrightarrow}}{x + y \xrightarrow{\sigma} x'} \qquad \frac{y \xrightarrow{\sigma} y' \quad x \stackrel{\sigma}{\nrightarrow}}{x + y \xrightarrow{\sigma} y'}$$

Table 3: Transition Rules for Discrete Time.

## 3 The Meaning of TSSs

A positive TSS specifies an LTS in a straightforward way as the set of all provable transitions (cf. Def. 2.14). However, as Groote [107, 108] pointed out, it is much less trivial to associate an LTS with a TSS containing negative premises. Several solutions were investigated in [56, 57, 107, 108], mostly originating from logic programming. This section presents an overview of how to associate one or more LTSs with a TSS. Our presentation here is heavily based upon the excellent systematic analysis of the meaning of TSSs by van Glabbeek [100, 102], and we heartily refer the reader to *op. cit.* for more details.

To see that it is sometimes unclear what the meaning of a TSS with negative premises is, consider the TSS $T_1$ consisting of the constant $a$ and transition rules

$$T_1 \quad \boxed{\frac{a\neg P_2}{aP_1} \quad \frac{a\neg P_1}{aP_2}}$$

$T_1$ can be regarded as an example of a TSS that does not specify a well-defined LTS. The above example suggests that some TSSs may indeed be meaningless. Hence there are two questions to answer:

$$\textit{Which TSSs are meaningful,} \tag{1}$$

$$\textit{and which LTSs can be associated with them?} \tag{2}$$

The papers [100, 102] present several possible answers to these questions, each consisting of a class of TSSs and a mapping from this class to LTSs.

Two such answers are consistent if they agree upon which LTS to associate with a TSS in the intersection of their domains. Answer $S_2$ extends answer $S_1$ if the class of meaningful TSSs according to $S_2$ extends that of $S_1$, and the two are consistent.

The collection of answers proposed by van Glabbeek in *op. cit.* can be grouped into those with a model-theoretic and those with a proof-theoretic flavour. These we now proceed to present.

### 3.1 Model-Theoretic Answers

**Answer 1** A first answer to questions (1) and (2) is to take the class of positive TSSs as the meaningful ones, and associate with each positive TSS the LTS consisting of the provable transitions.

Since negative premises sometimes allow for a clean description of important constructs found in programming and specification languages, the above answer is not really satisfactory. More general answers to questions (1) and (2) have been proposed in the literature. Before reviewing them, we recall two criteria from [55, 56] that can be imposed on reasonable answers.

**Definition 3.1 (Entailment)** *For an LTS $L$ and a set of literals $H$, we write $L \models H$ if:*

- $\alpha \in L$ *for all positive literals* $\alpha$ *in* $H$*;*
- $t \xrightarrow{a} t' \notin L$ *for all negative literals* $t \stackrel{a}{\nrightarrow}$ *in* $H$ *and all closed terms* $t'$*;*
- $tP \notin L$ *for all negative literals* $t\neg P$ *in* $H$*.*

**Definition 3.2 (Supported model)** *Let $T$ be a TSS and $L$ an LTS.*

- *$L$ is a* model *of $T$ if $\alpha \in L$ whenever there is a closed substitution instance $H/\alpha$ of a transition rule in $T$ with $L \models H$.*
- *$L$ is* supported *by $T$ if whenever $\alpha \in L$ there is a closed substitution instance $H/\alpha$ of a transition rule in $T$ with $L \models H$.*

The first requirement, of being a model, says that $L$ contains all transitions for which $T$ offers a justification. The second requirement, of being supported, says that $L$ only contains transitions for which $T$ offers a justification. Note that the LTS containing all possible transitions is a model of any TSS, while the LTS containing no transitions is supported by any TSS.

The following result is standard, and has its roots in the classic theory of inductive definitions.

**Proposition 3.3** *Let $T$ be a positive TSS and $L$ the set of transitions provable from $T$. Then $L$ is a supported model of $T$. Moreover, $L$ is the least model of $T$.*

Starting from Prop. 3.3, there are two ways to generalize Answer 1 to TSSs with negative premises.

**Answer 2** A TSS is meaningful iff it has a least model.

**Answer 3** A TSS is meaningful iff it has a least supported model.

Note that, in general, no unique least (supported) model may exist. A counter-example is given by the TSS $T_1$, which has two least models, namely $\{aP_1\}$ and $\{aP_2\}$, both of which are supported. Answers 2 and 3 are incomparable. For example, the TSS $T_2$ below has $\{aP_1\}$ as its least model, but no supported models. On the other hand, the TSS $T_3$ has two least models, namely $\{aP_1\}$ and $\{aP_2\}$, of which only the first one is supported, and this is its least supported model.

$$T_2 \quad \boxed{\frac{a\neg P_1}{aP_1}} \qquad\qquad T_3 \quad \boxed{\frac{a\neg P_2}{aP_1}}$$

Answers 2 and 3 both extend Answer 1, but they are inconsistent with each other. For example, the TSS $T_4$ below has a least model $\{aP_1\}$ and a least supported model $\{aP_1, aP_2\}$.

$$T_4 \quad \boxed{\frac{a\neg P_1}{aP_1} \quad \frac{aP_2}{aP_1} \quad \frac{aP_2}{aP_2}}$$

In [54, 55] the following answer was proposed.

**Answer 4** A TSS is meaningful iff it has a unique supported model.

The positive TSS $T_5$ below has two supported models, viz. $\varnothing$ and $\{aP_1\}$, so Answer 4 does not extend Answer 1.

$$T_5 \quad \boxed{\frac{aP_1}{aP_1}}$$

For the *GSOS languages* considered in *op. cit.* (cf. Sect. 5.4), Answer 4 coincides with all acceptable answers mentioned in this section. Note, however, that the least supported model of $T_4$ is also its unique supported model. This seems to entail that Answer 4 is not satisfactory for TSSs in general.

Fages [80] proposed a strengthening of the notion of support, in the setting of logic programming. Being supported means that a transition may only be present if there is a non-empty proof of its presence, starting from transitions that are also present. These premises in the proof may include the transition under derivation, thereby allowing for loops, as in the case of $T_4$. The notion of a *well-supported* model is based on the idea that the absence of a transition may be assumed a priori, provided that this assumption is consistent, but the presence of a transition needs to be proven, in the sense of Def. 2.14, building upon a set of assumptions that only contains negative literals.

**Definition 3.4 (Well-supported model)** *An LTS $L$ is a* well-supported model *of a TSS $T$ if for each transition $\alpha$ in $L$, $T$ proves a closed transition rule $N/\alpha$ where $N$ only contains negative literals and $L \models N$.*

A *stable* model, developed by Gelfond and Lifschitz [92] in the area of logic programming, and adapted to TSSs in [56, 57], only allows for transitions that are well-supported.

**Definition 3.5 (Stable model)** *An LTS $L$ is a* stable model *of a TSS $T$ if a transition $\alpha$ is in $L$ iff $T$ proves a closed transition rule $N/\alpha$ where $N$ contains only negative literals and $L \models N$.*

An LTS is a stable model of a TSS $T$ iff it is a well-supported model of $T$ [100, 102].

**Answer 5** A TSS is meaningful iff it has a unique stable model.

Answer 5 extends Answer 1, and it improves upon Answers 3 and 4 by rejecting the TSS $T_4$ as meaningless. It also improves upon Answer 2 by rejecting the TSS $T_2$ (whose least model was not supported). Furthermore, Answer 5 gives meaning to perfectly acceptable TSSs that could not be handled by Answers 1–4. As an example consider the TSS $T_6$ below.

$$T_6 \quad \boxed{\frac{aP_1}{aP_1} \qquad \frac{a\neg P_1}{aP_2}}$$

Since there is no compelling way to obtain $aP_1$, we would expect that $aP_1$ does not hold, and consequently that $aP_2$ holds. Indeed, $\{aP_2\}$ is the unique stable model of this TSS. However, $T_6$ has two least models, both of which are supported, namely $\{aP_1\}$ and $\{aP_2\}$.

A *three-valued* stable model, introduced by Przymusinski [183] in logic programming, partitions the set of transitions into three disjoint subsets: the set $C$ of transitions that are *certainly* true, the set $U$ of transitions for which it is *unknown* whether or not they are true, and the set $F$ of remaining transitions that are *false*.

**Definition 3.6 (Three-valued stable model)** *A disjoint pair of sets of transitions $\langle C, U\rangle$ constitutes a* three-valued stable model *for a TSS $T$ if:*

- *a transition $\alpha$ is in $C$ iff $T$ proves a closed transition rule $N/\alpha$ where $N$ contains only negative literals and $C \cup U \models N$;*
- *a transition $\alpha$ is in $C \cup U$ iff $T$ proves a closed transition rule $N/\alpha$ where $N$ contains only negative literals and $C \models N$.*

Each TSS has one or more three-valued stable models. For example, the TSS $T_1$ has $\langle\{aP_1\}, \varnothing\rangle$, $\langle\{aP_2\}, \varnothing\rangle$, and $\langle\varnothing, \{aP_1, aP_2\}\rangle$ as its three-valued stable models. Each TSS $T$ affords an *(information-)least* three-valued stable model $\langle C, U\rangle$, in the sense that the set $U$ is maximal. Przymusinski [183] showed that this least three-valued stable model coincides with the so-called well-founded model that was introduced by van Gelder, Ross, and Schlipf [90, 91] in logic programming.

**Answer 6** A TSS is meaningful iff its least three-valued stable model does not contain unknown transitions. The associated LTS consists of the true transitions in this three-valued stable model.

Answer 6 extends Answer 1 and is extended by Answer 5, but it is inconsistent with Answers 2–4. In particular, the TSSs $T_1$, $T_2$, and $T_4$ are outside its domain, while it associates $\{aP_1\}$ to $T_3$, $\varnothing$ to $T_5$, and $\{aP_2\}$ to $T_6$. In Sect. 5, Answer 6 will stand us in good stead, as it allows for the formulation of congruence results in the presence of negative premises (cf. Thm. 5.3, Thm. 5.50, and Thm. 5.53), where Answers 1–5 would all be unsatisfactory.

## 3.2 Proof-Theoretic Answers

*Note for the Reader.* In this section only we extend the notion of negative literals to expressions of the form $t \stackrel{a}{\nrightarrow} t'$. Intuitively, this expression denotes that term $t$ cannot evolve to term $t'$ by the execution of action $a$.

This section reviews possible answers to questions (1) and (2) based on a generalization of the concept of proof. van Glabbeek [102] proposed two generalizations of the concept of a proof in Def. 2.14, to enable the derivation of negative literals. These two generalizations are based on the notions of supported model (Def. 3.2) and well-supported model (Def. 3.4), respectively.

**Definition 3.7 (Denying literal)** *The following pairs of literals* deny *each other:*

- $t \xrightarrow{a} t'$ *and* $t \stackrel{a}{\nrightarrow} t'$;
- $t \xrightarrow{a} t'$ *and* $t \stackrel{a}{\nrightarrow}$;
- $tP$ *and* $t\neg P$.

**Definition 3.8 (Supported proof)** *A* supported proof *of a literal $\alpha$ from a TSS $T$ is like a proof (see Def. 2.14), but with one extra clause:*

3. *$\beta$ is negative, and for each closed substitution instance $H'/\gamma$ of a transition rule in $T$ such that $\gamma$ denies $\beta$, a literal in $H'$ denies one in $K$.*

*We write $T \vdash_s \alpha$ if a supported proof of $\alpha$ from $T$ exists.*

**Definition 3.9 (Well-supported proof)** *A* well-supported proof *of a literal $\alpha$ from a TSS $T$ is like a proof (Def. 2.14), but with one extra clause:*

3. *$\beta$ is negative, and for each set $N$ of negative literals such that $T \vdash N/\gamma$ for $\gamma$ a literal denying $\beta$, a literal in $N$ denies one in $K$.*

*We write $T \vdash_{ws} \alpha$ if a well-supported proof of $\alpha$ from $T$ exists.*

Clause 3 in Def. 3.9 allows one to infer $t \stackrel{a}{\nrightarrow} t'$ or $t\neg P$ whenever it is impossible to infer $t \xrightarrow{a} t'$ or $tP$, respectively. Clause 3 in Def. 3.8 allows such inferences only if the impossibility to derive $t \xrightarrow{a} t'$ or $tP$ can be detected by examining all possible proofs that consist of one step only. As a consequence, for each TSS, $\vdash_s$ is included in $\vdash_{ws}$. The following results stem from [102].

**Proposition 3.10** *For each TSS, the induced relation $\vdash_{ws}$ does not contain denying literals.*

**Proposition 3.11** *For any TSS $T$ and literal $\alpha$:*

1. *$T \vdash_s \alpha$ implies $L \models \alpha$ for each supported model $L$ of $T$;*
2. *$T \vdash_{ws} \alpha$ implies $L \models \alpha$ for each well-supported model $L$ of $T$.*

Following [102], we now introduce the concept of a *complete* TSS, in which every transition is either provable or refutable.

**Definition 3.12 (Completeness)** *A TSS $T$ is $x$-*complete *($x \in \{s, ws\}$) if for any transition $t \xrightarrow{a} t'$ (resp. $tP$) either $T \vdash_x t \xrightarrow{a} t'$ (resp. $T \vdash_x tP$) or $T \vdash_x t \stackrel{a}{\nrightarrow} t'$ (resp. $T \vdash_x t\neg P$).*

**Answer 7** A TSS is meaningful iff it is $s$-complete. The associated LTS consists of the $s$-provable transitions.

**Answer 8** A TSS is meaningful iff it is $ws$-complete. The associated LTS consists of the $ws$-provable transitions.

From now on, by 'complete' we shall mean '$ws$-complete'.

A TSS is complete iff its least three-valued stable model does not contain any unknown transitions (see [100]), so Answer 6 agrees with Answer 8. Moreover, Answer 8 extends Answer 7. In [56], an example in the area of process theory was given (viz. the modelling of a priority operator in basic process algebra with silent step) that can be handled by Answer 8 but not by Answer 7, showing that the full generality of Answer 8 can be useful in applications.

We proceed to show how to associate an LTS with any TSS, using the concept of a well-supported proof. As illustrated by the TSSs $T_1$ and $T_2$, such an LTS cannot always be a supported model. Since being a model is a basic requirement, van Glabbeek [102] proposed a universal answer that gives up the requirement of supportedness. Let us examine $T_1$. Since the associated LTS should be a model, it must contain either $aP_1$ or $aP_2$. By symmetry, the associated LTS should include both transitions. As there is no reason to include any more transitions, the LTS associated with $T_1$ should be $\{aP_1, aP_2\}$. These considerations lead to the following proposal.

**Answer 9** Any TSS is meaningful. The associated LTS consists of the transitions for which none of the denying negative literals are $ws$-provable.

Answer 9 is inspired by the observation in [102] that for each TSS, the set of transitions for which none of the denying negative literals are $ws$-provable constitutes a model. Answer 9 extends Answer 8, but it is inconsistent with Answers 2–5. Answer 9 associates the LTS $\{aP_1, aP_2\}$ with $T_1$, and the LTS $\{aP_1\}$ with $T_2$ and $T_4$.

### 3.3 Answers Based on Stratification

Finally, we review two methods to assign meaning to TSSs based on the technique of *(local) stratification*, as proposed in the setting of logic programming by Przymusinski [182]. This technique was adapted to TSSs in [107, 108].

**Definition 3.13 (Stratification)** *A mapping $S$ from transitions to ordinal numbers is a* stratification *of a TSS $T$ if for every transition rule $H/\alpha$ in $T$ and every closed substitution $\sigma$:*

- *for positive premises $\beta$ in $H$, $S(\sigma(\beta)) \leq S(\sigma(\alpha))$; and*
- *for negative premises $t \stackrel{a}{\nrightarrow}$ and $t\neg P$ in $H$, $S(\sigma(t) \stackrel{a}{\rightarrow} t') < S(\sigma(\alpha))$ for all closed terms $t'$ and $S(\sigma(t)P) < S(\sigma(\alpha))$, respectively.*

*A stratification is* strict *if $S(\sigma(\beta)) < S(\sigma(\alpha))$ also holds for all the positive premises $\beta$ in $H$. A TSS with a (strict) stratification is* (strictly) stratifiable.

In a stratifiable TSS no transition depends negatively on itself. An LTS associated with such a TSS may be built one stratum of transitions with the same $S$-value at a time. A transition with $S$-value zero is present only if it is provable in the sense of Def. 2.14, and as soon as the validity of all transitions with $S$-value no greater than $\kappa$ is known for some ordinal number $\kappa$, the validity of closed instantiations of negative premises that could occur in a proof of a transition with $S$-value $\kappa + 1$ is known, which determines the validity of those transitions.

Let $T$ be a TSS with a stratification $S$. The stratum $L_\kappa$ of transitions, for an ordinal number $\kappa$, is defined thus (using ordinal induction):

> $\alpha \in L_\kappa$ iff $S(\alpha) = \kappa$ and $T$ proves a closed transition rule $H/\alpha$ with $\cup_{\mu<\kappa} L_\mu \models H$.

Similarly, for a TSS $T$ with a strict stratification $S$, the stratum $M_\kappa$ of transitions, for an ordinal number $\kappa$, is defined thus (using ordinal induction):

> $\alpha \in M_\kappa$ iff $S(\alpha) = \kappa$ and there is a closed substitution instance $H/\alpha$ of a transition rule in $T$ with $\cup_{\mu<\kappa} M_\mu \models H$.

Groote [107, 108] proved that the sets $\cup_\kappa L_\kappa$ and $\cup_\kappa M_\kappa$ are independent of the chosen (strict) stratification. This justifies the following answers to questions (1) and (2).

**Answer 10** A TSS is meaningful iff it is stratifiable. The associated LTS is $\cup_\kappa L_\kappa$.

**Answer 11** A TSS is meaningful iff it is strictly stratifiable. The associated LTS is $\cup_\kappa M_\kappa$.

Answer 10 extends Answer 1 and is extended by Answer 8. Answer 11 is extended by Answers 7 and 10.

### 3.4 Evaluation of the Answers

We have presented several possible answers to the questions of which TSSs are meaningful and which LTSs are associated with them.

Answer 1 (positive) is the classical interpretation of TSSs without negative premises, and Answers 2 (least model) and 3 (least supported model) are two straightforward generalizations. Answer 4 (unique supported model) stems from [54], where it was used to ascertain that TSSs in GSOS format (cf. Sect. 5.4) are meaningful. The TSS $T_4$, however, shows that Answer 4 yields counter-intuitive results in general. Fortunately, TSSs in GSOS format are even strictly stratifiable, which is one of the most restrictive criteria for meaningful TSSs considered. For GSOS languages with recursion, however, it is no longer straightforward to find an associated LTS (see, e.g., [55]). A solution for this, involving a special divergence predicate, will be discussed in Sect. 5.4.6.

Answer 5 (unique stable) is generally considered to be the most general acceptable answer available. Answer 8 (complete) is the most general answer without undesirable properties. Answer 8 is based on a concept of provability incorporating the notion of *negation as failure* of Clark [63]. Answer 6 (no unknown transitions) agrees with Answer 8; i.e., a TSS is complete iff its least three-valued stable model does not contain unknown transitions. Answer 7 (complete with support) only yields unique supported models. Moreover, it is based on a notion of provability that is somewhat simpler to apply, and only incorporates the notion of *negation as finite failure* [63].

Answer 9 (irrefutable), which gives a meaning to each TSS, has the disadvantage that it sometimes yields unstable models, and even unsupported models. A good example from process algebra of a TSS without supported models is BPA with the priority operator, unguarded recursion, and renaming, as defined in [107, 108]. Although Answer 9 gives a meaning to this TSS, it appears rather arbitrary and not very useful. In particular, recursively defined processes do not satisfy their defining equations—a highly undesirable feature by all accounts.

Answer 10 (stratification) is perhaps the best known answer in logic programming. A variant that only allows TSSs with a unique supported model is Answer 11 (strict stratification). Answers 10 and 11 are of practical importance, because they are extended by Answer 8. Thus, giving a (strict) stratification is a useful tool for showing that a TSS is complete, and this technique is applied in several examples in the remainder of this chapter.

## 3.5 Applications

We show that the three TSSs from Sect. 2.6 are complete, using a stratification. We use the fact that Answer 8 extends Answers 1 and 10.

**BPA with Empty Process** The TSS for BPA$_\epsilon$ is positive.

**Priorities** The TSS for BPA$_{\epsilon\theta}$ is complete, which can be seen by giving a suitable stratification $S$, counting the number of occurrences of the priority operator in the left-hand side of a transition. That is, if the closed term $t$ contains $n$ occurrences of $\theta$, then $S(t \xrightarrow{a} t') = n$ and $S(t\surd) = n$. Consider for instance the second transition rule in Table 2:

$$\frac{x \xrightarrow{a} x' \qquad x \stackrel{b}{\nrightarrow} \ \text{for } a < b}{\theta(x) \xrightarrow{a} \theta(x')}$$

Clearly $S(t \xrightarrow{a} t') < S(\theta(t) \xrightarrow{a} \theta(t'))$ and $S(t \xrightarrow{b} u) < S(\theta(t) \xrightarrow{a} \theta(t'))$ for all closed terms $t$, $t'$, and $u$, because $\theta(t)$ contains one more occurrence of the priority operator than $t$. In a similar fashion it can be verified for the other transition rules of BPA$_{\epsilon\theta}$ that $S$ is a stratification. Hence, the TSS for BPA$_{\epsilon\theta}$ is complete.

**Discrete Time** The TSS for BPA$_\epsilon^{\mathrm{dt}}$ is complete, which can be seen by giving a suitable stratification $S$, counting the occurrences of alternative composition on the left-hand side of a timed transition. That is, if the closed term $t$ contains $n$ occurrences of $+$, then $S(t \xrightarrow{\sigma} t') = n$. Moreover, $S(t \xrightarrow{a} t') = 0$ for $a \in \mathsf{Act}$. Consider for instance the last transition rule in Table 3:

$$\frac{y \xrightarrow{\sigma} y' \qquad x \stackrel{\sigma}{\nrightarrow}}{x + y \xrightarrow{\sigma} y'}$$

Clearly $S(u \xrightarrow{\sigma} u') < S(t + u \xrightarrow{\sigma} u')$ and $S(t \xrightarrow{\sigma} t') < S(t + u \xrightarrow{\sigma} u')$ for all closed terms $t$, $t'$, $u$, and $u'$, because $t + u$ contains more occurrences of the alternative composition than $t$ and $u$. In a similar fashion it can be verified for the other transition rules of $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ that $S$ is a stratification. Hence, the TSS for $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ is complete.

## 4 Conservative Extension

Over and over again, process calculi such as CCS [158], CSP [187], and ACP [29] have been extended with new features, and the original TSSs, which provide the semantics for these process algebras, were extended with transition rules to describe these features; see, e.g., [28] for a systematic approach. A question that arises naturally is whether or not the LTSs associated with the original and with the extended TSS contain the same transitions $t \xrightarrow{a} t'$ and $tP$ for closed terms $t$ in the original domain. Usually it is desirable that an extension is operationally conservative, meaning that the provable transitions for an original term are the same both in the original and in the extended TSS.

Groote and Vaandrager [111, Thm. 7.6] proposed syntactic restrictions on a TSS, which automatically yield that an extension of this TSS with transition rules that contain fresh function symbols in their sources is operationally conservative (cf. the notion of a disjoint extension from [8] in Def. 5.32). Bol and Groote [57, 108] supplied this conservative extension format with negative premises. Verhoef [227] showed that, under certain conditions, a transition rule in the extension can be allowed to have an original term as its source. D'Argenio and Verhoef [69, 70] formulated a generalization in the context of inequational specifications. Fokkink and Verhoef [87] relaxed the syntactic restrictions on the original TSS, and lifted the operational conservative extension result to higher-order languages (see Sect. 6.4).

Operational conservative extension seems such a natural notion that in the literature this property is often a hidden assumption: its formulation and proof are omitted without justification. For example, this happens in the design of process algebras, and in applications of a strategy to prove $\omega$-completeness mentioned in Sect. 4.3.3. Paying attention to operational conservative extension not only leads to more accurate contemplations on concurrency theory, but is also beneficial in other respects. Namely, operational conservative extension can be applied to obtain results in process algebra that are much harder to obtain using more classical term rewriting approaches or customized techniques.

The organization of this section is as follows. Sect. 4.1 presents syntactic constraints to ensure that an extension of a TSS is operationally conservative. Sect. 4.2 studies the relation between the three-valued stable models of a TSS and of its operational conservative extension. Sects. 4.3 and 4.4 show how operational conservative extension can be applied to derive useful properties concerning axiomatizations and term rewriting systems.

## 4.1 Operational Conservative Extension

Often one wants to add new operators and rules to a given TSS. Therefore, a natural operation on TSSs is to take their componentwise union. The following definition stems from [111].

**Definition 4.1 (Sum of TSSs)** *Let $T_0$ and $T_1$ be TSSs whose signatures $\Sigma_0$ and $\Sigma_1$ agree on the arity of the function symbols in their intersection. We write $\Sigma_0 \oplus \Sigma_1$ for the union of $\Sigma_0$ and $\Sigma_1$.*

*The* sum *of $T_0$ and $T_1$, notation $T_0 \oplus T_1$, is the TSS over signature $\Sigma_0 \oplus \Sigma_1$ containing the rules in $T_0$ and $T_1$.*

An *operational conservative extension* requires that an original TSS and its extension prove exactly the same closed transition rules that have only negative premises and an original closed term as their source. This notion of an operational conservative extension is related to an equivalence notion for TSSs in [85, 100] (see also Thm. 5.6): two TSSs are equivalent if they prove exactly the same closed transition rules that have only negative premises. Such a definition is inspired by the notion of a well-supported proof in Def. 3.9.

**Definition 4.2 (Operational conservative extension)** *A TSS $T_0 \oplus T_1$ is an* operational conservative extension *of TSS $T_0$ if for each closed transition rule $N/\alpha$ such that:*

- *$N$ contains only negative literals;*
- *the left-hand side of $\alpha$ is in $\mathrm{T}(\Sigma_0)$;*
- *$T_0 \oplus T_1 \vdash N/\alpha$;*

*we have that $T_0 \vdash N/\alpha$.*

We proceed to define the notion of a *source-dependent* variable [87, 98], which will be an important ingredient of a rule format to ensure that an extension of a TSS is operationally conservative (see Thm. 4.4). In order to conclude that an extended TSS is operationally conservative over the original TSS, we need to know that the variables in the original transition rules are source-dependent. In the literature this criterion is sometimes neglected. For example, in [167] an extended TSS is considered in which each transition rule in the extension contains a fresh operator in its source, and from this fact alone it is concluded that the extension is operationally conservative. In general, however, this characteristic is not sufficient, as is shown in the next example.

**Example:** Let $a$ and $b$ be constants. Consider the TSS over signature $\{a\}$ that consists of the transition rule $xP/aP$. Extend this TSS with the TSS over signature $\{b\}$ that consists of the transition rule $\varnothing/bP$, which contains the fresh constant $b$ in its source. The transition $aP$ can be proven in the extended TSS, but not in the original one, so this extension is not operationally conservative. $\square$

**Definition 4.3 (Source-dependency)** *The* source-dependent *variables in a transition rule $\rho$ are defined inductively as follows:*

- *all variables in the source of $\rho$ are source-dependent;*
- *if $t \xrightarrow{a} t'$ is a premise of $\rho$ and all variables in $t$ are source-dependent, then all variables in $t'$ are source-dependent.*

*A transition rule is* source-dependent *if all its variables are.*

Note that the transition rule $xP/aP$ from the example above is not source-dependent, because its variable $x$ is not.

Thm. 4.4 below, which stems from [87], formulates sufficient criteria for a TSS $T_0 \oplus T_1$ to be an operational conservative extension of TSS $T_0$. We say that a term in $\mathbb{T}(\Sigma_0 \oplus \Sigma_1)$ is *fresh* if it contains a function symbol from $\Sigma_1 \backslash \Sigma_0$. Similarly, an action or predicate symbol in $T_1$ is *fresh* if it does not occur in $T_0$.

**Theorem 4.4** *Let $T_0$ and $T_1$ be TSSs over signatures $\Sigma_0$ and $\Sigma_1$, respectively. Under the following conditions, $T_0 \oplus T_1$ is an operational conservative extension of $T_0$.*

1. *Each $\rho \in T_0$ is source-dependent.*
2. *For each $\rho \in T_1$,*
    - *either the source of $\rho$ is fresh,*

- *or $\rho$ has a premise of the form $t \xrightarrow{a} t'$ or $tP$, where:*
  - $t \in \mathbb{T}(\Sigma_0)$*;*
  - *all variables in $t$ occur in the source of $\rho$;*
  - *$t'$, $a$, or $P$ is fresh.*

We apply Thm. 4.4 to our running examples from Sect. 2.6.

**BPA with Empty Process** The transition rules for $\mathrm{BPA}_\epsilon$ are all source-dependent. For example, consider the third transition rule for sequential composition in Table 1:

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

The variables $x$ and $y$ are source-dependent, because they occur in the source. Moreover, since $x$ is source-dependent, the premise $x \xrightarrow{a} x'$ ensures that $x'$ is source-dependent. Since the three variables $x$, $x'$, and $y$ in this transition rule are source-dependent, the transition rule is source-dependent.

**BPA with Empty Process and Silent Step** The process algebra $\mathrm{BPA}_{\epsilon\tau}$ is obtained by extending the syntax of $\mathrm{BPA}_\epsilon$ with a fresh constant $\tau$ called the silent step (see Sect. 5.5 for details on the intuition behind this constant). The TSS for $\mathrm{BPA}_{\epsilon\tau}$ is the TSS for $\mathrm{BPA}_\epsilon$ in Table 1, with the proviso that $a$ ranges over $\mathsf{Act} \cup \{\tau\}$. We make the following observations concerning the extra transition rules in the TSS for $\mathrm{BPA}_{\epsilon\tau}$:

- the source of the transition rule $\tau \xrightarrow{\tau} \epsilon$ for the silent step contains the fresh constant $\tau$;
- each transition rule for alternative or sequential composition with $\tau$-transitions, such as
$$\frac{x \xrightarrow{\tau} x'}{x + y \xrightarrow{\tau} x'}$$
contains a premise with the fresh relation symbol $\xrightarrow{\tau}$ and with as left-hand side a variable from the source.

Hence, since the transition rules for $\mathrm{BPA}_\epsilon$ are source-dependent, Thm. 4.4 implies that $\mathrm{BPA}_{\epsilon\tau}$ is an operational conservative extension of $\mathrm{BPA}_\epsilon$.

**Priorities** The two transition rules for the priority operator in Table 2 contain the fresh function symbol $\theta$ in their sources. Hence, since the transition rules for $\mathrm{BPA}_\epsilon$ are source-dependent, Thm. 4.4 implies that $\mathrm{BPA}_{\epsilon\theta}$ is an operational conservative extension of $\mathrm{BPA}_\epsilon$.

**Discrete Time** As for the TSS for $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ in Table 3, we make the following observations. First, the transition rule for the delay operator contains the fresh operator $\sigma_d$ in its source. Second, the transition rules for sequential composition and the three transition rules for alternative composition (which do not have a fresh operator in their sources) all contain the premise $x \stackrel{\sigma}{\to} x'$ or $y \stackrel{\sigma}{\to} y'$, where the relation symbol $\stackrel{\sigma}{\to}$ is fresh and the variable on the left-hand side occurs in the source. Hence, since the transition rules for $\mathrm{BPA}_\epsilon$ are source-dependent, Thm. 4.4 implies that $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ is an operational conservative extension of $\mathrm{BPA}_\epsilon$.

### 4.2 Implications for Three-Valued Stable Models

In [87] it was noted that the operational conservative extension notion as formulated in Def. 4.2 implies a conservativity property for three-valued stable models (cf. Def. 3.6). If an extended TSS is operationally conservative over the original TSS, in the sense of Def. 4.2, and if a three-valued stable model of the extended TSS is restricted to those transitions that have an original term as left-hand side, then the result is a three-valued stable model of the original TSS.

**Proposition 4.5** *Let $T_0 \oplus T_1$ be an operational conservative extension of $T_0$. If $\langle C, U\rangle$ is a three-valued stable model of $T_0 \oplus T_1$, then*

$$\begin{aligned} C' &\triangleq \{\alpha \in C \mid \textit{the left-hand side of } \alpha \textit{ is in } \mathrm{T}(\Sigma_0)\} \\ U' &\triangleq \{\alpha \in U \mid \textit{the left-hand side of } \alpha \textit{ is in } \mathrm{T}(\Sigma_0)\} \end{aligned}$$

*is a three-valued stable model of $T_0$.*

The converse of Prop. 4.5 also holds, in the following sense. If an extended TSS is operationally conservative over the original TSS, then each three-valued stable model of the original TSS can be obtained by restricting some three-valued stable model of the extended TSS to those transitions that have an original term as left-hand side.

**Proposition 4.6** *Let $T_0 \oplus T_1$ be an operational conservative extension of $T_0$. If $\langle C, U\rangle$ is a three-valued stable model of $T_0$, then there exists a three-valued stable model $\langle C', U'\rangle$ of $T_0 \oplus T_1$ such that*

$$\begin{array}{rcl} C & \triangleq & \{\alpha \in C' \mid \textit{the left-hand side of } \alpha \textit{ is in } \mathrm{T}(\Sigma_0)\} \\ U & \triangleq & \{\alpha \in U' \mid \textit{the left-hand side of } \alpha \textit{ is in } \mathrm{T}(\Sigma_0)\} \ . \end{array}$$

**Corollary 4.7** *Let $T_0 \oplus T_1$ be an operational conservative extension of $T_0$. If $\langle C, U\rangle$ is the least three-valued stable model of $T_0 \oplus T_1$, then*

$$\begin{array}{rcl} C' & \triangleq & \{\alpha \in C \mid \textit{the left-hand side of } \alpha \textit{ is in } \mathrm{T}(\Sigma_0)\} \\ U' & \triangleq & \{\alpha \in U \mid \textit{the left-hand side of } \alpha \textit{ is in } \mathrm{T}(\Sigma_0)\} \end{array}$$

*is the least three-valued stable model of $T_0$.*

It is easy to see that Prop. 4.5 also holds for stable models (cf. Def. 3.5). The following example, however, shows that Prop. 4.6 does not hold for stable models.

**Example:** Let $T_0$ be the empty TSS. Obviously, the empty LTS is a stable model of $T_0$. Let $a$ be a constant, and let $T_1$ consist of the single transition rule $a\neg P/aP$. According to Thm. 4.4, $T_0 \oplus T_1$ is an operational conservative extension of $T_0$. However, $T_0 \oplus T_1$ does not have a stable model (but only the three-valued stable model $\langle \varnothing, \{aP\}\rangle$). □

## 4.3 Applications to Axiomatizations

This section discusses how operational conservative extension can be used to derive that an extension of an axiomatization is so-called axiomatically conservative, or that an axiomatization is complete or $\omega$-complete with respect to some behavioural equivalence.

### 4.3.1 Axiomatic Conservative Extension

**Definition 4.8 (Axiomatization)** *A* (conditional) axiomatization *over a signature $\Sigma$ consists of a set of (conditional) equations, called* axioms, *of the form $t_0 = u_0 \Leftarrow t_1 = u_1, \ldots, t_n = u_n$ with $t_i, u_i \in \mathbb{T}(\Sigma)$ for $i = 0, \ldots, n$.*

An axiomatization gives rise to a binary equality relation $=$ on $\mathbb{T}(\Sigma)$ thus:

- if $t_0 = u_0 \Leftarrow t_1 = u_1, \ldots, t_n = u_n$ is an axiom, and $\sigma$ a substitution such that $\sigma(t_i) = \sigma(u_i)$ for $i = 1, \ldots, n$, then $\sigma(t_0) = \sigma(u_0)$;

- the relation $=$ is closed under reflexivity, symmetry, and transitivity;
- if $f$ is a function symbol and $u = u'$, then

$$f(t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_{ar(f)}) = f(t_1, \ldots, t_{i-1}, u', t_{i+1}, \ldots, t_{ar(f)}).$$

**Definition 4.9 (Soundness and completeness)** *Assume an axiomatization $\mathcal{E}$, together with an equivalence relation $\sim$ on $\mathrm{T}(\Sigma)$.*

1. *$\mathcal{E}$ is* sound *modulo $\sim$ iff $t = u$ implies $t \sim u$ for all $t, u \in \mathrm{T}(\Sigma)$.*
2. *$\mathcal{E}$ is* complete *modulo $\sim$ iff $t \sim u$ implies $t = u$ for all $t, u \in \mathrm{T}(\Sigma)$.*

Note that the above definitions of soundness and completeness, albeit standard in the literature on process algebras, are weaker than the classic ones in logic and universal algebra, where they are required to apply to arbitrary *open* expressions.

**Definition 4.10 (Axiomatic conservative extension)** *Let $\mathcal{E}_0$ and $\mathcal{E}_1$ be axiomatizations over signatures $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively. Their union $\mathcal{E}_0 \cup \mathcal{E}_1$ is an* axiomatic conservative extension *of $\mathcal{E}_0$ if every equality $t = u$ with $t, u \in \mathrm{T}(\Sigma_0)$ that can be derived from $\mathcal{E}_0 \cup \mathcal{E}_1$ can also be derived from $\mathcal{E}_0$.*

The next theorem from [227] can be used to derive that an extension of an axiomatization is axiomatically conservative.

**Theorem 4.11** *Let $\sim$ be an equivalence relation on $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$. Assume axiomatizations $\mathcal{E}_0$ and $\mathcal{E}_1$ over $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively, such that:*

1. *$\mathcal{E}_0 \cup \mathcal{E}_1$ is sound over $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ modulo $\sim$;*
2. *$\mathcal{E}_0$ is complete over $\mathrm{T}(\Sigma_0)$ modulo $\sim$.*

*Then $\mathcal{E}_0 \cup \mathcal{E}_1$ is an axiomatic conservative extension of $\mathcal{E}_0$.*

The idea behind Thm. 4.11 is as follows. Suppose that $t = u$ can be derived from $\mathcal{E}_0 \cup \mathcal{E}_1$ for $t, u \in \mathrm{T}(\Sigma_0)$. Soundness of $\mathcal{E}_0 \cup \mathcal{E}_1$ (requirement 1) yields $t \sim u$. Hence, completeness of $\mathcal{E}_0$ (requirement 2) yields that $t = u$ can be derived from $\mathcal{E}_0$.

Thm. 4.11 is particularly helpful in the case of an operational conservative extension of a TSS. Namely, assume TSSs $T_0$ and $T_1$ over signatures $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively, where $T_0 \oplus T_1$ is an operational conservative extension of $T_0$. Moreover, let $\sim$ be an equivalence relation on states in LTSs. Since the states in the LTSs associated with $T_0$ and $T_0 \oplus T_1$ are closed terms, the equivalence relation $\sim$ carries over to $\mathrm{T}(\Sigma_0)$ and $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$, respectively. Owing to operational conservativity, the equivalence relation $\sim$ on $\mathrm{T}(\Sigma_0)$ as induced by $T_0$ agrees with this equivalence relation on $\mathrm{T}(\Sigma_0)$ as induced by $T_0 \oplus T_1$. Applications of Thm. 4.11 in process algebra, in the presence of an operational conservative extension of a TSS, are abundant in the literature; we give a typical example.

**Example:** Using Thm. 4.4 it is easily seen that the process algebra $\mathrm{ACP}_\theta$ [24] is an operational conservative extension of ACP. Baeten, Bergstra, and Klop presented in *op. cit.* an axiomatization $\mathcal{E}_0$ that is complete over ACP modulo bisimulation equivalence, and an axiomatization $\mathcal{E}_0 \cup \mathcal{E}_1$ that is sound over $\mathrm{ACP}_\theta$ modulo bisimulation equivalence. Hence, Thm. 4.11 says that $\mathcal{E}_0 \cup \mathcal{E}_1$ is an axiomatic conservative extension of $\mathcal{E}_0$. (In [24], fifteen pages were needed to prove this fact for the more general case of open terms, by means of a term rewriting analysis.) $\square$

### 4.3.2 Completeness of Axiomatizations

The next theorem from [227] can be used to derive that an axiomatization is complete.

**Theorem 4.12** *Let $\sim$ be an equivalence relation on $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$. Assume axiomatizations $\mathcal{E}_0$ and $\mathcal{E}_1$ over $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively, such that:*

1. *$\mathcal{E}_0 \cup \mathcal{E}_1$ is sound over $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ modulo $\sim$;*
2. *$\mathcal{E}_0$ is complete over $\mathrm{T}(\Sigma_0)$ modulo $\sim$;*
3. *for each $t \in \mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ there is a $t' \in \mathrm{T}(\Sigma_0)$ such that $t = t'$ can be derived from $\mathcal{E}_0 \cup \mathcal{E}_1$.*

*Then $\mathcal{E}_0 \cup \mathcal{E}_1$ is complete over $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ modulo $\sim$.*

The idea behind Thm. 4.12 is as follows. Let $t, u \in \mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ with $t \sim u$. There exist terms $t', u' \in \mathrm{T}(\Sigma_0)$ such that $\mathcal{E}_0 \cup \mathcal{E}_1$ proves $t = t'$ and $u = u'$ (requirement 3). Soundness of $\mathcal{E}_0 \cup \mathcal{E}_1$ (requirement 1) yields $t \sim t'$ and $u \sim u'$, which together with $t \sim u$ implies $t' \sim u'$. Finally, owing to completeness of $\mathcal{E}_0$ over $\mathrm{T}(\Sigma_0)$ (requirement 2), we may derive $t' = u'$, and thus $t = t' = u' = u$.

Similar to Thm. 4.11, Thm. 4.12 is particularly helpful in the case of an operational conservative extension of a TSS. In order to clarify the link between Thm. 4.12 and operational conservative extensions, we reiterate the following observation from Sect. 4.3.1. Assume TSSs $T_0$ and $T_1$ over signatures $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively, where $T_0 \oplus T_1$ is an operational conservative extension of $T_0$. Moreover, let $\sim$ be an equivalence relation on states in LTSs. Since the states in the LTSs associated with $T_0$ and $T_0 \oplus T_1$ are closed terms, the equivalence relation $\sim$ carries over to $\mathrm{T}(\Sigma_0)$ and $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$, respectively. Owing to operational conservativity, the equivalence relation $\sim$ on $\mathrm{T}(\Sigma_0)$ as induced by $T_0$ agrees with this equivalence relation on $\mathrm{T}(\Sigma_0)$ as induced by $T_0 \oplus T_1$. Applications of Thm. 4.12 in process algebra, in the presence of an operational conservative extension of a TSS, are abundant in the literature; we give a typical example.

**Example:** Using Thm. 4.4 it is easily seen that the process algebra ACP [37] is an operational conservative extension of $\mathrm{BPA}_\delta$. Bergstra and Klop presented in *op. cit.* an axiomatization $\mathcal{E}_0$ that is complete over $\mathrm{BPA}_\delta$ modulo bisimulation equivalence, and an axiomatization $\mathcal{E}_0 \cup \mathcal{E}_1$ that is sound over ACP modulo bisimulation equivalence, and that satisfies requirement 3 above. Hence, Thm. 4.12 says that $\mathcal{E}_0 \cup \mathcal{E}_1$ is complete over ACP modulo bisimulation equivalence. $\square$

For the precise proofs of Thm. 4.11 and Thm. 4.12, and for more detailed information such as generalizations of these results to axiomatizations based on inequalities, the reader is referred to [69, 70, 227].

#### 4.3.3 $\omega$-Completeness of Axiomatizations

**Definition 4.13 ($\omega$-completeness)** *An axiomatization $\mathcal{E}$ over a signature $\Sigma$ is* $\omega$-complete *if an equation $t = u$ with $t, u \in \mathbb{T}(\Sigma)$ can be derived from $\mathcal{E}$ whenever $\sigma(t) = \sigma(u)$ can be derived from $\mathcal{E}$ for all closed substitutions $\sigma$.*

Milner [159] introduced a technique to derive $\omega$-completeness of an axiomatization using SOS. The idea is to give a semantics to open (as opposed to closed) terms; in particular, variables need to be incorporated in the transition rules. See, e.g., [9, 97] for further applications of this technique in the realm of process algebra.

The next theorem can be used to derive that an axiomatization is $\omega$-complete.

**Theorem 4.14** *Let $\sim$ be an equivalence relation on $\mathbb{T}(\Sigma)$. Suppose that for all $t, u \in \mathbb{T}(\Sigma)$, $t \sim u$ whenever $\sigma(t) \sim \sigma(u)$ for all closed substitutions $\sigma$. If $\mathcal{E}$ is an axiomatization over $\Sigma$ such that*

1. *$\mathcal{E}$ is sound over $\mathrm{T}(\Sigma)$ modulo $\sim$, and*

2. *$\mathcal{E}$ is complete over $\mathbb{T}(\Sigma)$ modulo $\sim$,*

*then $\mathcal{E}$ is $\omega$-complete.*

The idea behind Thm. 4.14 is as follows. Let $t, u \in \mathbb{T}(\Sigma)$ and suppose that $\sigma(t) = \sigma(u)$ can be derived from $\mathcal{E}$ for all closed substitutions $\sigma$. Soundness of $\mathcal{E}$ over $\mathrm{T}(\Sigma)$ modulo $\sim$ (requirement 1) yields $\sigma(t) \sim \sigma(u)$ for all closed substitutions $\sigma$, so $t \sim u$. Then completeness of $\mathcal{E}$ over $\mathbb{T}(\Sigma)$ modulo $\sim$ (requirement 2) yields that $t = u$ can be derived from $\mathcal{E}$.

Thm. 4.14 is particularly helpful in the case of an operational conservative extension of a TSS. Namely, assume a TSS $T_0$ over a signature $\Sigma$, and let $T_0$ be extended with a TSS $T_1$ that provides semantics to variables; thus, $T_0 \oplus T_1$ gives semantics to open terms in $\mathbb{T}(\Sigma)$. Suppose that $T_0 \oplus T_1$ is an operational conservative extension of $T_0$. Moreover, let $\sim$ be an equivalence relation on states in LTSs. Since the states in the LTSs associated with $T_0$ and $T_0 \oplus T_1$ are closed and open terms, respectively, the equivalence relation $\sim$ carries over to $\mathrm{T}(\Sigma)$ and $\mathbb{T}(\Sigma)$. Owing to operational conservativity, the equivalence relation $\sim$ on $\mathrm{T}(\Sigma)$ as induced by $T_0$ agrees with this equivalence relation on $\mathrm{T}(\Sigma)$ as induced by $T_0 \oplus T_1$. Applications of Thm. 4.14 in process algebra are abundant in the literature; we give a typical example.

**Example:** Extend the TSS for $\mathrm{BPA}_\epsilon$ in Table 1 by letting the symbol $a$ range not only over the set $\mathsf{Act}$ of actions, but also over the set $\mathsf{Var}$ of variables. In a sense this means that variables are considered to be constants. This extension is operationally conservative, which follows from Thm. 4.4 by the following facts:

- the transition rules for $\mathrm{BPA}_\epsilon$ are source-dependent;
- the sources of transition rules $z \xrightarrow{z} \epsilon$ for variables $z$ are fresh;
- each transition rule for alternative or sequential composition with $z$-transitions, such as
$$\frac{x \xrightarrow{z} x'}{x + y \xrightarrow{z} x'}$$
contains a premise with the fresh relation symbol $\xrightarrow{z}$ and as left-hand side a variable from the source.

Furthermore, the following properties can be derived for the axiomatization $\mathcal{E}$ of BPA$_\epsilon$ in [229]:

1. $\mathcal{E}$ is sound over BPA$_\epsilon$ modulo bisimulation equivalence;

2. open terms $t$ and $u$ in BPA$_\epsilon$ are bisimilar whenever $\sigma(t)$ and $\sigma(u)$ are bisimilar for all closed substitutions $\sigma$;

3. $\mathcal{E}$ is complete over the open terms in BPA$_\epsilon$ modulo bisimulation equivalence.

So Thm. 4.14 implies that $\mathcal{E}$ is $\omega$-complete over BPA$_\epsilon$ modulo bisimulation equivalence. $\square$

## 4.4 Applications to Rewriting

This section discusses how operational conservative extension can be used to derive that an extension of a conditional term rewriting system is so-called rewrite conservative, or that a conditional term rewriting system is ground confluent.

### 4.4.1 Rewrite Conservative Extension

**Definition 4.15 (Conditional term rewriting system)** *Assume a signature* $\Sigma$. *A* conditional term rewriting system *(CTRS) [17, 38] over* $\Sigma$ *consists of a set of* rewrite rules

$$t_0 \to u_0 \Leftarrow t_1 \to^* u_1, \ldots, t_n \to^* u_n$$

*with* $t_i, u_i \in \mathbb{T}(\Sigma)$ *for* $i = 0, \ldots, n$.

Intuitively, a rewrite rule is a directed axiom that can only be applied from left to right. A CTRS induces a binary rewrite relation $\to^*$ on terms, similar to the way that an axiomatization induces an equality relation on terms (the only difference is that the rewrite relation is not closed under symmetry), thus:

- if $t_0 \to u_0 \Leftarrow t_1 \to^* u_1, \ldots, t_n \to^* u_n$ is a rewrite rule, and $\sigma$ a substitution such that $\sigma(t_i) \to^* \sigma(u_i)$ for $i = 1, \ldots, n$, then $\sigma(t_0) \to^* \sigma(u_0)$;

- the relation $\to^*$ is closed under reflexivity and transitivity;

- if $f$ is a function symbol and $u \to^* u'$, then

$$f(t_1,\ldots,t_{i-1},u,t_{i+1},\ldots,t_{ar(f)}) \to^* f(t_1,\ldots,t_{i-1},u',t_{i+1},\ldots,t_{ar(f)}).$$

**Definition 4.16 (Rewrite conservative extension)** *Let $R_0$ and $R_1$ be CTRSs over signatures $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively. Their union $R_0 \oplus R_1$ is a* rewrite conservative extension *of $R_0$ if every rewrite relation $t \to^* u$ with $t \in \mathrm{T}(\Sigma_0)$ that can be derived from $R_0 \oplus R_1$ can also be derived from $R_0$.*

The conservative extension theorem for TSSs, Thm. 4.4, applies to CTRSs just as well; see [88] for more details, and for applications of this result in the realm of software renovation (see, e.g., [222]). Note that the definition of source-dependent variables in transition rules, Def. 4.3, also applies to rewrite rules (where, in a rewrite rule $t_0 \to u_0 \Leftarrow t_1 \to^* u_1,\ldots,t_n \to^* u_n$, the expression $t_0 \to u_0$ is the conclusion and the $t_i \to^* u_i$ for $i = 1,\ldots,n$ are the premises).

**Theorem 4.17** *Let $R_0$ and $R_1$ be CTRSs over signatures $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively. Under the following conditions, $R_0 \oplus R_1$ is a rewrite conservative extension of $R_0$.*

1. *Each $\rho \in R_0$ is source-dependent.*
2. *For each $\rho \in T_1$,*
   - *either the source of $\rho$ is fresh,*
   - *or $\rho$ has a premise of the form $t \to t'$ where:*
     - *$t \in \mathbb{T}(\Sigma_0)$;*
     - *all variables in $t$ occur in the source of $\rho$;*
     - *$t'$ is fresh.*

### 4.4.2 Ground Confluence of CTRSs

A CTRS is *ground confluent* if for all $t, t_0, t_1 \in \mathrm{T}(\Sigma)$ with $t \to^* t_0$ and $t \to^* t_1$ there is a $u \in \mathrm{T}(\Sigma)$ with $t_0 \to^* u$ and $t_1 \to^* u$. Ground confluence is an important property, for instance, to prove that an axiomatization is complete modulo some behavioural equivalence relation.

The next theorem from [227] can be used to derive that a CTRS is ground confluent. We say that a CTRS $R$ is *sound* modulo an equivalence relation $\sim$ on $\mathrm{T}(\Sigma)$ if $t \to^* u$ implies $t \sim u$ for all $t, u \in \mathrm{T}(\Sigma)$.

**Theorem 4.18** *Let $\sim$ be an equivalence relation on $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$. Assume CTRSs $R_0$ and $R_1$ over $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively, such that:*

1. *$R_0 \oplus R_1$ is sound over $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ modulo $\sim$;*
2. *if $t, t' \in \mathrm{T}(\Sigma_0)$ with $t \sim t'$, then there is a $u \in \mathrm{T}(\Sigma_0)$ such that $t \to^* u$ and $t' \to^* u$ can be derived from $R_0$;*
3. *for each $t \in \mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ there is a $t' \in \mathrm{T}(\Sigma_0)$ such that $t \to^* t'$ can be derived from $R_0 \oplus R_1$.*

*Then $R_0 \oplus R_1$ is ground confluent over $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$.*

The idea behind Thm. 4.18 is as follows. Let $t \in \mathrm{T}(\Sigma_0 \oplus \Sigma_1)$ such that $t \to^* t_0$ and $t \to^* t_1$ can be derived from $R_0 \oplus R_1$. There exist $t_0', t_1' \in \mathrm{T}(\Sigma_0)$ such that $t_0 \to^* t_0'$ and $t_1 \to^* t_1'$ can be derived from $R_0 \oplus R_1$ (requirement 3). Soundness of $R_0 \oplus R_1$ (requirement 1) yields $t \sim t_0 \sim t_0'$ and $t \sim t_1 \sim t_1'$, so $t_0' \sim t_1'$. Then there exists a $u \in \mathrm{T}(\Sigma_0)$ such that $t_0' \to^* u$ and $t_1' \to^* u$ (requirement 2). Hence, $t_0 \to^* u$ and $t_1 \to^* u$.

Similar to Thm. 4.11 and Thm. 4.12, Thm. 4.18 is particularly helpful in the case of an operational conservative extension of a TSS. In order to clarify the link between Thm. 4.18 and operational conservative extensions, we reiterate the following observation from Sect. 4.3.1. Assume TSSs $T_0$ and $T_1$ over signatures $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively, where $T_0 \oplus T_1$ is an operational conservative extension of $T_0$. Moreover, let $\sim$ be an equivalence relation on states in LTSs. Since the states in the LTSs associated with $T_0$ and $T_0 \oplus T_1$ are closed terms, the equivalence relation $\sim$ carries over to $\mathrm{T}(\Sigma_0)$ and $\mathrm{T}(\Sigma_0 \oplus \Sigma_1)$, respectively. Owing to operational conservativity, the equivalence relation $\sim$ on $\mathrm{T}(\Sigma_0)$ as induced by $T_0$ agrees with this equivalence relation on $\mathrm{T}(\Sigma_0)$ as induced by $T_0 \oplus T_1$. Applications of Thm. 4.18, in the presence of an operational conservative extension of a TSS, are abundant in the literature; we give a typical example.

**Example:** Using Thm. 4.4 it is easily seen that the process algebra ACP [37] is an operational conservative extension of $\mathrm{BPA}_\delta$. Bergstra and Klop presented in *op. cit.* an (unconditional) CTRS $R_0 \oplus R_1$ for the process algebra ACP, which reduces each closed term in ACP to a closed term in $\mathrm{BPA}_\delta$. Moreover, $R_0 \oplus R_1$ is sound over ACP modulo bisimulation equivalence, and it is easily shown that $R_0$ can reduce bisimilar closed terms in $\mathrm{BPA}_\delta$ to the same closed term in $\mathrm{BPA}_\delta$. Hence, Thm. 4.11 says that $R_0 \oplus R_1$ is ground confluent. (In [37, p. 122], an analysis of about 400 cases was needed to prove this fact for the more general case of open terms.) □

# 5 Congruence Formats

The development of process theory in the 1980s led to several process description languages and a large body of results. As the field of process theory matured, it became apparent that many similar results proven for different languages in different papers in the research literature were, in fact, instances of more general theorems which are independent of the chosen process description language. This realization paved the way to the development of a meta-theory of process description languages in which one can prove theorems for whole classes of languages at the same time. Indeed, as it is the case in science and mathematics, when asked to produce one or more results, it is usual to generalize the problem, and to consider these results as specific instances of more general problems. In following such a more abstract approach to the development of our theories, we have two obligations:

1. to be very specific as to how we generalize, i.e., we have to choose the wider class of problems carefully and to define it explicitly, because our arguments must apply to the whole class;

2. to choose a generalization that is helpful to our purpose.

Process description languages are often equipped with an SOS. Thus, this way of giving operational semantics to terms has been a natural handle to establish results that hold for all process calculi whose transition rules fit a certain rule format, imposing syntactic constraints on the form of the allowed rules. Rule formats have proven to be suitable tools for the generalization of specific results in process theory. A central issue in the area of SOS is to define rule formats ensuring that a behavioural equivalence relation is a congruence, meaning that each function symbol respects this equivalence. This section presents an overview of the congruence formats for TSSs that have been studied in the literature, and hints at results that have been proven for them.

The most basic rule format to guarantee that bisimulation equivalence is a congruence is the *De Simone* format [201]. The *GSOS* format [54, 55] allows negative premises but no look-ahead, and the *tyft/tyxt* format [110, 111] allows look-ahead but no negative premises. The *positive GSOS* format is, so the speak, the greatest common divisor of the GSOS and the tyft/tyxt format. The *ntyft/ntyxt* format [108] extends the tyft/tyxt format with negative premises. Finally, the *path* format [27] generalizes the tyft/tyxt format with predicates, while the *panth* format [226, 228] extends

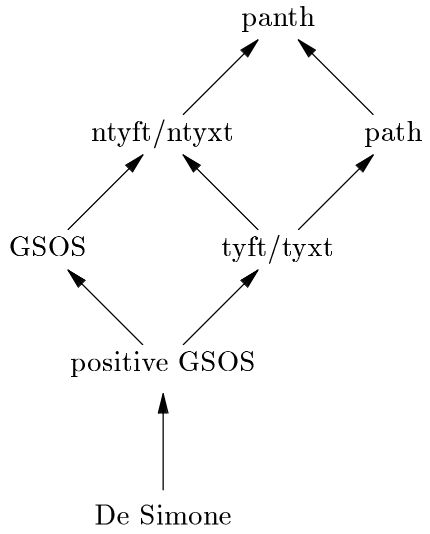


Figure 1: Lattice of Congruence Formats

the ntyft/ntyxt format with predicates. Figure 1 presents the lattice of congruence formats for bisimulation equivalence. An arrow from one rule format to another indicates that all transition rules in the first format are also in the second format. If there are no arrows connecting two rule formats, then they are (syntactically) incomparable.

If a TSS is both panth and (ws-)complete (in the sense of Def. 3.12), then bisimulation equivalence is a congruence with respect to all the function symbols in the signature. The panth format is the most general known syntactic format to guarantee that bisimulation equivalence is a congruence. However, more restrictive rule formats such as De Simone and GSOS guarantee other nice properties. Therefore, these rule formats are treated separately in this section.

For each TSS in panth format there exists an equivalent TSS in *ntree* format [85]. This result facilitates reasoning about the panth format, because it is often much easier to prove a theorem for TSSs in ntree format than for TSSs in panth format. For example, this is the case for the congruence theorem for bisimulation equivalence itself. Furthermore, the reduction of panth to ntree made it possible to remove the well-foundedness criterion on premises from an earlier version of the congruence theorem, owing to the fact that TSSs in ntree format satisfy this well-foundedness criterion by default; see [85].

For the sake of presentation, the lattice in Figure 1 focuses on the rule formats that are of practical importance; that is, we left out the ntree format and its derived (unnamed) formats that disallow predicates and/or negative premises. Other rule formats that are not mentioned in Figure 1 are those that deal with silent actions explicitly. Of particular interest here are the rule formats presented in, e.g., [51, 84, 215, 218].

The organization of this section is as follows. Sect. 5.1 presents the panth format, while Sect. 5.2 deals with the equally expressive ntree format. Sects. 5.3 and 5.4 study De Simone languages and GSOS languages, respectively. Sect. 5.5 introduces a congruence format in the presence of silent actions, while Sect. 5.6 presents congruence formats for a wide range of behavioural preorders. Finally, Sect. 5.7 studies the completed trace congruence induced by a number of congruence formats.

### 5.1 Panth Format

This section presents the panth format, and states a congruence theorem (cf. Def. 2.11) from [57, 111, 228] with respect to bisimulation equivalence (cf. Def. 2.3).

**Definition 5.1 (Panth format)** *A transition rule $\rho$ is in* panth *format if it satisfies the following three restrictions:*

1. *for each positive premise $t \xrightarrow{a} t'$ of $\rho$, the right-hand side $t'$ is a single variable;*
2. *the source of $\rho$ contains no more than one function symbol;*
3. *the variables that occur as right-hand sides of positive premises or in the source of $\rho$ are all distinct.*

*A TSS is in* panth *format if it consists of panth rules only.*

The three subformats path, ntyft/ntyxt, and tyft/tyxt of the panth format do not allow for negative premises and/or predicates.

**Definition 5.2 (Path, ntyft/ntyxt, and tyft/tyxt format)** *A TSS is in* path *format if it is in panth format and positive.*

*A TSS is in* ntyft/ntyxt *format if it is in panth format and its transition rules do not contain predicates.*

*A TSS is in* tyft/tyxt *format if it is both path and ntyft/ntyxt.*

*A TSS is in* tyft *format if it is tyft/tyxt and the source of each rule contains exactly one function symbol.*

**Theorem 5.3** *If a TSS is complete and panth, then bisimulation equivalence is a congruence with respect to the LTS associated with it.*

The interested reader is referred to [85, 111, 228] for a proof of Thm. 5.3. Groote and Vaandrager [111] presented a string of examples of TSSs which show that all syntactic requirements of the panth format are essential for the congruence result in Thm. 5.3. We give an example to show that the restriction in Thm. 5.3 to complete TSSs is essential. In particular, it cannot be relaxed to TSSs that have exactly one (not necessarily least) three-valued stable model that does not contain unknown transitions. The example is derived from [57, Ex. 8.12].

**Example:** Let the signature consist of constants $a$, $b$ and of a unary function symbol $f$. Moreover, let $P$, $Q_1$, and $Q_2$ be predicates. Consider the following TSS in panth format:

$$\frac{}{aP} \qquad \frac{}{bP} \qquad \frac{xP \quad f(x)\neg Q_1 \quad f(a)\neg Q_2}{f(x)Q_2} \qquad \frac{xP \quad f(x)\neg Q_2 \quad f(b)\neg Q_1}{f(x)Q_1}$$

Its least three-valued stable model contains the following unknown transitions: $f(a)Q_1$, $f(a)Q_2$, $f(b)Q_1$, and $f(b)Q_2$. So the TSS is not complete.

However, the TSS does have a three-valued stable model in which the set of unknown transitions is empty, and $aP$, $bP$, $f(a)Q_1$, and $f(b)Q_2$ are the true transitions. $a \underline{\leftrightarrow} b$ but $f(a) \not\underline{\leftrightarrow} f(b)$ with respect to this three-valued stable model. $\square$

van Oostrom and de Vink [169] generalized the tyft/tyxt format (so in the absence of predicates and negative premises) to the *stalk* format, by somewhat relaxing the constraint that sources of transition rules can contain no more than one function symbol. They proved that the congruence result in Thm. 5.3 holds for the stalk format.

We apply the congruence theorem for the panth format to the running examples from Sect. 2.6.

**Basic Process Algebra** The TSS for $\mathrm{BPA}_\epsilon$ in Table 1 is panth. For example, the transition rule

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

for sequential composition is panth, because the right-hand side of its premise is a single variable $x'$, its source contains only one function symbol (sequential composition), and the variables in the right-hand side of its premise ($x'$) and in its source ($x$ and $y$) are distinct. It is left to the reader to verify that the remaining transition rules in Table 1 are panth.

The TSS for $\mathrm{BPA}_\epsilon$ is positive, so it is complete. Since this TSS is both panth and complete, Thm. 5.3 says that bisimulation equivalence is a congruence with respect to $\mathrm{BPA}_\epsilon$.

**Priorities** It is not hard to check that the TSS for $\mathrm{BPA}_{\epsilon\theta}$ in Table 2 is panth. Furthermore, it was noted in Sect. 3.5 that the TSS for $\mathrm{BPA}_{\epsilon\theta}$ is complete. Hence, by Thm. 5.3, bisimulation equivalence is a congruence with respect to $\mathrm{BPA}_{\epsilon\theta}$.

**Discrete Time** It is not hard to check that the TSS for $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ in Table 3 is panth. Furthermore, it was noted in Sect. 3.5 that the TSS for $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ is complete. Hence, by Thm. 5.3, bisimulation equivalence is a congruence with respect to $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$.

### 5.2 Ntree Format

In this section we present a result from [82, 85] to the effect that for each TSS in panth format there exists an equivalent TSS in the more restrictive ntree format. The following terminology originates from [55, 111].

**Definition 5.4 (Variable dependency graph)** *The* variable dependency graph *of a set $S$ of premises is a directed graph, with the set of variables as vertices, and with as edges the set*

$$\{\langle x,y\rangle \mid \textit{there is a } t \xrightarrow{a} t' \textit{ in } S \textit{ such that } x \textit{ occurs in } t \textit{ and } y \textit{ in } t'\} \ .$$

*$S$ is* well-founded *if any backward chain of edges in its variable dependency graph is finite.*

*A transition rule is* pure *if its set of premises is well-founded and moreover each variable in the rule occurs in the source or as the right-hand side of a positive premise.*

Typical examples of sets of premises that are not well-founded are $\{y \xrightarrow{a} y\}$, $\{y_1 \xrightarrow{a} y_2,\ y_2 \xrightarrow{b} y_1\}$, and $\{y_{i+1} \xrightarrow{a} y_i \mid i \in \mathbb{N}\}$.

**Definition 5.5 (Ntree format)** *A transition rule $\rho$ is an* ntree *rule if it satisfies the following three criteria:*

1. *$\rho$ is panth;*

2. *$\rho$ is pure;*

3. *the left-hand sides of positive premises in $\rho$ are single variables.*

*A TSS is in* ntree *format if it consists of ntree rules only.*

For example, the TSSs for $\mathrm{BPA}_\epsilon$, $\mathrm{BPA}_{\epsilon\theta}$, and $\mathrm{BPA}_\epsilon^{\mathrm{dt}}$ from Sect. 2.6 are in ntree format. The following theorem originates from [85].

**Theorem 5.6** *For each TSS $T$ in panth format there exists a TSS $T'$ in ntree format such that for any closed transition rule $N/\alpha$ where $N$ contains only negative literals, $T \vdash N/\alpha \Leftrightarrow T' \vdash N/\alpha$.*

## 5.3 De Simone Format

A *De Simone language* [199, 201] consists of a signature together with a TSS whose transition rules are in *De Simone* format, extended with transition rules for recursion. Most process description languages encountered in the literature, including CCS [158], SCCS [156], CSP [187], ACP [29], and MEIJE [16], are De Simone languages.

### 5.3.1 De Simone Languages

For consistency with subsequent developments, amongst the various definitions of De Simone languages presented in the literature we adopt the one given by Vaandrager [221].

**Definition 5.7 (De Simone format)** *Let $\Sigma$ be a signature. A transition rule $\rho$ is in* De Simone *format if it has the form*

$$\frac{\{x_i \xrightarrow{a_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{a} t}$$

*where $I \subseteq \{1, \ldots, ar(f)\}$ and the variables $x_i$ and $y_i$ are all distinct and the only variables that occur in $\rho$. Moreover, the target $t \in \mathbb{T}(\Sigma)$ does not contain variables $x_i$ for $i \in I$ and has no multiple occurrences of variables. We say that $f$ is the* type *and $a$ the* action *of $\rho$.*

In conjunction with the signature $\Sigma$, we assume a countably infinite set of *recursion variables*, ranged over by $X, Y$. The recursive terms over $\Sigma$ are given by the BNF grammar

$$t ::= X \mid f(t_1, \ldots, t_{ar(f)}) \mid \mathrm{fix}(X = t)$$

where $X$ is any recursion variable, $f$ any function symbol in $\Sigma$, and fix a binding construct. The latter construct gives rise to the usual notions of free and bound recursion variables in recursive terms. We use $t[u/X]$ to denote the recursive term $t$ in which each occurrence of the recursion variable $X$ has been replaced by $u$ (after possibly renaming bound recursion variables in $t$). For every recursive term $\mathrm{fix}(X = t)$ and action $a$, we introduce a transition rule

$$\frac{t[\mathrm{fix}(X = t)/X] \xrightarrow{a} y}{\mathrm{fix}(X = t) \xrightarrow{a} y}$$

The reader is referred to Sect. 6 for a formal treatment of such transition rules that incorporate binding constructs. A *De Simone language* is a set of De Simone rules, extended with the transition rules above for recursion.

**Remark:** In De Simone's original definition for his rule format (cf. [201, Def. 1.9]), transition rules carried side conditions $Pr(a_1, \ldots, a_n)$ where $Pr$ is a predicate on actions (not to be confused with the predicates on states allowed in LTSs). No particular syntax for such predicates was considered in De Simone's work, but natural computability restrictions were imposed on the allowed sets of tuples. Most subsequent literature on rule formats for transition rules has abstracted from such predicates.

### 5.3.2 Expressiveness of De Simone Languages

The main original motivation for the development of the De Simone format was to gain insight into the expressive power of the process calculi SCCS and MEIJE, in the semantic realm of LTSs. In particular, what De Simone was aiming at in his seminal papers [199, 201] was an expressive completeness result for the aforementioned process calculi that would fully justify the choice of basic operators made by their developers. After all, the motivation for the study of foundational calculi for concurrency stems from Milner's idea that for a proper understanding of the basic issues in the behaviour of concurrent systems it would be helpful to look for a simple language "with as few operators or combinators as possible, each of which embodies some distinct and intuitive idea, and which together give completely general expressive power" [156, p. 264]. Before embarking on such an investigation, however, one has to choose an appropriate measure of expressiveness for a calculus. As argued by Vaandrager [221], there are at least three different ways in which a language can have completely general expressive power:

1. each Turing machine can be simulated in lock step;
2. each recursively enumerable LTS can be specified up to some notion of behavioural equivalence;
3. each operation in a "natural" class of operations is realizable by means of the primitive operations in the language up to some notion of behavioural equivalence.

Since most languages that have been proposed in the literature are completely expressive in the first sense, this criterion does not offer a useful means to classify the expressiveness of languages. (This is what Meyer [149] calls the "Turing tarpit".) The remaining two criteria have been investigated by De Simone in *op. cit.*, and, since then, by several other authors. Indeed, this kind of expressiveness results has, to the best of our knowledge, only been developed for De Simone languages, and similar investigations are lacking for more general rule formats. For this reason, the rest of this section is devoted to a brief review of such results. The other results that have been developed for this format are instances of theorems for the more general formats we survey.

The question of whether there exists a process description language which is completely expressive with respect to the collection of operations definable by means of De Simone rules has been addressed by several authors since De Simone's original work. In *op. cit.*, De Simone showed the following result.

**Theorem 5.8** *Let* $\mathsf{Act}$ *be a finite set of actions, and* $f$ *the type of only finitely many De Simone rules. Then* $f$ *can be expressed, up to bisimulation equivalence, in the calculi SCCS and* MEIJE.

As a corollary of this expressiveness result pertaining to the class of operators specifiable in SCCS and MEIJE, De Simone was able to prove that the calculi SCCS and MEIJE, and, *a fortiori*, reasonably expressive De Simone languages, can denote every recursively enumerable LTS up to graph isomorphism.

**Definition 5.9 (Properties of LTSs)** *An LTS is:*

- countably branching *if for every state $s$ there are at most countably many outgoing transitions $s \xrightarrow{a} s'$;*

- recursively enumerable *if there exists an algorithm enumerating all transitions $s \xrightarrow{a} s'$;*

- decidable *if there exists an algorithm that determines for every transition whether it is in the LTS;*

- computable *[25] if there exists an algorithm that computes for every state $s$ its complete finite list of outgoing transitions $s \xrightarrow{a} s'$;*

- primitive recursive *[180] if there is such an algorithm that is primitive recursive.*

Note that "computable" is a stronger requirement than "decidable and finitely branching" (cf. Def. 2.2). The following result is from [201, Thm. 3.2].

**Theorem 5.10** *Every recursively enumerable LTS can be realized by a recursive term in SCCS-*MEIJE *up to graph isomorphism.*

Several variations on Thm. 5.10 have been presented in the literature. Before stating some of these results, we need to introduce some preliminary definitions from [221].

**Definition 5.11 (Testing arguments)** *Assume a De Simone language. A function symbol $f$* tests *its $i$th argument if one of the De Simone rules has a source $f(x_1, \ldots, x_{ar(f)})$ and a premise $x_i \xrightarrow{a_i} y_i$.*

**Definition 5.12 (Guardedness)** *Assume a De Simone language. For recursive terms $t$, the sets $U(t)$ of* unguarded *recursion variables are defined thus:*

$$\begin{array}{rcl} U(X) & \triangleq & \{X\} \\ U(f(t_1, \ldots, t_{ar(f)})) & \triangleq & U(t_{i_1}) \cup \cdots \cup U(t_{i_k}) \quad \text{if } f \text{ tests arguments } i_1, \ldots, i_k \\ U(\mathrm{fix}(X = t)) & \triangleq & U(t) \backslash \{X\} \ . \end{array}$$

*A recursive term $t$ is* guarded *if for every subterm $\mathrm{fix}(X = t')$ of $t$ we have $X \notin U(t')$.*

In particular, if a function symbol $f$ tests none of its arguments, then owing to the second clause in Def. 5.12 no recursion variable is unguarded in a recursive term of the form $f(t_1, \ldots, t_{ar(f)})$.

Guarded recursive specifications in any De Simone language have unique solutions up to bisimulation equivalence. That is, let $t$ be a guarded recursive term with no other free recursion variables than $X$, and let $u$ and $u'$ be recursive terms without free recursion variables. If $u \underline{\leftrightarrow} t[u/X]$ and $u' \underline{\leftrightarrow} t[u'/X]$, then $u \underline{\leftrightarrow} u'$ (cf. [158, Sect. 4.5]).

It is interesting to remark here that the proof of Thm. 5.10 makes an essential use of unguarded recursive terms in SCCS and MEIJE (cf. [101, 221] for detailed comments on this issue). The use of infinite summations vis-à-vis that of unguarded recursive definitions is addressed in [200].

**Definition 5.13 (Trigger of a rule)** *Consider a De Simone rule $\rho$ with the term $f(x_i, \ldots, x_{ar(f)})$ as source and premises $\{x_i \xrightarrow{a_i} y_i \mid i \in I\}$. The* trigger *of $\rho$ is the tuple $(l_1, \ldots, l_{ar(f)})$, with $l_i \triangleq a_i$ if $i \in I$ and $l_i \triangleq *$ otherwise.*

We say that a signature $\Sigma$ is *decidable* if there is an algorithm that answers yes if the input is the encoding of a function symbol in $\Sigma$, and no for all other inputs (see, e.g., [186]). The following classification of De Simone languages, and the subsequent result, stem from [101].

**Definition 5.14 (Properties of De Simone languages)** *A De Simone language over a signature $\Sigma$ is:*

- recursively enumerable *if $\Sigma$ is decidable and the set of De Simone rules is recursively enumerable;*
- bounded *[221] if only guarded recursion is allowed and for each type and trigger the set of corresponding De Simone rules is finite;*
- effective *[221] if $\Sigma$ is decidable, only guarded recursion is allowed, and there exists a total recursive function associating with each type and trigger the finite set of corresponding De Simone rules;*
- coeffective *if $\Sigma$ is decidable, only guarded recursion is allowed, and there exists a total recursive function associating with each type, action, and target the finite set of corresponding De Simone rules;*
- primitive effective *if $\Sigma$ is primitive decidable, only guarded recursion is allowed, and there exists a primitive recursive function associating*

*with each type and trigger the finite set of corresponding De Simone rules;*

- primitive coeffective *if $\Sigma$ is primitive decidable, only guarded recursion is allowed, and there is a primitive recursive function giving for each type, action, and target the finite set of corresponding De Simone rules.*

**Proposition 5.15** *In a De Simone language with a property on the left, each recursive term gives rise to an LTS with the corresponding property on the right.*

| | |
|---|---|
| *countable* | *countably branching* |
| *recursively enumerable* | *recursively enumerable* |
| *bounded* | *finitely branching* |
| *effective* | *computable* |
| *coeffective* | *decidable* |
| *primitive effective* | *primitive recursive* |
| *primitive coeffective* | *primitive decidable* |

### 5.3.3 De Simone Languages and Process Algebras

The process algebra apr$\mathrm{ACP}_R$ [22] is a variant of ACP [36], containing *prefix multiplication* [158] in lieu of general sequential composition, where the first argument is restricted to actions, and a *relational renaming* operator $\rho_R$ for any binary relation $R \subseteq \mathsf{Act} \times \mathsf{Act}$. We use apr$\mathrm{ACP}_F$ for the sublanguage of apr$\mathrm{ACP}_R$ that only contains functional renamings like those considered in, e.g., CCS, ACP, and many other standard process calculi. (Exceptions are CSP [60], because of its inverse image operator, and the less standard calculus PC [221].) Suppose that $\mathsf{Act}$ contains actions $a_n$ and $b_n$ for $n \in \mathbb{N}$, and that there is an inert constant $\mathbf{0}$ for which there are no transition rules. Let $U$ denote the process that consists of the alternative composition of the terms $a_n{\cdot}b_n{\cdot}\mathbf{0}$ for $n \in \mathbb{N}$; i.e., $U$ can execute action $a_n$ followed by action $b_n$ to end up in $\mathbf{0}$. Adding this process as a special "constant" $U$ to the language apr$\mathrm{ACP}_F$ yields the language apr$\mathrm{ACP}_U$.

van Glabbeek [101] obtained several results concerning the expressibility of arbitrary De Simone languages in apr$\mathrm{ACP}_R$. In order to state these expressiveness results, more properties of De Simone languages need to be defined.

**Definition 5.16 (Operator dependency)** *Let $T$ be a TSS.* Operator dependency *is the smallest transitive binary relation between function symbols* *such that $f$ depends on $g$ if there is a transition rule in $T$ with type $f$ and with $g$ occurring in its target.*

**Definition 5.17 (Properties of De Simone languages II)** *A De Simone language is:*

- width-finitary *if for each type there are only finitely many corresponding targets (such that there is a De Simone rule with that type and target);*
- (primitive) width-effective *if there exists a (primitive) recursive function giving for each type the finite set of corresponding targets;*
- finitary *if*
  - *each function symbol depends on only finitely many function symbols, and*
  - *for each type there are only finitely many corresponding targets;*
- image-finite *if for each type and trigger the matching set of transition rules is finite;*
- functional *if there exists a finite upper bound on the number of transition rules with any given type and trigger.*

A language is finitary if the behaviour of each recursion-free term can be deduced by considering only finitely many transition rules. Thus a finitary De Simone language can be obtained as the combination of a number of De Simone languages with finitely many transition rules, each of which is trivially primitive width-effective. The following proposition originates from [101].

**Proposition 5.18** *Any De Simone language satisfying certain properties at the left side is expressible in $aprACP_R$ with the corresponding features at the right.*

| | |
|---|---|
| *finitary* | *with guarded recursion* |
| *image-finite* | *with image-finite renaming* |
| *functional* | *with functional renaming* |
| – | *recursively enumerable* |
| *primitive width-effective* | *primitive effective* |

In what follows we use two superscripts. The superscript r.e. denotes the recursively enumerable version of a language, i.e., its version that only includes a partial recursive communication function (see [101, Sect. 3.4] for details). The superscript p.e. denotes the primitive effective version of a language (see [101, Def. 15] for details). Prop. 5.18 establishes several expressiveness results. Since virtually all De Simone languages encountered in practice are finitary, the most significant of these results are the following.

1. Any finitary De Simone language is expressible in $\mathrm{aprACP}_R$ with guarded recursion.
2. Any finitary image-finite De Simone language is expressible in $\mathrm{aprACP}_R$ with guarded recursion and image-finite renamings.
3. Any finitary functional De Simone language is expressible in $\mathrm{aprACP}_F$ with guarded recursion.
4. Any finitary recursively enumerable De Simone language is expressible in $\mathrm{aprACP}_R^{\mathrm{r.e.}}$ with guarded recursion.
5. Any finitary recursively enumerable image-finite De Simone language is expressible in $\mathrm{aprACP}_R^{\mathrm{r.e.}}$ with guarded recursion and image-finite renamings.
6. Any finitary recursively enumerable functional De Simone language is expressible in $\mathrm{aprACP}_F^{\mathrm{r.e.}}$ with guarded recursion.
7. Any finitary primitive effective De Simone language is expressible in $\mathrm{aprACP}_R^{\mathrm{p.e.}}$ with guarded recursion.
8. Any finitary primitive effective functional De Simone language is expressible in $\mathrm{aprACP}_F^{\mathrm{p.e.}}$ with guarded recursion.

Result 4 in the above list generalizes the original theorem by De Simone, saying that any finitary recursively enumerable De Simone language with recursion is expressible in the recursively enumerable version of MEIJE with recursion. The generalization is that, under the assumption that the source languages have only guarded recursion, the target language (now $\mathrm{aprACP}_R$) can be required to use only guarded recursion as well.

Using the constant $U$ yields an even stronger result for recursively enumerable De Simone languages, viz. without requiring finitariness. This result from [101] has no effective counterpart.

**Theorem 5.19** *Assume a recursively enumerable De Simone language $T$. Every closed recursive term in the LTS associated with $T$ is bisimilar to a closed guarded recursive term in the LTS associated with $aprACP_U$.*

CSP (and SCCS and Meije) can specify infinitely branching processes. To do so in CSP, one uses the inverse image operation, which is similar to the relational renaming operator. The following result, to the effect that the process algebra CSP [60] is completely expressive with respect to operations definable using De Simone rules, stems from [131].

**Theorem 5.20** *Assume a De Simone language $T$. Every closed recursive term in the LTS associated with $T$ is bisimilar to a closed CSP term in the LTS associated with CSP.*

The interested reader will find further expressiveness results for variations on De Simone languages and several notions of "expressiveness" in, e.g., [20, 71, 77, 101, 172, 221].

We end this section with a congruence result for trace equivalence (see Def. 2.4) with respect to De Simone languages.

**Theorem 5.21** *If a TSS is in De Simone format, then trace equivalence is a congruence with respect to the LTS associated with it.*

## 5.4 GSOS Format

This section introduces one of the most thoroughly studied rule formats, viz. the GSOS format of Bloom, Istrail, and Meyer [55]. We present some of the many results that have been developed for this rule format, focusing on its sanity properties, and its connections with axiomatizations modulo bisimulation equivalence.

### 5.4.1 GSOS Languages

**Definition 5.22 (GSOS format)** *A transition rule $\rho$ is in* GSOS *format if it has the form*

$$\frac{\{x_i \stackrel{a_{ij}}{\to} y_{ij} \mid 1 \leq i \leq ar(f), 1 \leq j \leq m_i\} \cup \{x_i \stackrel{b_{ik}}{\not\to} \mid 1 \leq i \leq ar(f), 1 \leq k \leq n_i\}}{f(x_1, \ldots, x_{ar(f)}) \stackrel{c}{\to} t}$$

*where $m_i, n_i \in \mathbb{N}$, and the variables $x_i$ and $y_{ij}$ are all distinct and the only variables that occur in $\rho$.*

*A* (finitary) GSOS language *is a finite set of GSOS rules over a finite signature, and a finite set* $\mathsf{Act}$ *of actions.*

Every De Simone rule is also a GSOS rule. Unlike De Simone rules, however, GSOS rules allow for negative premises, as well as for multiple occurrences of variables in left-hand sides of premises and in the target.

An example of a GSOS rule with negative premises is the second transition rule for the priority operator $\theta$; see Table 2 in Sect. 2.6. For actions $a$ that are not a supremum with respect to the ordering on $\mathsf{Act}$, the second transition rule for $\theta$ contains negative premises. The priority operator cannot be expressed, up to bisimulation equivalence, using De Simone rules. Namely, $\theta$ does not preserve trace equivalence, unlike any operator expressible using De Simone rules (cf. Thm. 5.21). For example, $a{\cdot}(b+c)$ and $a{\cdot}b + a{\cdot}c$ are trace equivalent, but $\theta(a{\cdot}(b+c))$ and $\theta(a{\cdot}b + a{\cdot}c)$ are not, if $c > b$.

A notable example of a GSOS rule that uses the same variable in the left-hand side of a premise and in the target is a transition rule for the *binary Kleene star* [135]:

$$\frac{x \stackrel{a}{\to} x'}{x^*y \stackrel{a}{\to} x' \cdot (x^*y)}$$

This operator has been studied in the realm of process algebra in, e.g., [34, 89, 157] (see also elsewhere in this handbook).

Each GSOS language allows a stratification (cf. Def. 3.13), and is therefore complete (cf. Def. 3.12). LTSs associated with GSOS languages are computable (cf. Def. 5.9) and finitely branching (cf. Def. 2.2). By contrast, there exist TSSs in tyft/tyxt format (cf. Def. 5.2), consisting of only finitely many transition rules with only finitely many premises, such that the associated LTSs are neither computable (see [45, 71]) nor finitely branching (see [111, p. 258]). It is not straightforward to associate LTSs to GSOS languages with recursion (see, e.g., [55]). A solution for this problem, involving a special divergence predicate $\uparrow$, is discussed in Sect. 5.4.6.

### 5.4.2 Junk Rules

The definition of a GSOS language does not exclude *junk* rules, i.e., transition rules that support no transitions in the associated LTS. For example, the transition rule

$$\frac{x \stackrel{a}{\to} y \qquad x \stackrel{a}{\nrightarrow}}{f(x) \stackrel{a}{\to} f(y)}$$

has contradictory premises, so under no closed substitution do these premises both hold. Furthermore, the seemingly innocuous transition rule

$$\frac{x \stackrel{a}{\rightarrow} y}{f(x) \stackrel{b}{\rightarrow} f(y)}$$

does not support any transition if the associated LTS does not contain $a$-transitions. We present an unpublished result by Aceto, Bloom, and Vaandrager [6] to the effect that it is decidable whether a transition rule in a GSOS language is junk. This decision procedure for "rule junkness" allows a language designer to check whether or not any of the transition rules describing some language features are used. Since this result has not appeared in the literature before, we present its proof here.

**Theorem 5.23** *It is decidable whether a transition rule $\rho$ in a GSOS language $T$ is junk.*

**Proof:** Let $\Sigma$ denote the signature. It is not hard to determine which GSOS rules in $T$ are junk once we have computed the set

$$\mathsf{initials}(\mathrm{T}(\Sigma)) \quad = \quad \{\mathsf{initials}(t) \mid t \in \mathrm{T}(\Sigma)\}$$

where we recall from Sect. 2.1 that $\mathsf{initials}(t)$ denotes $\{a \in \mathsf{Act} \mid \exists t' \in \mathrm{T}(\Sigma)\,(t \stackrel{a}{\rightarrow} t')\}$. The premises of a GSOS rule as in Def. 5.22 are satisfiable iff there exist closed terms $t_1, \ldots, t_{ar(f)} \in \mathrm{T}(\Sigma)$ such that $\{a_{ij} \mid j = 1, \ldots, m_i\} \subseteq \mathsf{initials}(t_i)$ and $\{b_{ik} \mid k = 1, \ldots, n_i\} \cap \mathsf{initials}(t_i) = \varnothing$ for $i = 1, \ldots, ar(f)$.

So we are left to give an effective way of computing the set $\mathsf{initials}(\mathrm{T}(\Sigma))$ for any GSOS language. Note that each function symbol $f$ determines a computable function

$$\hat{f} : \underbrace{2^{\mathsf{Act}} \times \cdots \times 2^{\mathsf{Act}}}_{ar(f)\text{ times}} \rightarrow 2^{\mathsf{Act}}$$

by $\hat{f}(S_1, \ldots, S_{ar(f)}) \triangleq S$, where for all $c \in \mathsf{Act}$, $c \in S$ iff there exists a GSOS rule as in Def. 5.22 (with type $f$ and action $c$) such that, for $i = 1, \ldots, ar(f)$, $\{a_{ij} \mid j = 1, \ldots, m_i\} \subseteq S_i$ and $\{b_{ik} \mid 1 \leq k \leq n_i\} \cap S_i = \varnothing$. Now, for each $\mathcal{S} \subseteq 2^{\mathsf{Act}}$, let $\mathcal{G}(\mathcal{S})$ be given by

$$\mathcal{G}(\mathcal{S}) \quad \triangleq \quad \{\hat{f}(S_1, \ldots, S_{ar(f)}) \mid f \in \Sigma,\ S_1, \ldots, S_{ar(f)} \in \mathcal{S}\} \ .$$

For each $\mathcal{S} \subseteq 2^{\mathsf{Act}}$ the set $\mathcal{G}(\mathcal{S})$ can be effectively computed, and $\mathcal{S} \subseteq \mathcal{S}'$ implies $\mathcal{G}(\mathcal{S}) \subseteq \mathcal{G}(\mathcal{S}')$.

The set $\mathsf{initials}(\mathrm{T}(\Sigma))$ can be computed by dividing $\mathrm{T}(\Sigma)$ into sets $U_i$ of closed terms that contain no more than $i$ function symbols, and computing the non-decreasing sequence

$$\mathsf{initials}(U_1) \subseteq \mathsf{initials}(U_2) \subseteq \cdots$$

until it stabilizes. Obviously, this sequence stabilizes in a finite number of steps, as $\mathsf{Act}$ is finite.

The set of terms $U_0$ is empty, so $\mathsf{initials}(U_0) = \varnothing$. Now suppose that we want to compute $\mathsf{initials}(U_{i+1})$, given that we already have $\mathsf{initials}(U_i)$. We claim that $\mathsf{initials}(U_{i+1}) = \mathcal{G}(\mathsf{initials}(U_i))$. In fact, each term in $U_{i+1}$ is of the form $f(t_1, \ldots, t_{ar(f)})$, where the $t_i$'s are all in $U_i$. Thus we know $\mathsf{initials}(t_i)$ for all $i = 1, \ldots, ar(f)$, and that is exactly what is needed to determine for each transition rule of type $f$ under which closed instantiations its premises hold. Hence we can compute $\mathsf{initials}(U_1)$, and each $\mathsf{initials}(U_{i+1})$ can be computed from $\mathsf{initials}(U_i)$ using the monotonic and effective operation $\mathcal{G}$. $\square$

Clearly, junk rules can be removed from a GSOS language $T$ without altering the associated LTS. Note, moreover, that it is legitimate to eliminate all the junk rules from $T$ at once. This is because whenever $\rho_1$ and $\rho_2$ are junk rules in $T$, then $\rho_2$ is still junk in the GSOS language obtained from $T$ by removing $\rho_1$, as the two GSOS languages are associated with the same LTS.

### 5.4.3 Coding a Universal 2-Counter Machine

Despite the finiteness restrictions imposed on GSOS languages, they are a Turing powerful model of computation. We exhibit a GSOS language with, for each $n$, a term $\mathrm{U2CM}_n$ that behaves as a universal 2-counter machine on input $n$. Then $\mathrm{U2CM}_n \leftrightarrow a^\omega$, where $a^\omega \xrightarrow{a} a^\omega$, iff the 2-counter machine diverges on input $n$, a prototypical undecidable problem.

Suppose that the 2-counter machine has code of the form:

$l_1$: `if I=0 goto` $l_5$
$l_2$: `inc I`
$l_3$: `dec J`
$l_4$: `goto` $l_7$
$\vdots$
$l_k$: `halt`

We assume a toy process algebra containing the inactive constant $\mathbf{0}$ and the unary prefix multiplication operators $zero \cdot \_$ and $succ \cdot \_$, while $\mathsf{Act} \triangleq \{a, succ, zero\}$. Since $\mathbf{0}$ does not exhibit any behaviour, it does not have any transition rules. The transition rules for prefix multiplication are

$$\frac{}{succ \cdot x \xrightarrow{succ} x} \qquad \frac{}{zero \cdot x \xrightarrow{zero} x}$$

Intuitively, *succ* and *zero* represent the successor function and the zero for the counters: a natural number $n$ is encoded by the term $succ^n \cdot zero \cdot \mathbf{0}$ (where $succ^n$ denotes $n$ nestings of the prefix multiplication function *succ*). Thus, if a closed term $t$ codes $n$ and $t \overset{succ}{\rightarrow} t'$, then $n > 0$ and $t'$ codes $n - 1$. Also, if $t$ codes $n$, then $t \overset{zero}{\rightarrow} \mathbf{0}$ iff $n = 0$. The action $a$ is the pulse emitted by a process as it performs a computation step.

Finally, the syntax also contains binary function symbols $l_1, \ldots, l_k$ to code the states of the 2-counter machine: $l_i(succ^m \cdot zero \cdot \mathbf{0}, succ^n \cdot zero \cdot \mathbf{0})$ codes the machine at label $l_i$, with the two counters $\mathtt{I} = m$ and $\mathtt{J} = n$. The transition rules for these function symbols are as follows.

- If the $i$th instruction is of the form `if I=0 goto` $l_j$, then

$$\frac{x \overset{zero}{\rightarrow} x'}{l_i(x,y) \overset{a}{\rightarrow} l_j(zero \cdot x', y)} \qquad \frac{x \overset{succ}{\rightarrow} x'}{l_i(x,y) \overset{a}{\rightarrow} l_{i+1}(succ \cdot x', y)}$$

- If the $i$th instruction is `goto` $l_j$, then

$$\frac{}{l_i(x,y) \overset{a}{\rightarrow} l_j(x,y)}$$

- If the $i$th instruction is `inc I`, then

$$\frac{}{l_i(x,y) \overset{a}{\rightarrow} l_{i+1}(succ \cdot x, y)}$$

- If the $i$th instruction is `dec I`, then

$$\frac{x \overset{zero}{\rightarrow} x'}{l_i(x,y) \overset{a}{\rightarrow} l_{i+1}(zero \cdot x', y)} \qquad \frac{x \overset{succ}{\rightarrow} x'}{l_i(x,y) \overset{a}{\rightarrow} l_{i+1}(x', y)}$$

Commands that deal with the other counter `J` are similar. There are no transition rules for labels of `halt` commands, as these cause the automaton to halt. We define $\mathrm{U2CM}_n \triangleq l_1(succ^n \cdot zero \cdot \mathbf{0}, zero \cdot \mathbf{0})$. The reader will not find it hard to see that $\mathrm{U2CM}_n \leftrightarrow a^\omega$ iff the universal machine diverges on input $n$.

### 5.4.4 Infinitary GSOS Languages Inducing Regular LTSs

Regular LTSs (cf. Def. 2.2) may be used to describe many interesting concurrent systems — e.g., several communication protocols and mutual exclusion algorithms [230] — and form the basis of semantic-based automated verification tools like those presented in, e.g., [66, 81, 202]. As (subsets of) programming languages that can be given semantics by means of regular LTSs are, at least in principle, amenable to automated verification techniques, it is interesting to develop techniques to check whether languages give rise to regular LTSs. Moreover, since such a property is in general undecidable, it is useful to single out sufficient syntactic restrictions on the transition rules in a TSS, to ensure regularity of the associated LTS.

We saw in Sect. 5.4.3 that GSOS languages can specify a universal 2-counter machine, and are therefore Turing powerful. In this section, we study an infinitary version of GSOS languages, in which the finiteness restrictions on the signature, set of actions, and transition rules are (temporarily) relaxed to countability restrictions. We present a restricted version of infinitary GSOS languages, which are guaranteed to give rise to regular LTSs.

**Definition 5.24 (Infinitary GSOS)** *An* infinitary *GSOS language is a countable set of GSOS rules over a countable signature and a countable set of actions.*

In order to ensure that the associated LTSs are regular, it is necessary to impose restrictions on the class of infinitary GSOS languages, ensuring that the LTS is finitely branching and that the set of closed terms reachable from any closed term is finite. We recall that the LTS associated with a finitary GSOS language is finitely branching. However, an infinitary GSOS language such as $\varnothing / a_0 \xrightarrow{a_i} a_i$ for $i \in \mathbb{N}$ gives rise to an LTS that is infinitely branching.

**Definition 5.25 (Positive trigger)** *The* positive trigger *of a GSOS rule as in Def. 5.22 is a tuple $\langle e_1, \ldots, e_{ar(f)} \rangle$ of subsets of* $\mathsf{Act}$*, where*

$$e_i = \{a_{ij} \mid 1 \leq j \leq m_i\} \qquad (\textit{for } i = 1, \ldots, ar(f)) \ .$$

**Definition 5.26 (Boundedness)** *Assume a function symbol $f$ in the signature of an infinitary GSOS language. We say that:*

- *$f$ is* bounded *if for each positive trigger, the corresponding set of GSOS rules of type $f$ is finite.*

- $f$ *is* uniformly bounded *if there exists a finite upper bound on the number of GSOS rules of type* $f$ *having the same positive trigger.*

As far as we know, all standard operations in process algebras that occur in the literature are uniformly bounded. The notion of a bounded function symbol was originally developed by Vaandrager [221] for De Simone languages (see Def. 5.14), and was extended to infinitary GSOS languages in [5]. The notion of a uniformly bounded function symbol stems from [4], and will reoccur in the definition of a regular GSOS language (see Def. 5.39). The following result is from [5].

**Proposition 5.27** *If each function symbol in the signature of an infinitary GSOS language is bounded, then the associated LTS is finitely branching.*

We introduce a further restriction on infinitary GSOS languages from [5], to ensure that in the associated LTSs each state can reach only finitely many other states.

**Definition 5.28 (Simple GSOS)** *A GSOS rule is* simple *if its target contains at most one function symbol. A GSOS language is* simple *if each of its transition rules is.*

Rule formats similar to the simple GSOS rules have emerged in work by several researchers, e.g., [20], [71, p. 230], and [172, Def. 13]. Most of the standard operations in process algebras are given operational semantics by means of simple GSOS rules. An exception is the binary Kleene star, which was discussed in Sect. 5.4.1. Two further exceptions are the "desynchronizing" $\Delta$ operation present in the early versions of Milner's SCCS [156] studied in [116, 155], and the parallel composition operation in the $\pi$-calculus [161]. The $\Delta$ operation has GSOS rules

$$\frac{x \xrightarrow{a} x'}{\Delta x \xrightarrow{a} \delta\Delta x'}$$

for $a \in \mathsf{Act}$, where $\delta$ is the *delay* operation of SCCS. The GSOS rules for the parallel composition operation of the $\pi$-calculus dealing with so-called scope extrusion (see [161, part II]) take the form

$$\frac{x \xrightarrow{\overline{v}(w)} x' \qquad y \xrightarrow{v(w)} y'}{x|y \xrightarrow{\tau} (w)(x'|y')}$$

where $(w)$ denotes the restriction operation of the $\pi$-calculus and $\tau$ a silent step (cf. Sect. 5.5).

The following result can be shown by structural induction on closed terms, following the lines of [5, Thm. 5.5].

**Theorem 5.29** *Assume a simple infinitary GSOS language. If each function symbol in its signature is bounded and depends on only finitely many function symbols (cf. Def. 5.16), then the associated LTS is regular.*

The above result would not hold if we allowed GSOS rules with more than one function symbol in their targets, as the following example shows.

**Example:** Consider a GSOS language with action $a$, constants $b$ and $c$, a unary function symbol $f$, and transition rules

$$\frac{}{c \xrightarrow{a} b} \qquad \frac{}{c \xrightarrow{a} f(c)} \qquad \frac{}{f(x) \xrightarrow{a} x} \qquad \frac{x \xrightarrow{a} y}{f(x) \xrightarrow{a} f(y)}$$

Note that the second transition rule of type $c$ is not simple, as its target carries two function symbols. It is not hard to see that $c$ can reach infinitely many states $f^n(c)$ and $f^n(b)$ for $n \in \mathbb{N}$, because $f^n(c) \xrightarrow{a} f^{n+1}(c)$ and $f^n(c) \xrightarrow{a} f^n(b)$. Moreover, these states are all non-bisimilar. $\square$

Madelaine and Vergamini [144] studied syntactic conditions on De Simone rules [199, 200] to ensure that the associated LTS is regular. They identify two classes of well-behaved function symbols, which they call *non-growing operations* and *sieves*. Intuitively, non-growing operations are function symbols which, when fed with (terms denoting) regular LTSs, build regular LTSs. Sieves are a special class of unary non-growing operations whose transition rules have the form

$$\frac{x \xrightarrow{a} x'}{f(x) \xrightarrow{b} f(x')}$$

For example, standard process algebra operations like CCS restriction and renaming [158] and CSP hiding [128] are sieves. Note that transition rules for sieves are simple. In view of Thm. 5.29, all function symbols in an infinitary GSOS language given by means of simple transition rules are non-growing in the sense of Madelaine and Vergamini.

The syntactic condition used by Madelaine and Vergamini to establish that some operations are non-growing is based on term rewriting techniques, to find a so-called *simplification ordering* over terms (see [146, Def. 4]). This is similar in spirit to showing that linear GSOS languages (see Def. 5.35) are syntactically well-founded (see Def. 5.36); the interested reader is referred to Sect. 5.4.5 and [8, Sect. 6] for more information. Unfortunately, the existence of a simplification ordering compatible with a set of rewrite rules is not decidable even for finitary GSOS languages.

Specialized techniques which can be used to show that certain closed terms can reach only finitely many other closed terms have been proposed for CCS and related languages. The interested reader is invited to consult [74] and the references therein. Not surprisingly, these specialized methods tend to be more powerful than general syntactic ones, as they rely on language-dependent semantic information. For instance, a method to check the regularity of a large set of CCS terms based on abstract interpretation techniques (see, e.g., [2]) has been proposed in [74].

### 5.4.5 Turning GSOS Rules into Equations

There are several methods for specifying and verifying processes behaviour, e.g. modal formulae [209] and variants of Hoare logic [170, 208]. A fairly successful verification technique is to approximate the specification by a (not necessarily implementable) term in some process algebra. In this setting, a set of axioms can be applied to try and show that the term is behaviourally equivalent to, or in some other sense a suitable approximation of, the required specification. Indeed, one of the major schools of theoretical concurrency and its applications, that of ACP [21, 29], takes the notion of behavioural equivalence as primary, and defines operational semantics to fit its axioms.

A logic of programs is complete (relative to a programming language) if all true formulas of the language are provable in the logic. As properties of interest are generally non-recursive, we are often obliged to have infinitary or other non-recursive rules in our logics to achieve completeness.

This section presents results from [7, 8], which offer an algorithmic solution to the problem of computing a sound and complete axiomatization (possibly including one infinitary conditional axiom) for any GSOS language, modulo bisimulation equivalence. That is, two closed terms can be equated by the axiom system iff they are bisimilar in the associated LTS (cf. Def. 4.9). The procedure introduces fresh function symbols as needed. Completeness results for axiomatizations have become rather standard in many cases. The generalization of extant completeness results given in [8] shows that, at least in principle, this burden can be completely removed if one gives GSOS rules for a process algebra. Of course, this does not mean that there is nothing to do on specific process algebras. For instance, sometimes it may be possible to eliminate some of the auxiliary function symbols (we will see an example of this later on in this section), or the infinitary conditional axiom. (The interested reader will find many results on complete axiomatizations of behavioural equivalences over several process algebras elsewhere in this handbook.)

We first define the GSOS language $T_{\mathsf{FINTREE}}$, which is a fragment of CCS suitable for expressing finite LTSs (cf. Def. 2.2). Its signature $\Sigma_{\mathsf{FINTREE}}$ consists of:

- the constant $\mathbf{0}$, denoting the inactive process;
- binary alternative composition $x+y$, which chooses non-deterministically between $x$ and $y$;
- unary prefix multiplication $a{\cdot}x$ for $a \in \mathsf{Act}$, which executes action $a$ and thereafter behaves as $x$.

The constant $\mathbf{0}$ does not exhibit any behaviour and consequently does not have any transition rules. The transition rules of alternative composition and prefix multiplication have been formulated earlier in this chapter (see Table 1 in Sect. 2.6, and Sect. 5.4.3, respectively). Most process algebras contain the function symbols above, either directly or as derived operations. The following completeness result (cf. Def. 4.9) is well-known [123, 158].

**Proposition 5.30** *Let $\mathcal{E}_{\mathsf{FINTREE}}$ denote the axiomatization*

$$\begin{aligned} x+y &= y+x && (3)\\ (x+y)+z &= x+(y+z) && (4)\\ x+x &= x && (5)\\ x+\mathbf{0} &= x && (6) \end{aligned}$$

*$\mathcal{E}_{\mathsf{FINTREE}}$ is sound and complete modulo bisimulation equivalence as induced by $T_{\mathsf{FINTREE}}$.*

Following [8], we show how to find for any GSOS language $T$ extending $T_{\mathsf{FINTREE}}$, an axiomatization $\mathcal{E}$, extending $\mathcal{E}_{\mathsf{FINTREE}}$, that is sound and complete modulo bisimulation equivalence. That is, two closed terms are bisimilar as states in the LTS associated with $T$ iff they can be equated by $\mathcal{E}$.

Moller [163] has shown that bisimulation equivalence over a subset of CCS with the interleaving operation $\|$, which can be defined in GSOS, cannot be completely characterized by any finite unconditional axiomatization over that language. Thus, the algorithm to produce $\mathcal{E}$ may require the addition of auxiliary function symbols to the signature, and of GSOS rules for these auxiliary function symbols to $T$.

We start with a typical example of the way in which the completeness result for $T_{\mathsf{FINTREE}}$ in Prop. 5.30 is used.

**Example:** Consider the GSOS language $T_{\partial^1}$ that is obtained by extending the signature of $T_{\mathsf{FINTREE}}$ with unary function symbols $\partial^1_H$ [22], where $H$ is a finite set of actions, and adding the transition rules

$$\frac{x \stackrel{a}{\to} y}{\partial^1_H(x) \stackrel{a}{\to} y} \qquad a \notin H$$

In other words, the process $\partial^1_H(t)$ behaves like $t$, except that it cannot do any actions from $H$ in its first move. Note that $\partial^1_H$ is different from the CCS restriction operation, as CCS restriction is persistent while $\partial^1_H$ disappears after one transition.

The following result, whose proof may be found in [8], is a corollary of Thm. 4.12 on completeness of axiomatizations over extended signatures, and a blueprint for developments to follow. The idea is that, using the axioms below, the completeness problem for a super-language of $T_{\mathsf{FINTREE}}$ can be reduced to the completeness problem for $T_{\mathsf{FINTREE}}$, which has been solved in Prop. 5.30.

**Proposition 5.31** *Let $\mathcal{E}_{\partial^1}$ be the axiomatization that extends $\mathcal{E}_{\mathsf{FINTREE}}$ with the axioms*

$$\begin{array}{rcll}
\partial^1_H(x+y) & = & \partial^1_H(x) + \partial^1_H(y) & \\
\partial^1_H(a \cdot x) & = & a \cdot x & \textit{if } a \notin H \\
\partial^1_H(a \cdot x) & = & \mathbf{0} & \textit{if } a \in H \\
\partial^1_H(\mathbf{0}) & = & \mathbf{0} &
\end{array}$$

*$\mathcal{E}_{\partial^1}$ is sound and complete modulo bisimulation equivalence as induced by $T_{\partial^1}$.*

Prop. 5.31 follows from Thm. 4.12, owing to the observations that $T_{\partial^1}$ is an operational conservative extension (cf. Def. 4.2) of $T_{\mathsf{FINTREE}}$ (this follows from Thm. 4.4), $\mathcal{E}_{\partial^1}$ is sound over the extended signature, $\mathcal{E}_{\mathsf{FINTREE}}$ is complete over $\Sigma_{\mathsf{FINTREE}}$ (Prop. 5.30), and each closed term over the extended signature can be equated to a closed term over $\Sigma_{\mathsf{FINTREE}}$ by means of the axiomatization $\mathcal{E}_{\partial^1}$. $\square$

The approach that yielded a sound and complete axiomatization for $T_{\partial^1}$ modulo bisimulation equivalence can be generalized to arbitrary GSOS superlanguages of $T_{\mathsf{FINTREE}}$. That is, we want to find axioms, on top of $\mathcal{E}_{\mathsf{FINTREE}}$, that allow us to eliminate all extra function symbols from closed terms. This requires a variety of methods, in which Prop. 5.31 plays an important role, because the $\partial^1_H$ operator can be used to encode negative premises.

The following notion from [8] is needed in presenting the main result on the automatic generation of axiomatizations modulo bisimulation equivalence for GSOS languages.

**Definition 5.32 (Disjoint extension)** *A GSOS language $T_1$ is a* disjoint extension *of a GSOS language $T_0$ if the signature and transition rules of $T_1$ include those of $T_0$, and the types of transition rules in $T_1$, which were not present in $T_0$, are not in the signature of $T_0$.*

Note that disjoint extension is a partial order. If $T_1$ disjointly extends $T_0$, then $T_0 \oplus T_1$ is an operational conservative extension (see Def. 4.2) of $T_0$. This follows immediately from Thm. 4.4, owing to the fact that GSOS rules are by default source-dependent (cf. Def. 4.3 and Def. 5.22), and the sources of transition rules in $T_1$ are fresh.

Before presenting the main results of [8], we need to discuss a subtlety. We want to know that the axioms in $\mathcal{E}_{\mathsf{FINTREE}}$ are sound modulo bisimulation equivalence as induced by any disjoint extension $T$ of $T_{\mathsf{FINTREE}}$. In general it is not the case that validity of axioms is preserved by taking disjoint extensions. For instance, consider the trivial GSOS language $T_{\mathsf{NIL}}$ consisting of the constant $\mathbf{0}$ and no transition rules. The axiom $x = y$ is sound modulo bisimulation equivalence as induced by $T_{\mathsf{NIL}}$, but clearly this law is not sound modulo bisimulation equivalence as induced by $T_{\mathsf{FINTREE}}$, even though $T_{\mathsf{FINTREE}}$ is a disjoint extension of $T_{\mathsf{NIL}}$. Fortunately, soundness of the axioms in $\mathcal{E}_{\mathsf{FINTREE}}$, and also all the other axioms that are needed in the developments of [8], is preserved by taking disjoint extensions of $T_{\mathsf{FINTREE}}$.

**Semantically Well-Founded GSOS Languages** We start with the generation of sound and complete axiomatizations modulo bisimulation equivalence for the limited class of *semantically well-founded* GSOS languages, generating finite LTSs (cf. Def. 2.2). For such languages we can do without infinitary conditional axioms.

**Definition 5.33 (Semantic well-foundedness)** *A GSOS language is* semantically well-founded *if its associated LTS is finite.*

The class of semantically well-founded GSOS languages contains the recursion-free finite-alphabet sublanguages of most of the standard process algebras.

In [8] an algorithm is presented that, given a disjoint extension $T$ of $T_{\mathsf{FINTREE}}$, constructs a disjoint extension $T'$ of $T$ and $T_{\partial^1}$, and a finite (unconditional) axiomatization $\mathcal{E}$ that is sound modulo bisimulation equivalence as induced by $T'$, such that each closed term over the signature of $T'$ can be equated by $\mathcal{E}$ to a closed term of the form $a_1{\cdot}t_1+\cdots+a_n{\cdot}t_n$ (where the empty sum represents $\mathbf{0}$). For semantically well-founded GSOS languages it is possible to iterate this reduction a finite number of times, thereby eliminating all function symbols that are not in $\Sigma_{\mathsf{FINTREE}}$. As in the completeness proof for $T_{\partial^1}$, this reduces completeness of $\mathcal{E}$ with respect to $T'$ to completeness of $\mathcal{E}_{\mathsf{FINTREE}}$ with respect to $T_{\mathsf{FINTREE}}$, which has been solved in Prop. 5.30. Thus we obtain the following result.

**Theorem 5.34** *There is an algorithm that, given a GSOS language $T$, which is a semantically well-founded disjoint extension of $T_{\mathsf{FINTREE}}$, constructs a disjoint extension $T'$ of $T$ and $T_{\partial^1}$, and a finite unconditional axiomatization $\mathcal{E}$, such that $\mathcal{E}$ is sound and complete modulo bisimulation equivalence as induced by $T'$.*

The requirement that $T$ is a disjoint extension of $T_{\mathsf{FINTREE}}$ is necessary in Thm. 5.34, because otherwise the quoted algorithm might not preserve semantic well-foundedness. Namely, the combination of a semantically well-founded GSOS language with $T_{\mathsf{FINTREE}}$ is in general not semantically well-founded; see [8].

**Example:** An operation found in many process algebras is the parallel composition without communication, which is defined by the transition rules

$$\frac{x \stackrel{a}{\to} x'}{x \parallel y \stackrel{a}{\to} x' \parallel y} \qquad \frac{y \stackrel{a}{\to} y'}{x \parallel y \stackrel{a}{\to} x \parallel y'}$$

for $a \in \mathsf{Act}$. This is an intuitively reasonable definition of parallel composition, and the transition rules are easy to explain. It is somewhat harder to see how to describe it equationally. Some axioms are clear enough—the operation $\parallel$ is commutative and associative, and the stopped process is its identity—but the first finite equational description did not appear until [35]. This equational characterization required an additional function symbol "left merge" $\mathbin{\|\!\_}$. Intuitively, $t \mathbin{\|\!\_} u$ behaves as $t \| u$ except that its first move must be taken by $t$. For each $a \in \mathsf{Act}$, left merge has a transition rule

$$\frac{x \stackrel{a}{\to} x'}{x \mathbin{\|\!\_}\, y \stackrel{a}{\to} x' \parallel y}$$

The axioms for $\|$ and $\mathbin{\|\!\!\!\_}$ are:

$$\begin{array}{rcl} x \parallel y & = & (x \mathbin{\|\!\!\!\_} y) + (y \mathbin{\|\!\!\!\_} x) \\ (x+y) \mathbin{\|\!\!\!\_} z & = & (x \mathbin{\|\!\!\!\_} z) + (y \mathbin{\|\!\!\!\_} z) \\ (a \cdot x) \mathbin{\|\!\!\!\_} y & = & a \cdot (x \parallel y) \\ \mathbf{0} \mathbin{\|\!\!\!\_} x & = & \mathbf{0} \end{array}$$

These axioms for $\|$ and $\mathbin{\|\!\!\!\_}$, together with the axioms in $\mathcal{E}_{\partial^1}$ for $+$, $a \cdot \_$, $\mathbf{0}$, and $\partial_H^1$, form a sound and complete axiomatization for the closed terms over this signature modulo bisimulation equivalence.

$T_{\mathsf{FINTREE}}$ with parallel composition is a semantically well-founded GSOS language and a disjoint extension of $T_{\mathsf{FINTREE}}$. The auxiliary operator $\mathbin{\|\!\!\!\_}$, and the axioms above for parallel composition and the left merge, are also produced by the algorithm from [8] that was mentioned in Thm. 5.34. In fact, due to the symmetric character of the parallel composition operator, the algorithm from [8] actually produces two auxiliary operators $\mathbin{\|\!\!\!\_}$ and $\mathbin{\_\!\!\!\|}$, where $t \mathbin{\_\!\!\!\|} u$ behaves as $t\|u$ except that its first move must be taken by $u$. Parallel composition is then axiomatized by

$$x \parallel y \;=\; (x \mathbin{\|\!\!\!\_} y) + (x \mathbin{\_\!\!\!\|} y) \;\; .$$

However, since $t \mathbin{\_\!\!\!\|} u \mathrel{\underline{\leftrightarrow}} u \mathbin{\|\!\!\!\_} t$, the right merge $\mathbin{\_\!\!\!\|}$ can be expressed by means of the left merge $\mathbin{\|\!\!\!\_}$. $\square$

**Syntactic Well-Foundedness** Since GSOS languages are Turing powerful, it is undecidable whether a GSOS language is semantically well-founded. However, for an interesting subclass of GSOS languages there exist effective syntactic constraints on GSOS rules that ensure semantic well-foundedness.

**Definition 5.35 (Linear GSOS)** *A GSOS rule as in Def. 5.22 is* linear *if each variable occurs at most once in the target $t$ and, for each argument $i$ that is tested positively (cf. Def. 5.11), $x_i$ does not occur in the target and at most one of the $y_{ij}$'s does.*

*A GSOS language is* linear *if all its transition rules are.*

As far as we know, all transition rules for standard process algebras are linear.

**Definition 5.36 (Syntactic well-foundedness)** *A GSOS language is* syntactically well-founded *if there exists a weight mapping $w$ from function symbols to natural numbers such that, for each GSOS rule $\rho$ with type $f$ and target $t$:*

- *if $\rho$ has no positive premise then $W(t) < w(f)$, and*
- *$W(t) \leq w(f)$ otherwise,*

*where $W(t)$ adds the weights of all function symbols in $t$:*

$$\begin{array}{rcl} W(x) & \triangleq & 0 \\ W(g(t_1, \ldots, t_{ar(g)})) & \triangleq & w(g) + W(t_1) + \cdots + W(t_{ar(g)}) \ . \end{array}$$

**Proposition 5.37** *If a GSOS language is linear and syntactically well-founded, then it is semantically well-founded. Moreover, it is decidable whether a GSOS language is syntactically well-founded.*

**General GSOS Languages** It follows from some recursion theoretic considerations, discussed in [8] and based upon the programming exercise in Sect. 5.4.3, that the extension of the completeness result given in Theorem 5.34 to general GSOS languages requires some proof rules beyond purely equational logic. However, it is possible to extend the completeness result to the whole class of GSOS languages in a rather standard way. Bisimulation equivalence over finitely branching LTSs supports a powerful induction principle, known as the *Approximation Induction Principle* (AIP) [39, 94]. Since the LTS associated with a GSOS language is finitely branching, AIP applies.

We introduce a family of unary function symbols $\pi_n(\_)$ for $n \in \mathbb{N}$, with transition rules

$$\frac{x \xrightarrow{a} y}{\pi_{n+1}(x) \xrightarrow{a} \pi_n(y)}$$

for $a \in \mathsf{Act}$. These function symbols are known as *projection* operations in the literature on ACP [29]. Intuitively, $\pi_n(t)$ allows $t$ to perform $n$ moves freely, and then stops it. AIP is the infinitary conditional axiom

$$\text{AIP} \qquad \forall n \in \mathbb{N} \, (\pi_n(x) = \pi_n(y)) \ \Rightarrow \ x = y \ .$$

Intuitively, this proof rule states a "continuity" property of bisimulation equivalence over finitely branching LTSs: if two states are bisimilar at any finite depth, then they are bisimilar.

The projection operators are somewhat heavy-handed, as there are infinitely many of them, and GSOS languages are defined to be finite. It is, however, possible to mimic the projection operations by means of a binary function symbol $\_/\_$. Intuitively, a closed term $t/u$ executes $t$ (where $u$ also silently takes a step) until the "hourglass" process $u$ runs out and halts the execution. That is, for all actions $a, b \in \mathsf{Act}$ we have the following transition rule for $\_/\_$:

$$\frac{x \xrightarrow{a} x' \qquad y \xrightarrow{b} y'}{x/y \xrightarrow{a} x'/y'} \tag{7}$$

For each $n \in \mathbb{N}$, $\pi_n(t) \leftrightarrow t/c^n$ with $c$ an arbitrarily chosen action. In this formulation, we may rephrase AIP as follows:

$$\text{AIP}' \qquad \forall n \in \mathbb{N}\,(x/c^n = y/c^n) \;\Rightarrow\; x = y \ .$$

Now we are ready to formulate the analogue of Thm. 5.34 for GSOS languages that need not be semantically well-founded.

**Theorem 5.38** *There is an algorithm that, given a GSOS language $T$, constructs a disjoint extension $T'$ of $T$, $T_{\partial^1}$, and the operation $\_/\_$ defined by (7), and a finite unconditional axiomatization $\mathcal{E}$, such that $\mathcal{E}$ and AIP′ together are sound and complete modulo bisimulation equivalence as induced by $T'$.*

Term rewriting properties of the axiomatizations generated by (variations on) the methods we have just surveyed have been studied by Bosscher [59]. Ulidowski [216] proposed a modification of the approach in [8] that produces complete axiomatizations for a subclass of the De Simone languages, modulo the refusal simulation preorder from [215] that takes into account the silent step $\tau$ (cf. Sect. 5.5).

**Regular GSOS Languages** In [4] it was shown that for a subclass of infinitary recursive GSOS languages generating regular LTSs it is possible to obtain sound and complete axiomatizations modulo bisimulation equivalence that do not rely on infinitary proof rules like AIP. The following definition introduces the class of *regular* GSOS languages that is considered in *op. cit.*, which is a subclass of the simple infinitary GSOS languages (cf. Def. 5.28).

**Definition 5.39 (Regular GSOS)** *A simple infinitary GSOS language is* regular *if for each function symbol $f$ in its signature:*

1. *$f$ is uniformly bounded (cf. Def. 5.26);*

2. *$f$ depends on only finitely many function symbols (cf. Def. 5.16);*

3. *there is a finite upper bound on the number of positive premises in transition rules with type $f$.*

Most of the TSSs for standard process algebras in the literature are regular. It follows immediately from the theory outlined in Sect. 5.4.4 that every regular GSOS language induces a regular LTS. In [4] it was shown how to reduce the completeness problem for regular GSOS languages to that for regular LTSs, which was solved earlier in [40, 157].

We recall from Sect. 5.3.1 on De Simone languages that the recursive terms over a signature $\Sigma$ are given by the BNF grammar

$$t ::= X \mid f(t_1, \ldots, t_{ar(f)}) \mid \mathrm{fix}(X = t)$$

where $X$ ranges over a countably infinite set of recursion variables, $f$ ranges over the signature $\Sigma$, and fix is a binding construct. The latter construct gives rise to the usual notions of free and bound recursion variables in recursive terms. For every recursive term $\mathrm{fix}(X = t)$ there is a transition rule

$$\frac{t[\mathrm{fix}(X = t)/X] \stackrel{a}{\to} y}{\mathrm{fix}(X = t) \stackrel{a}{\to} y}$$

We consider the subset of guarded recursive terms (cf. Def. 5.12) without free recursion variables.

Bisimulation equivalence over guarded recursive terms has been completely axiomatized by Milner [157] and Bergstra and Klop [40]. The following proof rules, called the *Recursive Definition Principle* (RDP) and the *Recursive Specification Principle* (RSP), play a key role in these completeness proofs. Let $t$ denote a guarded recursive term with no other free recursion variables than $X$, and let $u$ denote a guarded recursive term without free recursion variables:

$$\begin{array}{ll} \mathrm{RDP} & \mathrm{fix}(X = t) \;=\; t[\mathrm{fix}(X = t)/X] \\ \mathrm{RSP} & u \;=\; t[u/X] \;\;\Rightarrow\;\; u \;=\; \mathrm{fix}(X = t)\;\;. \end{array}$$

**Theorem 5.40** *There is an algorithm that, given a regular GSOS language $T$, constructs a disjoint extension $T'$ of $T$ over guarded recursive terms, and an unconditional axiomatization $\mathcal{E}$, such that $\mathcal{E}$, RDP, and RSP together are sound and complete modulo bisimulation equivalence as induced by $T'$.*

The algorithm used in the proof of Thm. 5.40 does not work for GSOS languages that are not regular; see [4] for details.

Apart from the work we have just reviewed, the automatic generation of complete axiomatizations from TSSs has received a good deal of attention in the literature. Further results on this line of research may be found in, e.g., [12, 51, 217].

### 5.4.6 From Recursive GSOS to LTSs with Divergence

This section considers GSOS languages for recursive terms (cf. Sect. 5.3.1 and the end of the previous section) over a signature $\Sigma$. Let $\mathsf{CREC}(\Sigma)$ denote the set of recursive terms that do not contain free recursion variables.

We recall that, in the absence of recursion, GSOS languages are strictly stratified (cf. Def. 3.13), yielding one of the most restrictive criteria for meaningful TSSs considered in Sect. 3. For recursive GSOS languages, however, it is no longer straightforward to find an associated LTS. In this section it is shown how this problem can be overcome by the use of a special divergence predicate in the definition of LTSs. The interested reader is referred to, e.g., [116, 126, 154, 231] for motivation and more information on (variations on) this semantic model for reactive systems.

**Definition 5.41 (LTS with divergence)** *An* LTS with divergence *is an LTS, in the sense of Def. 2.1, whose only predicates are the* divergence *predicate* $\uparrow$ *and the* convergence *predicate* $\downarrow$*.*

In [11, 12] it is shown how a recursive GSOS language specifies an LTS with divergence over $\mathsf{CREC}(\Sigma)$. The recursive GSOS languages considered in *op. cit.* contain a special constant $\Omega$ that is akin to the constant $\mathbf{0}$ from CCS; i.e., there are no transition rules of type $\Omega$. (The difference between $\mathbf{0}$ and $\Omega$ is that whereas the former denotes the convergent process without transitions, the latter stands for the divergent stopped process.) The transition rules in a recursive GSOS language are used to define a divergence (or under-specification) predicate on $\mathsf{CREC}(\Sigma)$. In fact, as is common practice in the literature on process algebras, we first define the notion of convergence, and use it to define the divergence predicate.

**Definition 5.42 (Convergence)** *Assume a recursive GSOS language over a signature $\Sigma$. The convergence predicate $\downarrow$ is the least predicate over the set of closed recursive terms $\mathsf{CREC}(\Sigma)$ that satisfies the following clauses:*

1. $f(t_1, \ldots, t_{ar(f)}) \downarrow$ *if*
   (a) $f \neq \Omega$*, and*

*(b) for every argument $i$ of $f$, if $f$ tests $i$ (cf. Def. 5.11) then $t_i \downarrow$;*

*2.* $\mathrm{fix}(X = t) \downarrow$ *if* $t[\mathrm{fix}(X = t)/X] \downarrow$.

*We write $t \uparrow$ if it is not the case that $t \downarrow$.*

The motivation for the above definition is the following: a term $t$ is divergent if its initial transitions are not fully specified. This occurs either when the initial behaviour of term $t$ depends on under-specified arguments like $\Omega$, or in the presence of unguarded recursive definitions (cf. Def. 5.12). For example, the terms $\mathrm{fix}(X = X)$ and $\mathrm{fix}(X = a.\mathbf{0} + X)$ are not convergent, as the initial behaviours of these terms depend on themselves. We remark here that, when applied to SCCS [156] and the version of CCS considered in [231], the above definition delivers exactly the convergence predicates given by Hennessy [116] and Walker [231], respectively.

We hint at how an LTS with divergence can be associated with a recursive GSOS language — the interested reader is referred to [11, 12] for full technical details. We want the LTS with divergence to be at least a supported model in the sense of Def. 3.2. Moreover, reminiscent of positive TSSs, we want to associate the least such LTS with the recursive GSOS language in question. As described in, e.g., [55], this is not possible in the absence of divergence. However, the extra structure given by the convergence predicate can be put to good use in giving a simple way of constructing the desired LTS with divergence over $\mathsf{CREC}(\Sigma)$, in two steps. First, the transitions emanating from convergent terms are derived by induction on the convergence predicate. This is done according to the standard approach for GSOS languages outlined in Sect. 5.4, and using the transition rule for recursion to derive the transitions of recursive terms. Next, the information about the transitions that are possible for convergent terms is used to determine the outgoing transitions for all terms in $\mathsf{CREC}(\Sigma)$. The key point in the construction of the associated LTS is that negative premises can be satisfied by convergent terms only. Intuitively, to know that a closed term cannot initially perform a given action, we need to find out precisely all the initial actions that it can perform. If a closed term is divergent, then its set of initial actions is not fully specified; thus we cannot be sure whether such a term satisfies a negative premise or not.

The reader familiar with the literature on Hennessy-Milner logics (cf. Def. 2.7) for prebisimulation-like relations (cf. Def. 7.2 to follow) may have noticed that the notion of satisfaction for negative premises discussed above is akin to that for formulae of the form $[a]\varphi$ given in, e.g., [1, 10, 154, 207, 209]. In those references, the new interpretation is necessary to obtain monotonicity of the satisfaction relation with respect to the appropriate notion of prebisimulation. The intuitionistic interpretation of negative premises given in [11, 12] is crucial to obtain operations that are monotonic with respect to the notion of prebisimulation $\lesssim$ presented in Def. 7.2. Basically, it ensures that, for a closed term $t$, the transition formula $t \stackrel{a}{\nrightarrow}$ holds iff $u \stackrel{a}{\nrightarrow}$ holds for every closed term $u$ with $t \lesssim u$.

The following result is from [11, 12], where the details of the construction of the desired LTS with divergence may be found.

**Proposition 5.43** *For every recursive GSOS language with the inert constant $\Omega$, there exists a least sound and supported LTS with divergence over* $\mathsf{CREC}(\Sigma)$.

**Example:** Consider the term $\mathrm{fix}(X = \mathit{odd}(X))$, where the unary function symbol *odd* is defined by the transition rule

$$\frac{x \stackrel{a}{\nrightarrow}}{\mathit{odd}(x) \stackrel{a}{\rightarrow} \mathbf{0}}$$

This operation is a standard example used in the literature to show that negative premises and unguarded recursive definitions can lead to inconsistent specifications (see, e.g., [45]). The reason for this phenomenon is that, if we follow the standard GSOS approach, the recursive equation

$$X \quad = \quad \mathit{odd}(X)$$

does not have any solution modulo bisimulation equivalence. In fact, with the standard operational interpretation of GSOS languages and general TSSs with negative premises, a term $t$ solving the above recursive equation modulo bisimulation equivalence (i.e., $t \leftrightarroweq \mathit{odd}(t)$) would have to exhibit an initial $a$-transition iff it does not have one. In the approach of [11, 12], instead, the above recursive equation has a unique solution modulo prebisimulation equivalence (see Def. 7.2). Namely, $\mathrm{fix}(X = \mathit{odd}(X))$ is a divergent term. Since negative premises are interpreted over convergent terms only, the above transition rule cannot be applied to derive a transition for $\mathrm{fix}(X = \mathit{odd}(X))$, so this term is prebisimilar to the inert constant $\Omega$. Thus $\Omega$ is the unique solution of $X = \mathit{odd}(X)$ modulo prebisimulation equivalence. $\square$

### 5.4.7 Other Results for GSOS Languages

The theory of GSOS languages is rather rich in results, and we have only presented a sample of the body of work documented in the literature on this rule format. We end this overview of the work on GSOS languages with pointers to other interesting results.

**Ruloids** Bloom, Istrail, and Meyer [55] observed that the behaviour of each open term $C[x_1, \ldots, x_N]$ is completely determined by a finite set of derived transition rules of the form

$$\frac{\{x_i \stackrel{a_{ij}}{\rightarrow} y_{ij},\ x_i \stackrel{b_{ik}}{\nrightarrow} \mid 1 \leq i \leq N, 1 \leq j \leq m_i, 1 \leq k \leq n_i\}}{C[x_1, \ldots, x_N] \stackrel{c}{\rightarrow} t}$$

These derived transition rules are referred to as *ruloids*, to distinguish them from the GSOS rules defining the operational semantics of the language under consideration. Ruloids facilitate the development of theory for SOS. For instance, the use of ruloids simplifies the proof of the congruence result for TSSs, Thm. 5.3, in the restricted case of GSOS languages. The following result may be found in [55, Thm. 7.6].

**Theorem 5.44** *Let $T$ be a GSOS language. For every open term $C[x_1, \ldots, x_N]$ there exists a finite set $R$ of ruloids such that:*

- *every ruloid in $R$ has $C[x_1, \ldots, x_N]$ as its source;*
- *the LTS associated with $T$ is a model of $R$ (cf. Def. 3.2);*
- *if $C[t_1, \ldots, t_N] \stackrel{c}{\rightarrow} u$, then there exists a ruloid in $R$ supporting that transition in the sense of Def. 3.2.*

**Example:** Consider the process algebra BPA with the priority operator $\theta$ from Sect. 2.6. The set of ruloids determining the operational semantics of the term $\theta(a.x + y)$ consists of

$$\frac{y \stackrel{b}{\nrightarrow} \text{ for } b > a}{\theta(a.x + y) \stackrel{a}{\rightarrow} \theta(x)} \qquad \frac{y \stackrel{c}{\rightarrow} y' \quad y \stackrel{b}{\nrightarrow} \text{ for } b > c}{\theta(a.x + y) \stackrel{c}{\rightarrow} \theta(y')}$$

where the second ruloid is only present for actions $c$ that are not smaller than $a$. □

**Protean Specification Languages** Case studies in the literature on process algebras often use mechanisms to define new operations on terms. Vaandrager [219] formulated the "fresh atom principle" to formalize a standard practice in process algebra proofs, namely, the introduction of fresh constants. Verhoef [224, 225] introduced the "operator definition principle" (similar to RDP as discussed at the end of Sect. 5.4.5) to facilitate the specification of new unary function symbols in process algebra.

Bloom, Cheng, and Dsouza [52, 53] advocated the use of "Protean" languages to enhance the expressiveness, and ease of use, of specification languages. Intuitively, when writing specifications by means of a process algebra, one is often faced with the choice between the use of a few basic operations with a clear semantics, or the introduction of ad hoc operations that simplify the writing of specifications and enhance their readability, but which complicate reasoning about the resulting high-level description of the behaviour. The aforementioned paper argues that the use of SOS, combined with the theory presented in this overview work, allows for the systematic extension of process algebras in a way that is guaranteed to preserve the semantic properties of the original language.

**Compositional Proof Systems for HML** Proof systems for modal logics enable to give formal proofs that (states in) LTSs satisfy certain requirements. A desirable feature of such proof systems is that they should allow for a *compositional* style of proof development. Informally, a proof system is compositional if it builds a proof for a property of an LTS out of proofs for properties of certain sub-LTSs.

The work presented in [203] is based upon the realization that, in the context of pure first-order logic, the issue of compositionality was addressed by Gentzen [93] in his work on the sequent calculus. There, compositionality is obtained via cut-elimination. In [203], Simpson developed a sequent calculus for showing that closed terms in a process algebra with its operational semantics specified in the GSOS format satisfy assertions of the modal logic HML (see Sect. 2.3). Such process algebras provide interesting examples, because of the well-known difficulties in giving proof rules for flavours of parallel composition [207, 234]. As usual, the benefit of working with an arbitrary GSOS language is that one obtains a generic proof system that is applicable to a wide class of process algebras.

**Binary Decision Diagrams from GSOS Languages** Binary decision diagrams [62] are widely used to represent LTSs symbolically in the second generation of verification tools for concurrent processes — see, e.g., McMillan's textbook on the model checker SMV [148]. A binary decision diagram for such an application is often generated from an LTS over the closed terms in some process calculus, which in turn is generated using the transition rules defining the operational semantics of this calculus. Such a two-step approach can be avoided by using a direct construction of the binary decision diagram from the transition rules. This is the approach followed in [76] for GSOS languages. The results in *op. cit.* suggest that this general procedure yields binary decision diagrams that are of comparable quality to those generated for specific process calculi by ad hoc methods (cf., e.g., [79]).

### 5.5 RBB Safe Format

In order to abstract away from internal actions, Milner [153] introduced a special action $\tau$, called the *silent step*. The relation symbol $\xrightarrow{\tau}$ intuitively represents an internal computation. A number of equivalence notions have been developed to identify states in LTSs that incorporate silent steps, such as weak bisimulation [124] and branching bisimulation [104, 105]. In [51, 215, 218] rule formats have been introduced to ensure that weak and branching bisimulation equivalence are a congruence (cf. Def. 2.11). However, in general such equivalences are not a congruence with respect to most process algebras, because of the pre-emptive power of silent steps (see, e.g., [158, Sect. 2.3] for an intuitive discussion of this phenomenon). For this reason it has become standard practice to impose a rootedness condition on equivalences for the silent step [158].

Bloom [51] presented a rule format to ensure that rooted branching bisimulation is a congruence, imposing additional requirements on the GSOS format. The rule format recognizes so-called *patience* rules, via which a closed term can inherit the $\tau$-transitions of its arguments. The *RBB safe* format [84] relaxes some syntactic restrictions of Bloom's rule format, imposing additional requirements on the panth format. Notably, certain arguments of function symbols are labelled 'wild', and this labelling is used to restrict occurrences of variables in targets and in left-hand sides of premises. If a TSS is complete (cf. Def. 3.12) and satisfies the syntactic restrictions of the RBB safe format, then rooted branching bisimulation with respect to the associated LTS is a congruence.

**Rooted Branching Bisimulation** We assume that $\mathsf{Act}$ is extended with a special action $\tau$, representing the silent step. The reflexive and transitive closure of the relation $\stackrel{\tau}{\rightarrow}$ is denoted by $\stackrel{\varepsilon}{\rightarrow}$. First, we define the notion of branching bisimulation [104, 105].

**Definition 5.45 (Branching bisimulation)** *Assume an LTS. A binary relation $\mathcal{B}$ on states is a* branching bisimulation *if it is symmetric and, whenever $s_1\ \mathcal{B}\ s_2$:*

- *if $s_1 \stackrel{a}{\rightarrow} s_1'$, then either*

    1. *$a = \tau$ and $s_1'\ \mathcal{B}\ s_2$, or*
    2. *there are transitions $s_2 \stackrel{\varepsilon}{\rightarrow} s_2' \stackrel{a}{\rightarrow} s_2''$ such that $s_1\ \mathcal{B}\ s_2'$ and $s_1'\ \mathcal{B}\ s_2''$;*

- *if $s_1 P$, then there are transitions $s_2 \stackrel{\varepsilon}{\rightarrow} s_2' P$ such that $s_1\ \mathcal{B}\ s_2'$.*

*Two states $s_1, s_2$ are* branching bisimilar*, written $s_1 \underline{\leftrightarrow}_b\ s_2$, if there exists a branching bisimulation relation that relates them.*

Branching bisimulation is an equivalence relation; see [33]. However, branching bisimulation equivalence is not a congruence with respect to some standard operations in process algebras. For example, in $\mathrm{BPA}_{\epsilon\tau}$ (see Sect. 2.6) with constants $a$ an $c$, $a \underline{\leftrightarrow}_b\ a$ and $c \underline{\leftrightarrow}_b\ \tau c$, but $a + c$ and $a + \tau c$ are not branching bisimilar. Therefore, we introduce a rootedness condition.

**Definition 5.46 (Rooted branching bisimulation)** *Assume an LTS. A binary relation $\mathcal{R}$ on states is a* rooted branching bisimulation *if it is symmetric and, whenever $s_1\ \mathcal{R}\ s_2$,*

- *if $s_1 \stackrel{a}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{a}{\rightarrow} s_2'$ such that $s_1' \underline{\leftrightarrow}_b\ s_2'$;*
- *if $s_1 P$, then $s_2 P$.*

*Two states $s_1, s_2$ are* rooted branching bisimilar*, written $s_1 \underline{\leftrightarrow}_{rb}\ s_2$, if there exists a rooted branching bisimulation relation that relates them.*

Since branching bisimulation is an equivalence relation, it is easy to see that rooted branching bisimulation is also an equivalence relation.

**RBB Safe** We proceed to present a congruence format for rooted branching bisimulation equivalence from [84]. Let $C[]$ denote a context, viz. a term with one occurrence of the context symbol $[]$ (cf. Sect. 2.4). We assume that every argument of each function symbol is labelled either *tame* or *wild*. A context is *wild-nested* if the context symbol occurs inside a nested string of wild arguments.

**Definition 5.47 (Wild-nested context)** *The set of* wild-nested *contexts over a signature* $\Sigma$ *is defined inductively by:*

1. $[]$ *is wild-nested;*
2. *if* $C[]$ *is wild-nested, and argument* $i$ *of function symbol* $f \in \Sigma$ *is wild, then* $f(t_1, \ldots, t_{i-1}, C[], t_{i+1}, \ldots, t_{ar(f)})$ *is wild-nested.*

A patience rule for an argument $i$ of a function symbol $f$ implies that a closed term $f(t_1, \ldots, t_{ar(f)})$ inherits the $\tau$-transitions of its argument $t_i$.

**Definition 5.48 (Patience rule)** *A* patience *rule for the* $i$*th argument of a function symbol* $f$ *is a GSOS rule of the form*

$$\frac{x_i \xrightarrow{\tau} y}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{\tau} f(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{ar(f)})}$$

Now we are in a position to present the RBB safe format, which imposes additional restrictions on the panth format (cf. Def. 5.1).

**Definition 5.49 (RBB safe)** *A TSS* $T$ *in panth format is* RBB safe *if there exists a tame/wild labelling of arguments of function symbols such that each of its transition rules* $\rho$ *is*

1. *either a patience rule for a wild argument of a function symbol,*
2. *or a rule with source* $f(x_1, \ldots, x_{ar(f)})$ *for which the following requirements are fulfilled:*
   - *right-hand sides of positive premises do not occur in left-hand sides of premises of* $\rho$*;*
   - *if argument* $i$ *of* $f$ *is wild and does not have a patience rule in* $T$*, then* $x_i$ *does not occur in left-hand sides of premises of* $\rho$*;*
   - *if argument* $i$ *of* $f$ *is wild and has a patience rule in* $T$*, then* $x_i$ *occurs no more than once in the left-hand side of a premise of* $\rho$*, where this premise*
     - *is positive,*
     - *does not contain the relation symbol* $\xrightarrow{\tau}$*, and*
     - *has left-hand side* $x_i$*;*
   - *right-hand sides of positive premises and variables* $x_i$ *for* $i$ *a wild argument of* $f$ *only occur at wild-nested positions in the target of* $\rho$*.*

**Theorem 5.50** *If a complete TSS is RBB safe, then the rooted branching bisimulation equivalence that it induces is a congruence.*

See [84] for a string of examples of complete TSSs to show that all syntactic requirements of the RBB safe format are essential for the congruence result in Thm. 5.50.

**Computation of Tame/Wild Labels** The crux in determining whether a TSS $T$ is RBB safe is to find a suitable tame/wild labelling of arguments of function symbols. Assuming that the signature $\Sigma$ is finite, there exists an efficient procedure that computes a tame/wild labelling $\Lambda$ witnessing that $T$ is RBB safe if and only if one such a labelling exists.

> Procedure "Compute Wild Labels for $\Sigma$ and $T$":
>
> The red/green directed graph $G$ consists of vertices $\langle f, i\rangle$ for $f \in \Sigma$ and $1 \leq i \leq ar(f)$. There is an edge from $\langle f, i\rangle$ to $\langle g, j\rangle$ in $G$ iff there is a transition rule in $T$ with its conclusion of the form
>
> $$f(x_1, \ldots, x_{ar(f)}) \xrightarrow{a} C[g(t_1, \ldots, t_{j-1}, D[x_i], t_{j+1}, \ldots, t_{ar(g)})] \ .$$
>
> A vertex $\langle g, j\rangle$ is red iff there is a transition rule in $T$ with its target of the form
>
> $$C[g(t_1, \ldots, t_{j-1}, D[y], t_{j+1}, \ldots, t_{ar(g)})]$$
>
> where $y$ is the right-hand side of a positive premise of this rule. All other vertices in $G$ are coloured green.
>
> The procedure turns green vertices in $G$ red as follows. If a vertex $\langle f, i\rangle$ is red, and there exists an edge in $G$ from $\langle f, i\rangle$ to a green vertex $\langle g, j\rangle$, then $\langle g, j\rangle$ is coloured red.
>
> The procedure terminates if none of the green vertices can be coloured red anymore, at which point it outputs the red/green directed graph.

$\Lambda$ labels argument $i$ of function symbol $f$ 'wild' iff the vertex $\langle f, i\rangle$ in the output graph of the procedure above is red.

We proceed to apply Thm. 5.50 to two of the TSSs from Sect. 2.6, extended with the silent step.

**BPA with Empty Process and Silent Step** The process algebra $\mathrm{BPA}_{\epsilon\tau}$ is obtained from $\mathrm{BPA}_\epsilon$ by extending $\mathsf{Act}$ with the silent step $\tau$. The TSS for $\mathrm{BPA}_{\epsilon\tau}$ is the TSS for $\mathrm{BPA}_\epsilon$ in Table 1, with the proviso that $a$ ranges over $\mathsf{Act} \cup \{\tau\}$.

It was noted in Sect. 5.1 that the TSS in Table 1 is panth. The procedure to calculate a tame/wild labelling for TSS produces the following result: the first argument of sequential composition is wild (because of the target $x' \cdot y$ in the third transition rule for sequential composition), while both arguments of alternative composition and the second argument of sequential composition are tame. The TSS in Table 1, with $a$ ranging over $\mathsf{Act} \cup \{\tau\}$, is RBB safe with respect to this tame/wild labelling. As an example, for sequential composition we have that:

- its third transition rule with $a = \tau$ constitutes a patience rule for the first argument of sequential composition;
- in its first two transition rules, and in its third transition rule with $a \neq \tau$, the variable $x$ in the wild argument of the source occurs as the left-hand side of one positive premise, which does not contain the relation symbol $\xrightarrow{\tau}$;
- in its third transition rule, the variable $x'$ in the right-hand side of the premise occurs in a wild-nested position of the target.

It is left to the reader to verify that the remaining transition rules in Table 1 are RBB safe. It was proven in Sect. 3.5 that the TSS in Table 1 is complete. Hence, according to Thm. 5.50 rooted branching bisimulation equivalence is a congruence with respect to $\mathrm{BPA}_{\epsilon\tau}$.

**Priorities with Silent Step** In general, rooted branching bisimulation equivalence is not a congruence with respect to the priority operator. For example, suppose that $b < c$; then $a \cdot (\tau \cdot (b + c) + b)$ and $a \cdot (b + c)$ are rooted branching bisimilar, but $\theta(a \cdot (\tau \cdot (b + c) + b))$ and $\theta(a \cdot (b + c))$ are not rooted branching bisimilar. Consequently, in view of Thm. 5.50, the TSS for $\mathrm{BPA}_{\epsilon\tau\theta}$ in Table 2 (with $a$ and $b$ ranging over $\mathsf{Act} \cup \{\tau\}$) cannot be in the RBB safe format.

Since the second transition rule in Table 2 has target $\theta(x')$, the procedure in Sect. 5.5 labels the argument of $\theta$ wild. So, assuming there are one or more actions $b$ greater than action $a$ with respect to the ordering on $\mathsf{Act}$, the wild argument $x$ in the source of this transition rule occurs as the left-hand side of the negative premises $x \stackrel{b}{\nrightarrow}$. This violates the RBB safe format.

## 5.6 Precongruence Formats for Behavioural Preorders

The literature on SOS is particularly rich in results on (pre)congruence formats for behavioural equivalences and preorders, mostly in the absence of predicates. This section presents precongruence formats for simulation and ready simulation from [98], for readies preorder from [48], for failures preorder from [220], for accepting trace preorder from [83], and for trace preorder from [50, 220]. Vaandrager [220] moreover showed that the De Simone format constitutes a precongruence format for the external trace, external failure, and must [75, 118] preorders. van Glabbeek [98] sketched a congruence format for ready trace equivalence. We note that if a preorder is a precongruence, then its kernel is a congruence.

### 5.6.1 Simulation

Path (cf. Def. 5.2) is a precongruence format for simulation preorder (cf. Def. 2.3). (According to van Glabbeek [98], path is actually a congruence format for all $n$-nested simulation equivalences [111].)

**Theorem 5.51** *If a TSS is in path format, then the simulation preorder that it induces is a precongruence.*

For example, since the TSS for $\mathrm{BPA}_\epsilon$ from Sect. 2.6 is in path format, Thm. 5.51 implies that simulation preorder is a precongruence with respect to $\mathrm{BPA}_\epsilon$.

### 5.6.2 Ready Simulation

A precongruence format for ready simulation preorder (cf. Def. 2.3) is obtained by disallowing *look-ahead* in panth rules; i.e., right-hand sides of positive premises of a transition rule are not allowed to occur in left-hand sides of premises of this rule.

**Definition 5.52 (Ready simulation format)** *A panth rule is in* ready simulation *format if the variables at the right-hand sides of its positive premises do not occur in the left-hand sides of its premises.*

*A TSS is in* ready simulation *format if all its transition rules are.*

**Theorem 5.53** *If a complete TSS is in ready simulation format, then the ready simulation preorder that it induces is a precongruence.*

For example, the TSS for BPA$_\epsilon$ from Sect. 2.6 is positive and so complete. Furthermore, it is in path format, and none of its transition rules contains look-ahead. Hence, Thm. 5.53 implies that ready simulation preorder is a precongruence with respect to BPA$_\epsilon$.

The following counter-example shows that the omission of look-ahead from the ready simulation format is essential.

**Example:** Let $\mathsf{Act} = \{a, b, c\}$ and let $f$ be a unary function symbol. Extend the TSS for BPA$_\epsilon$ with

$$\frac{x \xrightarrow{a} y}{f(x) \xrightarrow{a} f(y)} \qquad \frac{x \xrightarrow{b} y \quad y \xrightarrow{c} z}{f(x)\surd}$$

Note that the premises of the second transition rule contain look-ahead.

Clearly, $a{\cdot}(b + b{\cdot}c) + a{\cdot}b$ and $a{\cdot}(b + b{\cdot}c)$ are ready simulation equivalent. However, $f(a{\cdot}(b + b{\cdot}c) + a{\cdot}b)$ and $f(a{\cdot}(b + b{\cdot}c))$ are not ready simulation equivalent. Namely, the transition $f(a{\cdot}(b + b{\cdot}c) + a{\cdot}b) \xrightarrow{a} f(b)$ can only be simulated by $f(a{\cdot}(b + b{\cdot}c)) \xrightarrow{a} f(b + b{\cdot}c)$, but $f(b)$ has no initial transitions while $f(b + b{\cdot}c)\surd$. $\square$

### 5.6.3 Readies

This section presents a precongruence format for readies preorder (cf. Def. 2.5) from [48], which re-uses the notion of a wild-nested context (cf. Def. 5.47).

**Definition 5.54 ($\mathcal{F}$-winterized)** *A GSOS language $T$ is* $\mathcal{F}$-winterized *if there exists a tame/wild labelling of arguments of function symbols such that for each of its transition rules $\rho$, with source $f(x_1, \ldots, x_{ar(f)})$, the following requirements are fulfilled:*

- *right-hand sides of positive premises do not occur in left-hand sides of premises of $\rho$;*
- *if argument $i$ of $f$ is wild and there is a positive premise $x_i \xrightarrow{a} y$ in $\rho$ where $y$ occurs in the target, then this is the only premise in $\rho$ to have $x_i$ as its left-hand side;*
- *right-hand sides of positive premises and variables $x_i$ for $i$ a wild argument of $f$ only occur at wild-nested positions in the target of $\rho$;*
- *the target of $\rho$ has no multiple occurrences of variables.*

**Theorem 5.55** *If a TSS is $\mathcal{F}$-winterized, then the readies preorder that it induces is a precongruence.*

Since the GSOS format does not cater for TSSs with predicates, we refrain from applying the above precongruence theorem to our running examples. The interested reader is referred to [48] for applications of Thm. 5.55, and for counter-examples to show the importance of each of the syntactic restrictions on $\mathcal{F}$-winterized TSSs.

### 5.6.4 Ready Traces

**Definition 5.56** *A panth rule $\rho$ is in* ready trace *format if:*

1. *right-hand sides of positive premises do not occur in left-hand sides of premises of $\rho$;*
2. *each pair of variables that occur at distinct positions in the target of $\rho$ are not connected in the symmetric closure of the variable dependency graph of the premises of $\rho$ (cf. Def. 5.4).*

*A TSS is in* ready trace *format if all its transition rules are.*

**Theorem 5.57** *If a complete TSS is in ready trace format, then the ready trace preorder that it induces is a precongruence.*

**Example:** The TSS for $\mathrm{BPA}_\epsilon$ from Sect. 2.6 is in path format and thus complete. Note that none of its transition rules contain look-ahead. Furthermore, the only non-variable target in this TSS, in the third transition rule for sequential composition, contains two variables $x'$ and $y$, which are not connected in the variable dependency graph $\{\langle x, x'\rangle\}$ of the premises of this rule. Hence, Thm. 5.57 implies that ready trace preorder is a precongruence with respect to $\mathrm{BPA}_\epsilon$. $\square$

The following counter-example shows that the non-connectedness restriction on variables in targets is essential for the ready trace format.

**Example:** Let $\mathsf{Act} = \{a, b, c, d\}$, $f$ a unary, and $g$ a binary function symbol. Extend the TSS for $\mathrm{BPA}_\epsilon$ with

$$\frac{x \stackrel{a}{\rightarrow} y}{f(x) \stackrel{a}{\rightarrow} f(y)} \qquad \frac{x \stackrel{b}{\rightarrow} y_1 \quad x \stackrel{b}{\rightarrow} y_2}{f(x) \stackrel{b}{\rightarrow} g(y_1, y_2)} \qquad \frac{x_1 \stackrel{c}{\rightarrow} y_1 \quad x_2 \stackrel{d}{\rightarrow} y_2}{g(x_1, x_2)\surd}$$

Note that the variables $y_1$ and $y_2$ in the target of the second transition rule are connected in the symmetric closure of the variable dependency graph of the premises of this rule.

Clearly, $a{\cdot}(b{\cdot}c + b{\cdot}d)$ and $a{\cdot}b{\cdot}c + a{\cdot}b{\cdot}d$ are ready trace equivalent. However, $f(a{\cdot}(b{\cdot}c + b{\cdot}d))$ and $f(a{\cdot}b{\cdot}c + a{\cdot}b{\cdot}d)$ are not ready trace equivalent: $\{a\}\,a\,\{b\}\,b\,\{\surd\}$ is a ready trace of $f(a{\cdot}(b{\cdot}c+b{\cdot}d))$, but not of $f(a{\cdot}b{\cdot}c+a{\cdot}b{\cdot}d)$. $\square$

### 5.6.5 Failures

Vaandrager [220] showed that the De Simone format (cf. Def. 5.7) is a precongruence format for failures preorder (cf. Def. 2.5). (van Glabbeek [98] sketched a more general congruence format for failures equivalence; that format, however, is flawed [103].)

**Theorem 5.58** *The failures preorder induced by a De Simone language is a precongruence.*

For example, the TSS for $\mathrm{BPA}_\epsilon$ from Sect. 2.6 is in De Simone format, so Thm. 5.58 implies that failures preorder is a precongruence with respect to $\mathrm{BPA}_\epsilon$.

The following counter-example shows that the restriction of the De Simone format that a variable cannot occur both as a left-hand side of a premise and in the target is essential for the failures format.

**Example:** Let $\mathsf{Act} = \{a, b, c\}$, while $f$ and $g$ are unary function symbols. Extend the TSS for $\mathrm{BPA}_\epsilon$ with

$$\frac{x \xrightarrow{a} y}{f(x) \xrightarrow{a} f(y)} \qquad \frac{x \xrightarrow{b} y}{f(x) \xrightarrow{b} g(x)} \qquad \frac{x \xrightarrow{c} y}{g(x) \xrightarrow{c} \epsilon}$$

Note that in the second transition rule, the variable $x$ occurs both as the left-hand side of the premise and in the target.

Clearly, $a{\cdot}(b+c)+a{\cdot}b+a{\cdot}c$ and $a{\cdot}b+a{\cdot}c$ are failures equivalent. However, $f(a{\cdot}(b + c) + a{\cdot}b + a{\cdot}c)$ and $f(a{\cdot}b + a{\cdot}c)$ are not failures equivalent: $abc\emptyset$ is a failure of $f(a{\cdot}(b + c) + a{\cdot}b + a{\cdot}c)$, but not of $f(a{\cdot}b + a{\cdot}c)$. $\square$

### 5.6.6 Accepting Traces

Similar to the RBB safe format, a precongruence format for accepting trace preorder (cf. Def. 2.5) from [83] is based on a tame/wild labelling of arguments of function symbols.

**Definition 5.59 (L cool)** *A TSS in path format is* L cool *if there exists a tame/wild labelling of arguments of function symbols such that each of its transition rules $\rho$ satisfies the following syntactic restrictions:*

- *each variable in $\rho$ that occurs in a wild argument of the source, or as the right-hand side of a premise, occurs exactly once either as the left-hand side of a premise or at a wild-nested position (see Def. 5.47) in the target of $\rho$;*
- *the variable dependency graph (see Def. 5.4) of the set of premises of $\rho$ does not contain an infinite forward chain of edges.*

**Theorem 5.60** *If a TSS is in L cool format, then the accepting trace preorder that it induces is a precongruence.*

The following counter-example shows that the L cool format cannot allow for an infinite forward chain of edges in the variable dependency graph of the premises of a transition rule.

**Example:** Let $\mathsf{Act} = \{a\}$, $f$ a unary function symbol, and $b$ and $c$ constants. Consider the TSS

$$\frac{}{b \xrightarrow{a} b} \qquad \frac{\{x_i \xrightarrow{a} x_{i+1} \mid i \in \mathbb{N}\}}{f(x_0)\surd}$$

Note that the edges $\langle x_i, x_{i+1}\rangle$ for $i \in \mathbb{N}$ in the variable dependency graph of the premises of the second transition rule form an infinite forward chain.

The terms $b$ and $c$ are accepting trace equivalent, because they both have no accepting traces. However, $f(b)$ and $f(c)$ are not accepting trace equivalent, as $f(b)$ has the accepting empty trace $\varepsilon$ while $f(c)$ has no accepting traces. □

See [83] for further examples of TSSs showing that all the syntactic requirements of the L cool format are essential for the precongruence result in Thm. 5.60. Similar to the procedure for the RBB safe format in Sect. 5.5, there exists an efficient procedure to compute a tame/wild labelling $\Lambda$ witnessing that $T$ is L cool if and only if such a labelling exists; see [83].

**Example:** The TSS for $\mathrm{BPA}_\epsilon$ from Sect. 2.6 is in path format, and for each transition rule the variable dependency graph of its premises does not contain an infinite forward chain of edges. Take the first argument of sequential composition to be wild, and the two arguments of alternative composition and the second argument of sequential composition to be tame. It is not hard to see, for the TSS of $\mathrm{BPA}_\epsilon$, that if a variable occurs in a wild argument of the source or as the right-hand side of a premise of a transition rule, then it occurs exactly once either as the left-hand side of a premise or at a wild-nested position in the target of this transition rule. Hence, Thm. 5.60 implies that accepting trace preorder is a precongruence with respect to $\mathrm{BPA}_\epsilon$. □

### 5.6.7 Traces

Bloom [50] formulated a congruence format for trace equivalence (cf. Def. 2.4), generalizing an earlier precongruence format for trace preorder by Vaandrager [220].

**Definition 5.61** *A finite TSS in tyft format (cf. Def. 5.2) is in* trace *format if each transition rule satisfies the following three restrictions:*

- *it contains finitely many premises;*
- *each variable occurs either as the right-hand side of a premise or in the source;*
- *no variable occurs more than once in the left-hand sides of the premises and in the target.*

**Theorem 5.62** *If a TSS is in trace format, then the trace equivalence that it induces is a congruence.*

The example in Sect. 5.6.5 shows that the trace format cannot allow multiple occurrences of variables at the left-hand sides of the premises and in the target. Namely, trace equivalence induced by the TSS in that example is not a congruence. For instance. $a{\cdot}(b+c)+a{\cdot}b+a{\cdot}c$ and $a{\cdot}b+a{\cdot}c$ are trace equivalent, but $f(a{\cdot}(b+c)+a{\cdot}b+a{\cdot}c)$ and $f(a{\cdot}b+a{\cdot}c)$ are not. Note that the variable $x$ in the second transition rule of that example occurs both as the left-hand side of the premise and in the target.

Although the definition of completed trace equivalence (cf. Def. 2.5) is closely related to that for (accepting) trace equivalence, the L cool and trace formats do not constitute congruence formats for completed trace equivalence. The following counter-example, featuring the *encapsulation* operator [37, 153], shows that one cannot really hope to formulate a general congruence format for completed trace equivalence (see also [220]).

**Example:** Let $\mathsf{Act} = \{a, b, c\}$, and add the unary encapsulation operator $\partial_{\{c\}}$ to BPA$_\epsilon$. The LTS for a term $\partial_{\{c\}}(t)$ is obtained from the LTS for $t$ by excluding all $c$-transitions. This is expressed by three transition rules for $\partial_{\{c\}}$, which are added to the TSS for BPA$_\epsilon$:

$$\frac{x \xrightarrow{a} y}{\partial_{\{c\}}(x) \xrightarrow{a} \partial_{\{c\}}(y)} \qquad \frac{x \xrightarrow{b} y}{\partial_{\{c\}}(x) \xrightarrow{b} \partial_{\{c\}}(y)} \qquad \frac{x\surd}{\partial_{\{c\}}(x)\surd}$$

Clearly, $a{\cdot}b + a{\cdot}c$ and $a{\cdot}(b + c)$ are completed trace equivalent. However, $\partial_{\{c\}}(a{\cdot}b + a{\cdot}c)$ and $\partial_{\{c\}}(a{\cdot}(b + c))$ are not completed trace equivalent: $a$ is a completed trace of $\partial_{\{c\}}(a{\cdot}b + a{\cdot}c)$, but not of $\partial_{\{c\}}(a{\cdot}(b + c))$. $\square$

### 5.7 Trace Congruences

One of the original aims for the development of the theory of GSOS languages was to characterize the observational congruence induced by a reasonable notion of CCS-like operations, under the assumption that what we can directly observe from the behaviour of a process is its set of completed traces (see Def. 2.5 for the definition of completed trace equivalence $\simeq_{CT}$). Intuitively, two closed terms $t$ and $u$ are *completed trace congruent* with respect to a TSS if, for every context $C[x]$, the completed traces of $C[t]$ and $C[u]$ in the associated LTS coincide. We proceed to formulate the notion of completed trace congruence induced by a rule format.

**Definition 5.63 (Completed trace congruence)** *Let $\mathcal{F}$ be some rule format and $T_0$ a TSS in $\mathcal{F}$ format over a signature $\Sigma_0$. Two closed terms $t$ and $u$ are* completed trace congruent *with respect to $T_0$ and $\mathcal{F}$, notation $t \simeq^{\mathcal{F}}_{CT} u$, if for every TSS $T_1$ in $\mathcal{F}$ format over some signature $\Sigma_1$ that can be added in an operationally conservative fashion to $T_0$ (cf. Def. 4.2), and for every context $C[]$ over $\Sigma_0 \oplus \Sigma_1$, the LTS associated with $T_0 \oplus T_1$ yields $C[t] \simeq_{CT} C[u]$.*

Bloom, Istrail, and Meyer [54, 55] characterized the completed trace congruence induced by the GSOS format in terms of the equivalence corresponding to the following sublanguage of the modal logic HML (cf. Def. 2.7).

**Definition 5.64 (Denial formula)** *The set $\mathcal{D}$ of* denial formulae *over* $\mathsf{Act}$ *is given by the following BNF grammar, with $a \in \mathsf{Act}$:*

$$\varphi ::= \ \mathsf{true} \mid \varphi_1 \land \varphi_2 \mid \langle a\rangle\varphi \mid \neg\langle a\rangle\mathsf{true} \ .$$

In particular, a state satisfies formula $\neg\langle a\rangle\mathsf{true}$ iff $a$ is not one of its initial actions. The equivalence relation $\sim_{\mathcal{D}}$ is defined over states in LTSs in a similar fashion as the equivalence relation $\sim_{\mathrm{HML}}$ (see Sect. 2.3): $s \sim_{\mathcal{D}} s'$ iff for all denial formulae $\varphi$ we have $s \models \varphi \iff s' \models \varphi$.

**Theorem 5.65** *Let $T$ be a GSOS language. The equivalence relation $\sim_{\mathcal{D}}$ induced by the LTS associated with $T$ coincides with the completed trace congruence $\simeq_{CT}^{\mathrm{GSOS}}$ with respect to $T$ and the GSOS format.*

The interested reader is referred to *op. cit.* and [111] for proofs of the above result, and further discussion. Here we limit ourselves to remarking that, perhaps surprisingly, negative premises do not add anything to the discriminating power of the GSOS format. In fact, the GSOS operators used in the aforementioned references for testing denial formulae do not make use of negative premises at all. Indeed, the use they make of copying in either the premises or the conclusions of transition rules is rather minimal. The reader might also recall that negative premises were not used in the coding of a universal 2-counter machine presented in Sect. 5.4.3.

Larsen and Skou [138] gave the following characterization of denial equivalence, which provides additional insight into the behavioural nature of the completed trace congruences induced by GSOS languages.

**Theorem 5.66** *In every finitely branching LTS (cf. Def. 2.2), two states are ready simulation equivalent (cf. Def. 2.3) iff they satisfy exactly the same denial formulae.*

Thus the GSOS completed trace congruence is the equivalence induced by the ready simulation preorder, which prompted the authors of [55] to coin the slogan "bisimulation can't be traced". The following result, due to Groote [108], shows that bisimulation equivalence can indeed be traced, provided that the power of negative premises offered by the pure ntyft/ntyxt format (cf. Def. 5.2 and Def. 5.4) is available.

**Theorem 5.67** *Assume a stratifiable TSS (cf. Def. 3.13) in pure ntyft/ntyxt format containing at least one constant in its signature. Then, for every pair of closed terms $t, u$,*

$$t \simeq_{CT}^{\text{pure ntyft/ntyxt}} u \iff t \sim_{\mathrm{HML}} u \ .$$

In view of Thm. 2.8 this means that the completed trace congruence induced by the pure ntyft/ntyxt format coincides with bisimulation equivalence if the LTS associated with the TSS in question is finitely branching.

The use of negative premises appears to be necessary in order to test for bisimulation equivalence. Indeed, Groote and Vaandrager [111] characterized the completed trace congruence induced by the tyft/tyxt format thus.

**Theorem 5.68** *Assume a TSS in pure tyft/tyxt format whose associated LTS is finitely branching. Then, for every pair of closed terms* $t, u$*,*

$$t \simeq_{CT}^{\text{pure tyft/tyxt}} u \iff t \text{ and } u \text{ are 2-nested simulation equivalent} .$$

We refer the interested reader to *op. cit.* (and elsewhere in this handbook) for the definition of 2-nested simulation equivalence, and for more details on completed trace congruences.

# 6 Many-Sorted Higher-Order Languages

This section presents a formal framework to describe TSSs in the style of Plotkin, allowing one to express many-sortedness, general binding mechanisms, and substitutions. Such variable binding mechanisms are widely used in SOS semantics for, e.g., concurrent and functional programming languages [32, 162, 178, 185, 197], the $\pi$-calculus [161], value-passing process algebras [109, 121, 122], process algebras with recursion [128, 158], and timed process algebra [86].

Several concepts in the setting of operational semantics with variable binding, which seem to be intuitively clear at first sight, turn out to be ambiguous when studied more carefully. In order to obtain a formal framework in which transition rules with a variable binding mechanism can be expressed rigorously, we distinguish between actual and formal variables, following conventions from programming languages, and formalize the binding construct $t[u/x]$ in transition rules. In many programming languages there are actual parameters and formal parameters. The formal parameters are used to define procedures or functions; the actual parameters are the "real" variables to be used in the main program. In the main program the formal parameters are bound to the actual parameters. When discussing procedures on a conceptual level, it is often useful to introduce a notational distinction between formal and actual parameters; see for instance [232]. A transition rule can be thought of as a procedure to establish a transition relation by means of substituting (actual) terms for the (formal) variables. The following example illustrates that it is useful to make a notational distinction between actual and formal variables.

**Example:** Consider the transition rule

$$\frac{y[z/x]P_1}{yP_2}$$

where $x, y, z$ are variables and $y[z/x]$ is a standard notation that binds the $x$ in $y$ and replaces it by $z$. Application of a substitution $\sigma$ to this transition rule yields

$$\frac{\sigma(y)[\sigma(z)/x]P_1}{\sigma(y)P_2}$$

$\square$

The example above highlights two matters.

1. The expression $y[z/x]$ is not a substitution (for then it would equal $y$), but a syntactic construct with a suggestive form, called a *substitution harness*. Only after application of a substitution $\sigma$, the result $\sigma(y)[\sigma(z)/x]$ can be evaluated to a term.

2. Substitutions only apply to part of the variables that occur in a transition rule. In order to distinguish such *formal* variables in a transition rule, they are marked with an asterisk.

Hence, the transition rule above takes the form

$$\frac{y^*[z^*/x]P_1}{y^*P_2}$$

The use of formal variables in SOS with variable binding was proposed in, e.g., [87, 129, 195].

The organization of this section is as follows. Sect. 6.1 introduces actual terms, while Sect. 6.2 introduces formal terms. Sect. 6.3 describes the framework for many-sorted higher-order SOS definitions. Finally, Sect. 6.4 explains how the operational conservative extension format from Thm. 4.4 carries over to this higher-order setting.

Binding mechanisms exist in many and diverse forms. Here, these mechanisms are described using a notational approach, based on [15], for the Nuprl proof development system [67]. An alternative formalism would of course be the $\lambda$-calculus [32].

## 6.1 The Actual World

We assume a set of sorts together with a countably infinite set $\mathsf{Var}$ of sorted actual variables. The actual world contains actual terms, actual substitutions, and so forth. Let $\vec{O}$ denote a sequence $O_1 \cdots O_k$, and $\vec{O}_i$ a sequence $O_{i1} \cdots O_{ik}$, with $k \in \mathbb{N}$.

**Definition 6.1 (Many-sorted higher-order signature)** *A* many-sorted higher-order signature $\Sigma$ *is a set of function symbols*

$$f : \vec{S}_1.S_1 \times \cdots \times \vec{S}_{ar(f)}.S_{ar(f)} \to S,$$

*where the $S_{ij}$, the $S_i$, and $S$ are sorts.*

The intuitive idea embodied by the above definition is that a function symbol $f$ denotes an operation that takes functions of type $\vec{S}_i \to S_i$ as arguments, and delivers a result of sort $S$.

**Definition 6.2 (Actual term)** *Let $\Sigma$ be a many-sorted higher-order signature. The collection $\mathbb{A}(\Sigma)$ of* actual terms *over $\Sigma$ is the least set satisfying:*

- *each actual variable from* $\mathsf{Var}$ *is in $\mathbb{A}(\Sigma)$;*
- *for each function symbol $f : \vec{S}_1.S_1 \times \cdots \times \vec{S}_{ar(f)}.S_{ar(f)} \to S$, the expression $f(\vec{x}_1.t_1, \ldots, \vec{x}_{ar(f)}.t_{ar(f)})$ is an actual term of sort $S$, if*
  - *the $t_i$ are actual terms of sort $S_i$, and*
  - *each sequence $\vec{x}_i$ consists of distinct actual variables in* $\mathsf{Var}$ *of sorts $\vec{S}_i$.*

Free occurrences of actual variables in actual terms are defined inductively as expected:

- $x$ occurs free in $x$ for each $x \in \mathsf{Var}$;
- if $x$ occurs free in $t_i$, and $x$ does not occur in the sequence $\vec{x}_i$, then $x$ occurs free in $f(\vec{x}_1.t_1, \ldots, \vec{x}_{ar(f)}.t_{ar(f)})$.

An actual term is *closed* if it does not contain any free occurrences of actual variables.

An *actual* substitution is a sort preserving mapping $\sigma : \mathsf{Var} \to \mathbb{A}(\Sigma)$, where sort preserving means that $x$ and $\sigma(x)$ are always of the same sort. An actual substitution extends to a mapping from actual terms to actual terms; the actual term $\sigma(t)$ is obtained by replacing free occurrences of actual variables $x$ in $t$ by $\sigma(x)$. As usual, $\_[t/x]$ is the postfix notation for the actual substitution that maps $x$ to $t$ and is inert otherwise. Such postfix denoted actual substitutions are called *explicit* (as opposed to *implicit* actual substitutions $\sigma$).

In the definition of actual substitutions on actual terms there is a well-known complication. Namely, consider an actual term $\sigma(t)$, and let $x$ occur free in $t$. After $x$ in $t$ has been replaced by $\sigma(x)$, actual variables $y$ that occur in $\sigma(x)$ are bound in actual subterms such as $f(y.u)$ of $t$. A solution for this problem, which originates from the $\lambda$-calculus, is to allow unrestricted substitution by applying $\alpha$-conversion; that is, by renaming bound actual variables. From now on, actual terms are considered modulo $\alpha$-conversion, and when an actual substitution is applied, bound actual variables are renamed. Stoughton [213] presented a clean treatment of this technique.

## 6.2 The Formal World

We argued that it is a good idea to distinguish between formal and actual variables when discussing transition rules with variable bindings and substitutions on an abstract level. A *formal* term $t^*$ is an actual term with possible occurrences of formal variables and substitution harnesses.

Assume a many-sorted higher-order signature $\Sigma$. The set $\mathsf{Var}^*$ of *formal* variables is defined as $\{x^* \mid x \in \mathsf{Var}\}$, where $x^*$ and $x$ are of the same sort.

**Definition 6.3 (Formal term)** *The collection $\mathbb{F}(\Sigma)$ of* formal terms *over a many-sorted higher-order signature $\Sigma$ is the least set satisfying:*

- *each actual variable from $\mathsf{Var}$ is in $\mathbb{F}(\Sigma)$;*
- *each formal variable from $\mathsf{Var}^*$ is in $\mathbb{F}(\Sigma)$;*
- *for each function symbol $f : \vec{S}_1.S_1 \times \cdots \times \vec{S}_{ar(f)}.S_{ar(f)} \to S$, the expression $f(\vec{x}_1.t_1^*, \ldots, \vec{x}_{ar(f)}.t_{ar(f)}^*)$ is a formal term of sort $S$, if*
  - *the $t_i^*$ are formal terms of sort $S_i$, and*
  - *each $\vec{x}_i$ consists of distinct actual variables in $\mathsf{Var}$ of sorts $\vec{S}_i$,*
- *if $t^*$ and $u^*$ are formal terms of sorts $S_0$ and $S_1$ respectively, and $x \in \mathsf{Var}$ is of sort $S_1$, then $t^*[u^*/x]$ is a formal term of sort $S_0$.*

A *formal* substitution is a sort preserving mapping $\sigma^* : \mathsf{Var}^* \to \mathbb{A}(\Sigma)$. It extends to a mapping $\sigma^* : \mathbb{F}(\Sigma) \to \mathbb{A}(\Sigma)$, where the actual term $\sigma^*(t^*)$

is obtained from the formal term $t^*$ as follows. First replace each formal variable $x^*$ in $t^*$ by $\sigma^*(x^*)$. Then the substitution harnesses in $t^*$ become explicit actual substitutions, so that the result evaluates to an actual term.

**Example:** An example of a formal term is $y^*[z^*/x]$, which evaluates to the actual constant $a$ after application of a formal substitution $\sigma^*$ with $\sigma^*(z^*) = a$ and $\sigma^*(y^*) = x$. Namely, the implicit formal substitution $\sigma^*$ turns the substitution harness $y^*[z^*/x]$ into the actual term $x[a/x]$, where $\_[a/x]$ is an explicit actual substitution, which evaluates to $a$. $\square$

We summarize the various notions of substitutions, and briefly discuss their differences. There are four notions in two worlds: implicit and explicit actual substitutions (which are semantically the same), formal substitutions, and substitution harnesses.

- Implicit actual substitutions $\sigma$ and explicit actual substitutions $\_[t/x]$ both denote mappings from actual variables to actual terms.
- Formal substitutions $\sigma^*$ are mappings from formal variables to actual terms.
- A substitution harness $t^*[u^*/x]$ is *not* a substitution, but a piece of syntax with a suggestive form. If a formal substitution $\sigma^*$ is applied to it, then the result is an expression $\sigma^*(t^*)[\sigma^*(u^*)/x]$, containing an explicit actual substitution, so that it can be evaluated to an actual term.

Substitution harnesses are used to formulate in a precise way how a formal substitution is to act on a transition rule. The formal and actual substitutions are used to move from transition rules to a proof tree (cf. Def. 2.14).

## 6.3 Actual and Formal Transition Rules

Before presenting the basic definitions of SOS for higher-order languages, we first consider as an example the recursive $\mu$-construct, which combines formal variables, a binding mechanism, and a substitution harness. The $\mu$-operator is similar to the construct $\mathrm{fix}(X = t)$ that was incorporated in De Simone languages (cf. Sect. 5.3.1).

**Example:** Intuitively, a closed actual term $\mu x.t$ executes $t$ until it encounters an expression $x$, in which case it starts executing $\mu x.t$ again. This intuition is expressed in the following transition rule, which we call the $\mu$-rule:

$$\frac{y^*[\mu x.y^*/x] \xrightarrow{a} z^*}{\mu x.y^* \xrightarrow{a} z^*}$$

The transition $\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x$ with $a{\cdot}\_$ the prefix multiplication operator from CCS can be derived from the $\mu$-rule together with the standard transition rule for prefix multiplication: $\varnothing/a{\cdot}w^* \xrightarrow{a} w^*$ (cf. Sect. 5.4.3). Namely, after application of the formal substitution $\sigma^*$ to the $\mu$-rule with $\sigma^*(y^*) = a{\cdot}x$ and $\sigma^*(z^*) = \mu x.a{\cdot}x$, the premise takes the form $a{\cdot}x[\mu x.a{\cdot}x/x] \xrightarrow{a} \mu x.a{\cdot}x$, which evaluates to $a{\cdot}\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x$. Since this is an instance of the transition rule for prefix multiplication, with $\mu x.a{\cdot}x$ for $w^*$, we conclude that the $\sigma^*$-instantiation of the conclusion of the $\mu$-rule is valid: $\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x$. The proof of $\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x$ is depicted below.

$$\begin{tikzcd}
\varnothing \arrow[d, "\text{instance of prefixing rule}"'] \\
a{\cdot}\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x \arrow[d, "\text{instance of } \mu\text{-rule}"] \\
\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x
\end{tikzcd}$$

□

**Definition 6.4 (Actual transition rule)** *An* actual transition rule *is an expression of the form $H/\alpha$, where $H$ is a set of literals (cf. Def. 2.13), and $\alpha$ is a positive literal.*

Actual transition rules are deduced by means of *formal* transition rules. The formal transition rules are the ones that are presented in the literature; they are the recipes that enable one to deduce an LTS. For instance, in the example above, the actual transition rule

$$\frac{a{\cdot}\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x}{\mu x.a{\cdot}x \xrightarrow{a} \mu x.a{\cdot}x}$$

was deduced from the $\mu$-rule, which is a formal transition rule.

**Definition 6.5 (Formal transition rule)** *A* formal transition rule *is an expression $H^*/\alpha^*$, where:*

- *$H^*$ is a set of premises of the form $t^* \xrightarrow{a} u^*$, $t^*P$, $t^* \stackrel{a}{\nrightarrow}$, and $t^*\neg P$;*
- *$\alpha^*$ is the conclusion of the form $t^* \xrightarrow{a} u^*$ or $t^*P$;*

*where $t^*, u^* \in \mathbb{F}(\Sigma)$, $a \in \mathsf{Act}$, and $P$ denotes any predicate. A* higher-order TSS *is a set of formal transition rules.*

We give an intricate example of a formal transition rule PRE from the $\pi$-calculus [161], which incorporates bound variables and parameterized transition labels. Recall that actual terms are considered modulo $\alpha$-conversion.

**Example:** Assume two sorts *Port* of port names and *Process* of processes. For actual variables $x$ and $y$ of sort *Port* we have the formal transition rule

$$\text{PRE} \qquad x(y).v^* \xrightarrow{x(y)} v^*$$

from [196], where $v^*$ is a formal variable of sort *Process*. The formal transition rule PRE expresses that process $x(y).t$ sends the bound port name $y$ via port $x$, and proceeds as process $t$. There is a subtle distinction between the two occurrences of $y$ in PRE; in $x(y).v^*$ it is a binder of $v^*$, while in the transition label it is a free parameter. A notation for PRE in the vein of this section would be

$$send(x, y.v^*) \xrightarrow{(x,y)} v^* \ .$$

From PRE we can deduce $x(y).t \xrightarrow{x(w)} t[w/y]$ for actual terms $t$ of sort *Process* that do not contain any free occurrences of the actual variable $w$ of sort *Port*, where $\_[w/y]$ is an explicit actual substitution. Namely, PRE yields $x(w).t[w/y] \xrightarrow{x(w)} t[w/y]$, and if $w$ does not occur free in $t$, then $x(w).t[w/y]$ is $\alpha$-convertible to $x(y).t$. $\square$

The example above shows that provability of an actual transition rule may depend in an essential way on the fact that actual terms are considered modulo $\alpha$-conversion.

## 6.4 Operational Conservative Extension

Only few rule formats for higher-order languages have appeared in the literature. Notably, Howe [129] presented a congruence format for higher-order TSSs with respect to bisimulation equivalence, which shows a strong resemblance with the tyft/tyxt format from Groote and Vaandrager (cf. Def. 5.2).

Interestingly, Bernstein [42] showed that in many cases the semantics of a higher-order language can be captured by a first-order TSS with terms as transition labels. It appears that existing rule formats can be extended with terms as transition labels in a straightforward manner; see [42, 87]. Thus, it might be the case that current first-order rule formats are sufficient to deal with higher-order languages. Another reference of interest in this area is [49].

We proceed to present a generalization from [87] of the operational conservative extension format (see Sect. 4) to a higher-order setting. This generalization is based on an adaptation of the notion of source-dependency (cf. Def. 4.3), requiring a distinction between occurrences of formal variables in- and outside the substitution harnesses of a formal term. $FV(t^*)$ denotes the set of formal variables that occur in the formal term $t^*$.

**Definition 6.6** **($FV(t^*)$)** *The sets $FV(t^*)$ are defined inductively by:*

$$\begin{array}{rcl} FV(x^*) & \triangleq & x^* \\ FV(f(\vec{x}_1.t_1^*,\ldots,\vec{x}_{ar(f)}.t_{ar(f)}^*)) & \triangleq & FV(t_1^*)\cup\cdots\cup FV(t_{ar(f)}^*) \\ FV(t^*[s^*/x]) & \triangleq & FV(t^*)\cup FV(s^*) \ . \end{array}$$

For example, $FV(f(v.x^*[y^*/w])) = \{x^*, y^*\}$. By contrast, $EV(t^*)$ denotes a more restricted set of formal variables in the formal term $t^*$, which does not take into account formal variables that occur inside a substitution harness.

**Definition 6.7** **($EV(t^*)$)** *The sets $EV(t^*)$ are defined inductively by:*

$$\begin{array}{rcl} EV(x^*) & \triangleq & x^* \\ EV(f(\vec{x}_1.t_1^*,\ldots,\vec{x}_{ar(f)}.t_{ar(f)}^*)) & \triangleq & EV(t_1^*)\cup\cdots\cup EV(t_{ar(f)}^*) \\ EV(t^*[s^*/x]) & \triangleq & EV(t^*) \ . \end{array}$$

For example, $EV(f(v.x^*[y^*/w])) = \{x^*\}$. The sets $FV(t^*)$ and $EV(t^*)$ are used in the definition of source-dependent variables in a formal transition rule.

**Definition 6.8 (Source dependency)** *For a formal transition rule $\rho^*$, the formal variables in $\rho^*$ that are* source-dependent *are defined inductively by:*

1. *if $t^*$ is the source of $\rho^*$, then all formal variables in $EV(t^*)$ are source-dependent in $\rho^*$;*

2. *if $t^* \xrightarrow{a} u^*$ is a premise of $\rho^*$, and all formal variables in $FV(t^*)$ are source-dependent in $\rho^*$, then all formal variables in $EV(u^*)$ are source-dependent in $\rho^*$.*

*A formal transition rule $\rho^*$ is* source-dependent *if so are all the variables in $FV(\rho^*)$.*

Thm. 6.9 formulates sufficient criteria for a higher-order TSS $T_0 \oplus T_1$ to be an operational conservative extension of $T_0$ (see Def. 4.2); it extends Thm. 4.4 to higher-order languages. We say that a formal term in $\mathbb{F}(\Sigma_1)$ is *fresh* if it contains a function symbol from $\Sigma_1 \backslash \Sigma_0$ outside its substitution harnesses. Similarly, an action or predicate symbol in $T_1$ is *fresh* if it does not occur in $T_0$.

**Theorem 6.9** *Let $T_0$ and $T_1$ be higher-order TSSs over many-sorted higher-order signatures $\Sigma_0$ and $\Sigma_0 \oplus \Sigma_1$, respectively. Under the following conditions, $T_0 \oplus T_1$ is an operational conservative extension of $T_0$.*

1. *Each $\rho^* \in T_0$ is source-dependent.*
2. *For each $\rho^* \in T_1$,*
   - *either the source of $\rho^*$ is fresh,*
   - *or $\rho^*$ has a premise of the form $t_0^* \xrightarrow{a} t_1^*$ or $t_0^* P$, where*
     - *$t_0^* \in \mathbb{F}(\Sigma_0)$;*
     - *$FV(t_0^*) \subseteq EV(u^*)$, where $u^*$ denotes the source of $\rho^*$;*
     - *$t_1^*$, $a$, or $P$ is fresh.*

Theorem 6.9 can be applied to extensions of higher-order TSSs such as process algebra with time [86, 167] or data [109], where binding constructs enable one to parameterize processes over the time or data domain, process algebra with a recursive operator like the $\mu$-construct [118, 122, 197], the $\pi$-calculus [161, 196], and the lazy $\lambda$-calculus [120, 195].

# 7 Denotational Semantics

Following a bias towards operational methods in process theory that dates back to Milner's original development of the theory of CCS [153, 160], most of the work in the field of the meta-theory of process description languages reported in this chapter is concerned with operational and axiomatic semantics for terms and the relationships between the two. In particular, it is by now clear that it is often possible to automatically translate an operational theory of processes into an axiomatic one [8]. Moreover, in certain circumstances it is also possible to derive an SOS semantics from an axiomatic one, as witnessed by the developments in [8, 132].

Axiomatic semantics and proof systems for programming and specification languages are often closely related to denotational semantics for them, particularly if the Scott-Strachey approach [198] is followed. A paradigmatic example of the development of a semantic theory of processes in which operational, axiomatic, and denotational semantics coexist harmoniously, and may be used to highlight different aspects of process behaviours, is the theory of testing equivalence developed by De Nicola and Hennessy [75, 118]. In this theory, a process can be characterized operationally by means of its reaction to experiments, and denotationally as an *acceptance tree* [117]. Acceptance trees allow one to fully describe the behaviour of a process while abstracting away from the operational details of its interactions with all the possible testers. Moreover, the domain-theoretic properties of this model allow one to establish properties of the behavioural semantics that would be very difficult to derive using purely operational methods (see, e.g., the results in [118, Sect. 4.5].)

To our mind, the coincidence of operational, axiomatic, and denotational semantics enjoyed by the theory of processes presented in [118] does not only reinforce the naturalness of the chosen notion of program semantics, but allows one to make good use of the complementary benefits afforded by these semantic descriptions in establishing properties of processes. However, developing these three views of processes for each process description language from scratch and proving their coincidence is hard, subtle work; in addition, to quote from [149], giving denotational semantics to programming languages using the Scott-Strachey approach "involves an armamentarium of mathematical weapons otherwise unfamiliar in Computer Science". It would therefore be beneficial to develop systematic ways of giving denotational semantics to process description languages, following the Scott-Strachey approach, starting from their SOS descriptions. Of course, this is only worthwhile if the denotational semantics produced by the proposed techniques is automatically guaranteed to be in agreement with the behavioural and axiomatic views of processes. In particular, we would like to generate a denotational semantics that matches exactly our operational intuition about process behaviour, i.e., that is *fully abstract* in the sense of Milner and Plotkin [151, 152, 175, 212], with respect to a reasonable notion of behavioural semantics.

This section reviews results from [11, 12], where it is shown how to generate a fully abstract denotational semantics from a class of recursive GSOS languages (cf. Sect. 5.4.5). Usually denotational semantics deals with recursion implicitly, by setting up a framework within which reasoning about recursion can be reduced to reasoning about recursion-free approximations. In line with this approach, a denotational semantics is first given to recursion-free terms from GSOS languages. Next, recursion-free approximations are used to extend this result to recursive terms over recursive GSOS languages.

## 7.1 Preliminaries

### 7.1.1 $\Sigma$-Domains

We assume familiarity with the basic notions of ordered and continuous algebras (see, e.g., [112, 118, 130]); however, in what follows we give a quick overview of the way a denotational semantics can be given to a recursive language following the standard lines of algebraic semantics [112]. The interested reader is invited to consult [118] for an explanation of the theory.

Let $\Sigma$ denote a signature, in the sense of Sect. 2.4, which includes a distinguished constant $\Omega$. A $\Sigma$-*algebra* consists of a carrier set $\mathcal{A}$, where for each function symbol $f \in \Sigma$ there is given an operator $f_{\mathcal{A}} : \mathcal{A}^{ar(f)} \to \mathcal{A}$. A mapping $\varphi : \mathcal{A} \to \mathcal{B}$ between two $\Sigma$-algebras is a $\Sigma$-*homomorphism* if for every $f \in \Sigma$ and elements $d_1, \ldots, d_{ar(f)} \in \mathcal{A}$:

$$\varphi(f_{\mathcal{A}}(d_1, \ldots, d_{ar(f)})) \quad = \quad f_{\mathcal{B}}(\varphi(d_1), \ldots, \varphi(d_{ar(f)})) \ .$$

We recall (from Sect. 2.4) that $\mathrm{T}(\Sigma)$ denotes the set of closed terms over $\Sigma$. We use $\mathrm{T}(\Sigma, \mathsf{RVar})$ to denote the set of closed terms over $\Sigma$ that may contain occurrences from a countably infinite set $\mathsf{RVar}$ of recursion variables, ranged over by $X, Y$.

A $\Sigma$-*domain* [118] $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ is a $\Sigma$-algebra whose carrier $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ is an algebraic complete partial order (cpo) (see, e.g., [176]) and whose operations are interpreted as continuous functions (in the sense of, e.g., [118, p. 123]). We require that $\Omega$ is $\perp_{\mathcal{A}}$, viz. the least element in the algebraic cpo $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$. The notion of a $\Sigma$-*poset* (resp. $\Sigma$-*preorder*) may be defined in a similar way by requiring that $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ be a partially ordered (resp. preordered) set and that the operators be monotonic. The notion of $\Sigma$-homomorphism extends to the ordered $\Sigma$-structures in the obvious way by requiring that such maps preserve the underlying order-theoretic structure as well as the $\Sigma$-structure.

The interpretation $\mathcal{A}[\![\cdot]\!]$ of $\mathrm{T}(\Sigma, \mathsf{RVar})$ in a $\Sigma$-algebra $\mathcal{A}$ associates each term in $\mathrm{T}(\Sigma, \mathsf{RVar})$ with a mapping from substitutions (going from recursion variables to $\mathcal{A}$) to $\mathcal{A}$. This interpretation is defined by induction as follows, where $\sigma$ is any mapping from recursion variables to $\mathcal{A}$:

$$\begin{aligned} \mathcal{A}[\![X]\!]\sigma &\triangleq \sigma(X) \\ \mathcal{A}[\![f(t_1,\ldots,t_n)]\!]\sigma &\triangleq f_{\mathcal{A}}(\mathcal{A}[\![t_1]\!]\sigma,\ldots,\mathcal{A}[\![t_n]\!]\sigma) \ , \end{aligned}$$

where $n = ar(f)$. We recall that the recursive terms over $\Sigma$ are given by the BNF grammar

$$t ::= X \mid f(t_1,\ldots,t_{ar(f)}) \mid \mathrm{fix}(X = t)$$

where $X$ is any recursion variable, $f$ any function symbol in $\Sigma$, and fix a binding construct. The latter construct gives rise to the usual notions of free and bound recursion variables in recursive terms. $\mathsf{CREC}(\Sigma)$ denotes the set of recursive terms that do not contain free recursion variables. Furthermore, $t[u/X]$ denotes the recursive term $t$ in which each occurrence of the recursion variable $X$ has been replaced by $u$. If $\mathcal{A}$ is a $\Sigma$-domain, then the interpretation $\mathcal{A}[\![\cdot]\!]$ extends to the set of recursive terms over $\Sigma$ by

$$\mathcal{A}[\![\mathrm{fix}(X = t)]\!]\sigma \ \triangleq\ \mathsf{Y}\lambda d.\, \mathcal{A}[\![t]\!]\sigma'$$

where $\mathsf{Y}$ denotes the least fixed-point operator, $d$ is a metavariable ranging over $\mathcal{A}$, $\sigma'(X) \triangleq d$, and $\sigma'(Y) \triangleq \sigma(Y)$ for $Y \neq X$. Note that for each $t \in \mathsf{CREC}(\Sigma)$, $\mathcal{A}[\![t]\!]\sigma$ does not depend on $\sigma$.

In what follows, we make use of some general results about the semantic mappings defined above, which may be found in [68, 112, 118]. The first result states that for any recursive term $t$ (possibly containing free recursion variables) there is a sequence of *finite approximations* $t_n \in \mathrm{T}(\Sigma, \mathsf{RVar})$ for $n \in \mathbb{N}$ such that, for any $\Sigma$-domain $\mathcal{A}$,

$$\mathcal{A}[\![t]\!] \ =\ \bigsqcup_{n\in\mathbb{N}} \mathcal{A}[\![t_n]\!] \ .$$

(That is, the interpretation of the term $t$ in $\mathcal{A}$ is the least upper bound of the interpretations of its finite approximations.) The second result states that if $\leq_\Omega$ is the least precongruence (cf. Def. 2.11) satisfying

$$\begin{aligned} \mathrm{fix}(X = t) &\leq_\Omega t[\mathrm{fix}(X = t)/X] \\ t[\mathrm{fix}(X = t)/X] &\leq_\Omega \mathrm{fix}(X = t) \\ \Omega &\leq_\Omega X \end{aligned}$$

then $t_n \leq_\Omega t$ for every $n \in \mathbb{N}$.

For any binary relation $\mathcal{R}$ over $\mathsf{CREC}(\Sigma)$, the algebraic part of $\mathcal{R}$, denoted by $\mathcal{R}^A$, is defined as follows [118]:

$$t \,\mathcal{R}^A\, u \quad \Leftrightarrow \quad \forall n \in \mathbb{N}\ \exists m \in \mathbb{N}\ (t_n \,\mathcal{R}\, u_m) \ .$$

We say that $\mathcal{R}$ is *algebraic* if $\mathcal{R}$ is equal to $\mathcal{R}^A$. Intuitively, a relation is algebraic if it is completely determined by how it behaves on recursion-free terms. Every denotational interpretation $\mathcal{A}[\![\cdot]\!]$ induces a preorder $\sqsubseteq_{\mathcal{A}}$ over $\mathsf{CREC}(\Sigma)$ by:

$$t \sqsubseteq_{\mathcal{A}} u \quad \Leftrightarrow \quad \mathcal{A}[\![t]\!] \sqsubseteq_{\mathcal{A}} \mathcal{A}[\![u]\!] \ .$$

The following result characterizes a class of denotational interpretations which induce relations over terms that are algebraic.

**Lemma 7.1** *Let $\mathcal{A}$ be a $\Sigma$-domain. If $\mathcal{A}[\![t]\!]$ is a compact element in $\mathcal{A}$ for every $t \in \mathrm{T}(\Sigma)$ (see,* e.g., *[118, p. 130]), then $\sqsubseteq_{\mathcal{A}}$ is algebraic.*

In view of the above general lemma, the relations over the recursive terms in $\mathsf{CREC}(\Sigma)$ induced by a denotational semantics are always algebraic, provided that the denotations of the recursion-free terms in $\mathrm{T}(\Sigma)$ are compact elements in the algebraic cpo $\mathcal{A}$.

### 7.1.2 Prebisimulation

We consider LTSs with divergence from Def. 5.41, which include a special divergence predicate $\uparrow$. The convergence predicate $\downarrow$ holds in a state iff the divergence predicate does not hold in this same state (cf. Def. 5.42). The behavioural relation over LTSs with divergence that we study in this section is that of *prebisimulation* [116, 126, 154, 231] (also known as *partial bisimulation* [1]). We recall (from Def. 2.1) that $\mathsf{Proc}$ and $\mathsf{Act}$ denote the sets of states and actions, respectively, of the LTS with divergence in question. Let $\mathsf{Rel}(\mathsf{Proc})$ denote the set of binary relations over $\mathsf{Proc}$.

**Definition 7.2 (Prebisimulation)** *Assume an LTS with divergence. The functional $G : \mathsf{Rel}(\mathsf{Proc}) \to \mathsf{Rel}(\mathsf{Proc})$ is defined as follows. Given a relation $\mathcal{R} \in \mathsf{Rel}(\mathsf{Proc})$, we have $s_1\, G(\mathcal{R})\, s_2$ whenever:*

- *if $s_1 \xrightarrow{a} s_1'$, then there is a transition $s_2 \xrightarrow{a} s_2'$ such that $s_1' \,\mathcal{R}\, s_2'$;*

- *if $s_1 \downarrow$, then $s_2 \downarrow$;*

- *if* $s_1 \downarrow$ *and* $s_2 \xrightarrow{a} s_2'$, *then there is a transition* $s_1 \xrightarrow{a} s_1'$ *such that* $s_1' \mathcal{R} s_2'$.

*A relation* $\mathcal{R}$ *is a* prebisimulation *iff* $\mathcal{R} \subseteq G(\mathcal{R})$. *We write* $s_1 \lesssim s_2$ *if there exists a prebisimulation* $\mathcal{R}$ *such that* $s_1 \mathcal{R} s_2$.

The relation $\lesssim$ is a preorder over $\mathsf{Proc}$; its kernel is denoted by $\simeq$. Prebisimulation is similar in spirit to the notion of bisimulation (cf. Def. 2.3). Intuitively, $s_1 \lesssim s_2$ if the behaviour of $s_2$ is at least as specified as that of $s_1$, and $s_1$ and $s_2$ can simulate each other when restricted to the part of their behaviour that is fully specified. A divergent state $s$ with no outgoing transition intuitively corresponds to a process whose behaviour is totally unspecified — essentially an operational version of the bottom element $\perp$ in Scott's theory of domains [176, 198, 211].

The following precongruence result for $\lesssim$ with respect to recursive GSOS languages including the inert constant $\Omega$ originates from [11, 12].

**Proposition 7.3** $\lesssim$ *is a precongruence with respect to the LTS with divergence over* $\mathsf{CREC}(\Sigma)$ *associated with a recursive GSOS language including the inert constant* $\Omega$ *(cf. Prop. 5.43).*

### 7.1.3 Finite Synchronization Trees

A useful source of examples for LTSs with divergence is the set of *finite synchronization trees* [153].

**Definition 7.4 (Finite synchronization tree)** *The set of* finite synchronization trees *over a set of actions* $\mathsf{Act}$, *denoted by* $\mathsf{ST(Act)}$, *is defined inductively by:*

1. $\varnothing \in \mathsf{ST(Act)}$;
2. *if* $S \in \mathsf{ST(Act)}$, *then* $S \cup \{\perp\} \in \mathsf{ST(Act)}$;
3. *if* $a_1, \ldots, a_n \in \mathsf{Act}$ *and* $S_1, \ldots, S_n \in \mathsf{ST(Act)}$, *then*

$$\{\langle a_1, S_1\rangle, \ldots, \langle a_n, S_n\rangle\} \in \mathsf{ST(Act)} \ .$$

The symbol $\perp$ is used to represent that a finite synchronization tree is divergent. The set of finite synchronization trees $\mathsf{ST(Act)}$ can be turned into an LTS with divergence by stipulating that, for $S \in \mathsf{ST(Act)}$:

- $S \uparrow$ iff $\perp \in S$;

- $S \xrightarrow{a} S'$ iff $\langle a, S' \rangle \in S$.

When relating behavioural semantics based upon bisimulation-like relations with denotational semantics (usually based upon algebraic domains), one is faced with the following mismatch:

1. on the one hand, in an algebraic domain $d \sqsubseteq e$ iff every compact element smaller than or equal to $d$ is also dominated by $e$;
2. on the other hand, there are closed terms that have the same finite approximations, but that are not bisimilar.

This implies that bisimulation is not algebraic, and thus cannot be captured in a standard domain theoretic framework. One way to address this problem is to study its finitary part. To this end, it is generally agreed upon in the literature that finite synchronization trees are a natural operational counterpart of compact elements.

In what follows, we are interested in relating the notion of prebisimulation to a preorder on finite synchronization trees induced by a denotational semantics given by means of an algebraic domain. As such preorders are completely determined by how they act on finite processes, we are interested in comparing them with the "finitely observable", or finitary, part of the bisimulation in the sense of, e.g., [112, 116]. The following definition from [1] is inspired by property 1 above for algebraic domains.

**Definition 7.5 (Finitary preorder)** *The* finitary preorder $\lesssim^F$ *is defined on any LTS by*

$$s_1 \lesssim^F s_2 \iff \forall S \in \mathsf{ST}(\mathsf{Act})\,(S \lesssim s_1 \Rightarrow S \lesssim s_2) \ .$$

Since it is, in general, technically difficult to work with $\lesssim^F$, it is common practice to try and obtain an alternative characterization of the finitary preorder (see, e.g., [13]). An alternative method for using the functional $G : \mathsf{Rel}(\mathsf{Proc}) \to \mathsf{Rel}(\mathsf{Proc})$ from Def. 7.2 to obtain a preorder is to apply it inductively as follows:

- $\lesssim_0 \triangleq \mathsf{Proc} \times \mathsf{Proc}$
- $\lesssim_{n+1} \triangleq G(\lesssim_n)$

and finally $\lesssim_\omega \triangleq \bigcap_{n \in \mathbb{N}} \lesssim_n$. Intuitively, the preorder $\lesssim_\omega$ is obtained by restricting the prebisimulation relation to observations of finite depth. The preorders $\lesssim$, $\lesssim_\omega$, and $\lesssim^F$ are related thus:

$$\lesssim \ \subseteq \ \lesssim_\omega \ \subseteq \ \lesssim^F \ .$$

Moreover the inclusions are, in general, strict. The interested reader is referred to [1] for a wealth of examples distinguishing these preorders, and for a thorough analysis of their general relationships and properties.

The following comparison of preorders with respect to recursive GSOS languages including the inert constant $\Omega$ originates from [11, 12].

**Proposition 7.6** *The preorders $\lesssim^F$ and $\lesssim_\omega$ coincide over the LTS with divergence associated with a recursive GSOS language including the inert constant $\Omega$ (cf. Prop. 5.43).*

### 7.1.4 A Domain of Synchronization Trees

The canonical domain used in [11, 12] to give a denotational semantics to a class of recursive GSOS languages is the domain of *synchronization trees* over a countably infinite set $\mathsf{Act}$ of actions, as considered by Abramsky [1]. This is defined to be the initial solution $\mathcal{D}$ (in the category **SFP**, cf. [174]) of the domain equation

$$D = (\mathbf{1})_\perp \oplus P[\sum_{a \in \mathsf{Act}} D]$$

where $(\_)_\perp$ denotes lifting, $\mathbf{1}$ a one-point domain used to model $\mathbf{0}$, $\oplus$ the coalesced sum, $\sum$ a separated sum, and $P[\_]$ the Plotkin powerdomain construction (see [174, 176] for details on these domain theoretic operations). Intuitively one constructs the least fixed-point $\mathcal{D}$ of the domain equation above by starting with the one-point domain $\mathbf{1}$, and at the $n$th iterative step building the finite synchronization trees of height $n$.

To streamline the presentation we abstract away from the domain theoretic description of $\mathcal{D}$ given by the domain equation above. Our description of the domain of synchronization trees $\mathcal{D}$ follows the one given in [130], and we rely on results presented in that reference showing how to construct $\mathcal{D}$ starting from a suitable preorder on the set of finite synchronization trees $\mathsf{ST}(\mathsf{Act})$. The reconstruction of $\mathcal{D}$ is given in three steps.

1. First, we define a preorder $\sqsubseteq$ on the set of finite synchronization trees $\mathsf{ST}(\mathsf{Act})$. (This preorder is a reformulation of the Egli-Milner preorder over $\mathsf{ST}(\mathsf{Act})$ presented in [130]; see Prop. 7.8.)

2. Next, we relate the poset of compact elements of $\mathcal{D}$ to the poset of equivalence classes induced by $(\mathsf{ST}(\mathsf{Act}), \sqsubseteq)$.

3. Finally, we use the fact that $\mathcal{D}$ is the ideal completion of its poset of compact elements to relate it to $(\mathsf{ST}(\mathsf{Act}), \sqsubseteq)$.

This approach allows us to factor the definition of the continuous algebra structure [106, 112, 118] on $\mathcal{D}$ in three similar steps.

**Definition 7.7 (ST-preorder)** *$\sqsubseteq$ is the least binary relation over* $\mathsf{ST}(\mathsf{Act})$ *such that the following conditions are satisfied if $S_1 \sqsubseteq S_2$:*

1. *if $\langle a, S_1' \rangle \in S_1$, then $S_1' \sqsubseteq S_2'$ for some $\langle a, S_2' \rangle \in S_2$;*
2. *if $\perp \in S_2$, then $\perp \in S_1$;*
3. *if $\langle a, S_2' \rangle \in S_2$, then either $\perp \in S_1$ or $S_1' \sqsubseteq S_2'$ for some $\langle a, S_1' \rangle \in S_1$.*

The relation $\sqsubseteq$ so defined is easily seen to be a preorder over $\mathsf{ST}(\mathsf{Act})$. Moreover, it coincides with $\lesssim$ over $\mathsf{ST}(\mathsf{Act})$. We proceed to relate the preorder $(\mathsf{ST}(\mathsf{Act}), \sqsubseteq)$ with the poset of compact elements of $\mathcal{D}$ in a way that allows us to define, in a canonical way, continuous operations on $\mathcal{D}$ from monotonic ones on $(\mathsf{ST}(\mathsf{Act}), \sqsubseteq)$.

First of all, we recall from [1] that $\mathcal{D}$ is, up to isomorphism, the algebraic cpo whose poset of compact elements $(\mathcal{K}(\mathcal{D}), \sqsubseteq_{\mathcal{K}(\mathcal{D})})$ is given as follows.

- $\mathcal{K}(\mathcal{D})$ is defined inductively by:
  - $\varnothing \in \mathcal{K}(\mathcal{D})$
  - $\{\perp\} \in \mathcal{K}(\mathcal{D})$
  - $a \in \mathsf{Act} \wedge d \in \mathcal{K}(\mathcal{D}) \quad \Rightarrow \quad \{\langle a, d \rangle\} \in \mathcal{K}(\mathcal{D})$
  - $d, e \in \mathcal{K}(\mathcal{D}) \quad \Rightarrow \quad \mathsf{Con}(d \cup e) \in \mathcal{K}(\mathcal{D})$, where $\mathsf{Con}$ denotes the convex closure operation (see, e.g., [1, p. 170]).
- $\sqsubseteq_{\mathcal{K}(\mathcal{D})}$ is defined by:

  $$d \sqsubseteq_{\mathcal{K}(\mathcal{D})} e \quad \Leftrightarrow \quad d = \{\perp\} \vee d \sqsubseteq_{EM} e$$

  where $\sqsubseteq_{EM}$ denotes the Egli-Milner preorder (see, e.g., [1, Def. 3.3]).

From the above definitions it follows that $\mathcal{K}(\mathcal{D})$ is a subset of $\mathsf{ST}(\mathsf{Act})$. Hence it makes sense to compare the relations $\sqsubseteq$ and $\sqsubseteq_{\mathcal{K}(\mathcal{D})}$ over it. The following result from [11, 12] lends credence to our previous claims.

**Proposition 7.8** *For all $d, e \in \mathcal{K}(\mathcal{D})$, $d \sqsubseteq e$ iff $d \sqsubseteq_{\mathcal{K}(\mathcal{D})} e$.*

As a consequence of this result, to ease the presentation of the technical results to follow, from now on we use $\sqsubseteq$ as our notion of preorder on $\mathcal{K}(\mathcal{D})$.

## 7.2 From Recursive GSOS to Denotational Semantics

We now characterize a class of GSOS languages, incorporating the inert constant $\Omega$ for which there are no transition rules, that map finite LTSs to finite LTSs (cf. Def. 2.2). The semantic counterparts of these function symbols have the property of being compact in the sense of [130], i.e., of mapping compact elements in the Plotkin powerdomain of synchronization trees to compact elements. In view of Lem. 7.1, denotational interpretations for the resulting languages induce preorders over terms that are algebraic.

**Definition 7.9 (Compact GSOS)** *A GSOS language including the inert constant $\Omega$ is* compact *if it is linear (cf. Def. 5.35) and syntactically well-founded (cf. Def. 5.36).*

For each function symbol $f$ we introduce a mapping $\mathbf{f}_{\mathsf{ST}}$, mapping $ar(f)$ finite synchronization trees to a finite synchronization tree.

**Definition 7.10 (The operation $\mathbf{f}_{\mathsf{ST}}$)** *Assume a compact GSOS language, and consider a function symbol $f$. The operation $\mathbf{f}_{\mathsf{ST}} : \mathsf{ST}(\mathsf{Act})^{ar(f)} \to \mathsf{ST}(\mathsf{Act})$ is defined inductively by stipulating that, for every $S_1, \ldots, S_{ar(f)} \in \mathsf{ST}(\mathsf{Act})$:*

- *$\perp \in \mathbf{f}_{\mathsf{ST}}(S_1, \ldots, S_{ar(f)})$ iff $f = \Omega$ or there is an argument $i$ of $f$ such that $f$ tests its $i$th argument (see Def. 5.11) and $\perp \in S_i$;*
- *$\langle c, S\rangle \in \mathbf{f}_{\mathsf{ST}}(S_1, \ldots, S_{ar(f)})$ iff there are a GSOS rule*

$$\frac{\{x_i \stackrel{a_{ij}}{\to} y_{ij},\ x_i \stackrel{b_{ik}}{\not\to} \mid 1 \leq i \leq ar(f), 1 \leq j \leq m_i, 1 \leq k \leq n_i\}}{f(x_1, \ldots, x_{ar(f)}) \stackrel{c}{\to} C[x_1, \ldots, x_{ar(f)}, y_{11}, \ldots, y_{ar(f)m_{ar(f)}}]}$$

  *and finite synchronization trees $S'_{i1}, \ldots, S'_{im_i}$ for $i = 1, \ldots, ar(f)$ such that:*

  1. *$\langle a_{ij}, S'_{ij}\rangle \in S_i$ for $i = 1, \ldots, ar(f)$ and $j = 1, \ldots, m_i$;*
  2. *if $n_i > 0$, then $\perp \notin S_i$ and $\langle b_{ik}, S'\rangle \notin S_i$ for $S' \in \mathsf{ST}(\mathsf{Act})$ and $k = 1, \ldots, n_i$;*
  3. *$\mathbf{C}_{\mathsf{ST}}[S_1, \ldots, S_{ar(f)}, S'_{11}, \ldots, S'_{ar(f)m_{ar(f)}}] = S$, where $\mathbf{C}_{\mathsf{ST}}$ denotes the derived semantic operation associated with the target of the GSOS rule.*

The above definition, which is discussed at length in [11, 12], endows the preorder of finite synchronization trees $\mathsf{ST}(\mathsf{Act})$ with a $\Sigma$-preorder structure (where $\Sigma$ is the signature under consideration), in the sense of [118]. Since the poset of compact elements of the domain $\mathcal{D}$ is a substructure of the preorder $(\mathsf{ST}(\mathsf{Act}), \sqsubseteq)$, this is enough to give a denotational interpretation for the recursion-free terms in a compact GSOS language in terms of compact elements in $\mathcal{D}$.

**Theorem 7.11** *Assume a compact GSOS language. For all* $t, u \in \mathrm{T}(\Sigma)$, $t \lesssim_\omega u$ *iff* $\mathcal{K}(\mathcal{D})[\![t]\!] \sqsubseteq \mathcal{K}(\mathcal{D})[\![u]\!]$.

The above full abstraction result can be extended to the whole of the language $\mathsf{CREC}(\Sigma)$, for any compact recursive GSOS language. In order to define an interpretation of programs in $\mathsf{CREC}(\Sigma)$ as elements of $\mathcal{D}$, we need to define a continuous $\Sigma$-algebra structure on $\mathcal{D}$.

From the theory of powerdomains [130, 174, 205], we know that the domain of synchronization trees $\mathcal{D}$ is, up to isomorphism, the ideal completion of the poset of compact elements $\mathcal{K}(\mathcal{D})$. (The construction of the ideal completion of a poset and a discussion of its basic properties can be found in [118, p. 139–145].) Let $\sqsubseteq_\mathcal{D}$ be standard set inclusion on $\mathcal{D}$. Since $\mathcal{D}$ is the ideal completion of $\mathcal{K}(\mathcal{D})$, the monotonic function $\mathbf{f}_{\mathsf{ST}} : (\mathcal{K}(\mathcal{D}), \sqsubseteq)^{ar(f)} \to (\mathcal{K}(\mathcal{D}), \sqsubseteq)$ for any function symbol $f$ can be extended to a continuous function $\mathbf{f}_\mathcal{D} : (\mathcal{D}, \sqsubseteq_\mathcal{D})^{ar(f)} \to (\mathcal{D}, \sqsubseteq_\mathcal{D})$ by:

$$\mathbf{f}_\mathcal{D}(e_1, \ldots, e_{ar(f)}) \quad \triangleq \quad \bigcup \left\{ \mathbf{f}_{\mathsf{ST}}(d_1, \ldots, d_{ar(f)}) \mid d_1 \in e_1, \ldots, d_{ar(f)} \in e_{ar(f)} \right\} \ ,$$

where $e_1, \ldots, e_{ar(f)}$ are ideals in $(\mathcal{K}(\mathcal{D}), \sqsubseteq_{\mathcal{K}(\mathcal{D})})$, and we identify an element of $(\mathcal{K}(\mathcal{D}), \sqsubseteq_{\mathcal{K}(\mathcal{D})})$ with the principal ideal it generates (see [118, p. 139]). The interested reader is invited to consult, e.g., [118, Sect. 3.3] for a discussion of the properties afforded by this canonical extension. By the general theory of algebraic semantics we then have that, for all closed terms $t, u$,

$$\mathcal{D}[\![t]\!] \sqsubseteq_\mathcal{D} \mathcal{D}[\![u]\!] \ \Leftrightarrow \ \mathcal{K}(\mathcal{D})[\![t]\!] \sqsubseteq \mathcal{K}(\mathcal{D})[\![u]\!] \ .$$

In view of Thm. 7.11, the desired full abstraction result follows if we prove that the preorder $\lesssim_\omega$ is algebraic. Namely, owing to our constructions each closed term $t$ is interpreted as a compact element of $\mathcal{D}$, so Lem. 7.1 implies that the relation $\sqsubseteq_\mathcal{D}$ is algebraic. And two algebraic relations that coincide over the collection of closed terms $\mathrm{T}(\Sigma)$ do, in fact, coincide over the whole of $\mathsf{CREC}(\Sigma)$.

The key to the proof of algebraicity of $\lesssim_\omega$ is the following general theorem from [11, 12], providing a partial completeness result for $\lesssim$ in the sense of Hennessy [10, 116] for arbitrary compact GSOS languages. This axiomatizability result uses the fact that $\lesssim$ is a precongruence with respect to the LTS with divergence associated with a recursive GSOS language including the inert constant $\Omega$; see Prop. 7.3.

**Proposition 7.12** *Let $G$ be a compact recursive GSOS language. Then there exists a compact recursive GSOS language $H$ over a signature $\Sigma'$, being a disjoint extension (cf. Def. 5.32) of $G$ and $T_{\mathsf{FINTREE}}$ (cf. Sect. 5.4.5), together with a set $\mathcal{I}$ of inequalities between recursive terms over $\Sigma'$, such that for all $t \in \mathrm{T}(\Sigma')$ and $u \in \mathsf{CREC}(\Sigma')$:*

$$H \textit{ induces } t \lesssim u \textit{ iff } \mathcal{I} \vdash t \leq u \ .$$

Apart from its intrinsic interest, the main consequence of Prop. 7.12 is the following key result from [11, 12], essentially stating that, for any compact GSOS language, finite synchronization trees are compact elements with respect to the preorder $\lesssim$.

**Proposition 7.13** *Assume a compact recursive GSOS language. If $S \in \mathsf{ST}(\mathsf{Act})$ and $t \in \mathsf{CREC}(\Sigma)$, then $S \lesssim^F t$ iff there exists a finite approximation $t_n$ of $t$ such that $S \lesssim^F t_n$.*

The above result, in conjunction with Prop. 7.6, yields that $\lesssim_\omega$ is indeed algebraic.

**Proposition 7.14** *$\lesssim_\omega$ is algebraic over the LTS with divergence associated with a compact recursive GSOS language.*

In light of Thm. 7.11 and Prop. 7.14, for any compact recursive GSOS language the induced denotational semantics over $\mathsf{CREC}(\Sigma)$ is fully abstract with respect to $\lesssim_\omega$.

**Theorem 7.15** *Assume a compact recursive GSOS language. For all $t, u \in \mathsf{CREC}(\Sigma)$, $t \lesssim_\omega u$ iff $\mathcal{D}[\![t]\!] \sqsubseteq_{\mathcal{D}} \mathcal{D}[\![u]\!]$.*

When applied to the version of SCCS considered by Abramsky [1], the techniques we have presented deliver a denotational semantics that is exactly the one given in *op. cit.*

**Related Work** The work reported in this section is by no means the only attempt to systematically derive denotational models from SOS language specifications. The main precursors to this work in the field of the meta-theory of process description languages may be found in the work by Bloom [47] and Rutten [189, 190, 191, 192]. In his unpublished paper [47], Bloom gives operational, logical, relational, and three denotational semantics for GSOS languages without negative premises and unguarded recursion, and shows that they coincide. Bloom's work is based on the behavioural notion of simulation [124], and two of his denotational semantics are given by means of Scott domains based on finite synchronization trees. On the other hand, the work by Rutten [189, 190, 191] gives methods for deriving a denotational semantics based on complete metric spaces and Aczel's non-well-founded sets [14] for languages specified by means of subformats of the tyft/tyxt format (cf. Def. 5.2). In particular, the reference [191] gives a detailed and clear introduction to a technique, called "processes as terms", for the definition of operations on semantic models from transition rules. Rutten's general "processes as terms" approach could have been applied to yield an equivalent formulation of the semantic operations on finite synchronization trees given above. The work presented in the aforementioned papers has been generalized by Rutten and Turi [192]. *Ibidem* it is shown how TSSs in tyft/tyxt format induce a denotational semantics, and the essential properties of semantic domains that make their definitions possible are investigated in a categorical perspective.

Abramsky and Vickers [3] consider various notions of process observations in a uniform algebraic framework provided by the theory of *quantales* (see, e.g., [188]). The methods developed in [3] yield, in a uniform fashion, observational logics and denotational models for each notion of process observation they consider. Their work is, however, semantic in nature, and ignores the algebraic structure of process expressions.

In the area of the semantics of functional programs, developments that are somewhat similar in spirit to those given above are presented in [145, 204]. Those papers study natural notions of preorder over programs written in a simple functional programming language, and show how any ordering on programs with certain basic properties can be extended to a term model that is fully abstract with respect to it.

The issue of defining abstract mathematical models *for*, rather than from, operational semantics has also received some attention. We refer the interested reader to, e.g., [19, 214], and the references therein, for details on this line of investigation.

## References


[1] S. Abramsky, *A domain equation for bisimulation*, Information and Computation, 92 (1991), pp. 161–218.

[2] S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[3] S. Abramsky and S. Vickers, *Quantales, observational logic and process semantics*, Mathematical Structures in Computer Science, 3 (1993), pp. 161–227.

[4] L. Aceto, *Deriving complete inference systems for a class of GSOS languages generating regular behaviours*, in Jonsson and Parrow [133], pp. 449–464.

[5] ——, *GSOS and finite labelled transition systems*, Theoretical Comput. Sci., 131 (1994), pp. 181–195.

[6] L. Aceto, B. Bloom, and F. Vaandrager, *Checking equations in GSOS systems*, 1992. Unpublished working paper.

[7] ——, *Turning SOS rules into equations*, in LICS92 [141], pp. 113–124. Preliminary version of [8].

[8] ——, *Turning SOS rules into equations*, Information and Computation, 111 (1994), pp. 1–52.

[9] L. Aceto, W. Fokkink, R. v. Glabbeek, and A. Ingólfsdóttir, *Axiomatizing prefix iteration with silent steps*, Information and Computation, 127 (1996), pp. 26–40.

[10] L. Aceto and M. Hennessy, *Termination, deadlock and divergence*, J. Assoc. Comput. Mach., 39 (1992), pp. 147–187.

[11] L. Aceto and A. Ingólfsdóttir, *CPO models for a class of GSOS languages*, in Mosses et al. [166], pp. 439–453. Preliminary version of [12].

[12] ——, *CPO models for compact GSOS languages*, Information and Computation, 129 (1996), pp. 107–141.

[13] ——, *A characterization of finitary bisimulation*, Inf. Process. Lett., 64 (1997), pp. 127–134.

[14] P. Aczel, *Non-well-founded Sets*, vol. 14 of CSLI Lecture Notes, Stanford University, 1988.

[15] S. Allen, R. Constable, D. Howe, and W. Aitken, *The semantics of reflected proof*, in Proceedings 5th Symposium on Logic in Computer Science, Philadelphia, PA, IEEE Computer Society Press, 1990, pp. 95–105.

[16] D. Austry and G. Boudol, *Algèbre de processus et synchronisations*, Theoretical Comput. Sci., 30 (1984), pp. 91–131.

[17] F. Baader and T. Nipkow, *Term Rewriting and All That*, Cambridge University Press, 1998.

[18] J. Backus, *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference*, in Proceedings ICIP, Unesco, 1960, pp. 125–131.

[19] E. Badouel, *Conditional rewrite rules as an algebraic semantics of processes*, Research Report 1226, INRIA Rennes, 1990.

[20] E. Badouel and P. Darondeau, *Structural operational specifications and trace automata*, in Cleaveland [64], pp. 302–316.

[21] J. Baeten, ed., *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[22] J. Baeten and J. Bergstra, *A survey of axiom systems for process algebras*, Report P9111, University of Amsterdam, Amsterdam, 1991.

[23] ——, *Discrete time process algebra*, in Cleaveland [64], pp. 401–420.

[24] J. Baeten, J. Bergstra, and J. W. Klop, *Syntax and defining equations for an interrupt mechanism in process algebra*, Fundamenta Informaticae, IX (1986), pp. 127–168.

[25] ——, *On the consistency of Koomen's fair abstraction rule*, Theoretical Comput. Sci., 51 (1987), pp. 129–176.

[26] J. Baeten and J. W. Klop, eds., *Proceedings 1st Conference on Concurrency Theory,* Amsterdam, The Netherlands, vol. 458 of Lecture Notes in Computer Science, Springer-Verlag, 1990.

[27] J. Baeten and C. Verhoef, *A congruence theorem for structured operational semantics with predicates*, in Best [44], pp. 477–492.

[28] ——, *Concrete process algebra*, in Handbook of Logic in Computer Science, S. Abramsky, D. Gabbay, and T. Maibaum, eds., vol. IV, Oxford University Press, 1995, pp. 149–268.

[29] J. Baeten and P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[30] J. d. Bakker, *Semantics of programming languages*, Advances in Information Systems Science, 2 (1969), pp. 173–227.

[31] J. d. Bakker and J. Zucker, *Processes and the denotational semantics of concurrency*, Information and Control, 54 (1982), pp. 70–120.

[32] H. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, vol. 103 of Studies in Logic and the Foundation of Mathematics, North-Holland, Amsterdam, 1984.

[33] T. Basten, *Branching bisimilarity is an equivalence indeed!*, Information Processing Letters, 58 (1996), pp. 141–147.

[34] J. Bergstra, I. Bethke, and A. Ponse, *Process algebra with iteration and nesting*, Computer Journal, 37 (1994), pp. 243–258.

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

[36] ——, *The algebra of recursively defined processes and the algebra of regular processes*, in Proceedings 11th Colloquium on Automata, Languages and Programming, Antwerp, Belgium, J. Paredaens, ed., vol. 172 of Lecture Notes in Computer Science, Springer-Verlag, 1984, pp. 82–95.

[37] ——, *Process algebra for synchronous communication*, Information and Control, 60 (1984), pp. 109–137.

[38] ——, *Conditional rewrite rules: Confluence and termination*, J. Comput. System Sci., 32 (1986), pp. 323–362.

[39] ——, *Verification of an alternating bit protocol by means of process algebra*, in Spring School on Mathematical Methods of Specification and Synthesis of Software Systems, Wendisch-Rietz, Germany, W. Bibel and K. Jantke, eds., no. 215 in Lecture Notes in Computer Science, Springer-Verlag, 1986, pp. 9–23.

[40] ——, *A complete inference system for regular processes with silent moves*, in Proceedings Logic Colloquium 1986, F. Drake and J. Truss, eds., Hull, 1988, North-Holland, pp. 21–81.

[41] J. Bergstra, A. Ponse, and J. v. Wamel, *Process algebra with backtracking*, in de Bakker et al. [73], pp. 46–91.

[42] K. Bernstein, *A congruence theorem for structured operational semantics of higher-order languages*, in Proceedings 13th Symposium on Logic in Computer Science, Indianapolis, Indiana, IEEE Computer Society Press, 1998, pp. 153–164.

[43] G. Berry, *A hardware implementation of Pure Esterel*, Rapport de Recherche 06/91, Ecole des Mines, CMA, Sophia-Antipolis, France, 1991.

[44] E. Best, ed., *Proceedings 4th Conference on Concurrency Theory,* Hildesheim, Germany, vol. 715 of Lecture Notes in Computer Science, Springer-Verlag, 1993.

[45] 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, 1989.

[46] ——, *Can LCF be topped? Flat lattice models of typed $\lambda$-calculus*, Information and Computation, 87 (1990), pp. 263–300.

[47] ——, *Many meanings of monosimulation: denotational, operational, and logical characterizations of a notion of simulation of concurrent processes*, 1991. Unpublished manuscript.

[48] ——, *Ready, set, go: structural operational semantics for linear-time process algebras*, Report TR 93-1372, Cornell University, Ithaca, New York, 1993.

[49] ——, *CHOCOLATE: Calculi of Higher Order COmmunication and LAmbda TErms*, in Conference Record 21st ACM Symposium on Principles of Programming Languages, Portland, Oregon, 1994, pp. 339–347.

[50] ——, *When is partial trace equivalence adequate?*, Formal Aspects of Computing, 6 (1994), pp. 317–338.

[51] ——, *Structural operational semantics for weak bisimulations*, Theoretical Comput. Sci., 146 (1995), pp. 25–68.

[52] ——, *Structured operational semantics as a specification language*, in Conference Record 22nd ACM Symposium on Principles of Programming Languages, San Francisco, California, 1995, pp. 107–117.

[53] B. Bloom, A. Cheng, and A. Dsouza, *Using a protean language to enhance expressiveness in specification*, IEEE Transactions on Software Engineering, 23 (1997), pp. 224–234.

[54] B. Bloom, S. Istrail, and A. Meyer, *Bisimulation can't be traced: preliminary report*, in Conference Record 15th ACM Symposium on Principles of Programming Languages, San Diego, California, 1988, pp. 229–239. Preliminary version of [55].

[55] ——, *Bisimulation can't be traced*, J. Assoc. Comput. Mach., 42 (1995), pp. 232–268.

[56] R. Bol and J. F. Groote, *The meaning of negative premises in transition system specifications (extended abstract)*, in Proceedings 18th Colloquium on Automata, Languages and Programming, Madrid, Spain, J. Leach Albert, B. Monien, and M. Rodríguez, eds., vol. 510 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 481–494. Preliminary version of [57].

[57] R. Bol and J. F. Groote, *The meaning of negative premises in transition system specifications*, J. Assoc. Comput. Mach., 43 (1996), pp. 863–914.

[58] T. Bolognesi and F. Lucidi, *Timed process algebras with urgent interactions and a unique powerful binary operator*, in Proceedings REX Workshop on Real-Time: Theory in Practice, Mook, The Netherlands, June 1991, J. de Bakker, C. Huizing, W. d. Roever, and G. Rozenberg, eds., vol. 600 of Lecture Notes in Computer Science, Springer-Verlag, 1992, pp. 124–148.

[59] D. Bosscher, *Term rewriting properties of SOS axiomatisations*, in Hagiya and Mitchell [114], pp. 425–439.

[60] S. Brookes, C. Hoare, and A. Roscoe, *A theory of communicating sequential processes*, J. Assoc. Comput. Mach., 31 (1984), pp. 560–599.

[61] S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, eds., *Proceedings 7th Conference on Mathematical Foundations of Programming Semantics,* Pittsburgh, PA, vol. 598 of Lecture Notes in Computer Science, Springer-Verlag, 1992.

[62] R. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Trans. Comput., C-35 (1986), pp. 677–691.

[63] K. Clark, *Negation as failure*, in Logic and Databases, H. Gallaire and J. Minker, eds., Plenum Press, New York, 1978, pp. 293–322.

[64] R. Cleaveland, ed., *Proceedings 3rd Conference on Concurrency Theory,* Stony Brook, NY, vol. 630 of Lecture Notes in Computer Science, Springer-Verlag, 1992.

[65] R. Cleaveland and M. Hennessy, *Priorities in process algebras*, Information and Computation, 87 (1990), pp. 58–77.

[66] R. Cleaveland, J. Parrow, and B. Steffen, *The concurrency workbench: A semantics-based verification tool for finite state systems*, ACM Trans. Prog. Lang. Syst., 15 (1993), pp. 36–72.

[67] R. Constable, S. Allen, H. Bromley, R. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.

[68] B. Courcelle and M. Nivat, *Algebraic families of interpretations*, in Proceedings 17th Symposium on Foundations of Computer Science, Houston, Texas, IEEE, 1976, pp. 137–146.

[69] P. D'Argenio, *A general conservative extension theorem in process algebras with inequalities*, in Proceedings 2nd Workshop on the Algebra of Communicating Processes, Eindhoven, The Netherlands, A. Ponse, C. Verhoef, and B. v. Vlijmen, eds., Report CS-95-14, Eindhoven University of Technology, 1995, pp. 67–79.

[70] P. D'Argenio and C. Verhoef, *A general conservative extension theorem in process algebras with inequalities*, Theoretical Comput. Sci., 177 (1997), pp. 351–380.

[71] P. Darondeau, *Concurrency and computability*, in Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on

Theoretical Computer Science, La Roche Posay, France, I. Guessarian, ed., vol. 469 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 223–238.

[72] J. de Bakker, W. d. Roever, and G. Rozenberg, eds., *Proceedings REX Workshop on Semantics: Foundations and Applications,* Beekbergen, The Netherlands, June 1992, vol. 666 of Lecture Notes in Computer Science, Springer-Verlag, 1993.

[73] ——, eds., *Proceedings REX School/Symposium on A Decade of Concurrency: Reflections and Perspectives,* Noordwijkerhout, The Netherlands, vol. 803 of Lecture Notes in Computer Science, Springer-Verlag, 1994.

[74] N. De Francesco and P. Inverardi, *Proving finiteness of CCS processes by non-standard semantics*, Acta Informatica, 31 (1994), pp. 55–80.

[75] R. De Nicola and M. Hennessy, *Testing equivalences for processes*, Theoretical Comput. Sci., 34 (1984), pp. 83–133.

[76] A. Dsouza and B. Bloom, *Generating BDD models for process algebra terms*, in Proceedings 7th Conference on Computer Aided Verification, Liege, Belgium, P. Wolper, ed., vol. 939 of Lecture Notes in Computer Science, Springer Verlag, 1995, pp. 16–30.

[77] ——, *On the expressive power of CCS*, in Proceedings 15th Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India, P. Thiagarajan, ed., vol. 1026 of Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 309–323.

[78] A. Emerson, *Automated temporal reasoning about reactive systems*, in Moller and Birtwistle [164], pp. 41–101.

[79] R. Enders, T. Filkorn, and D. Taubner, *Generating BDDs for symbolic model checking in CCS*, Distributed Computing, 6 (1993), pp. 155–164.

[80] F. Fages, *A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics*, New Generation Computing, 9 (1991), pp. 425–443.

[81] J.-C. Fernandez, *Aldébaran: a tool for verification of communicating processes*, technical report SPECTRE c14, LGI–IMAG, Grenoble, 1989.

[82] W. Fokkink, *The tyft/tyxt format reduces to tree rules*, in Hagiya and Mitchell [114], pp. 440–453. Preliminary version of [85].

[83] ——, *Language preorder as a precongruence*, Report CSR 11-98, University of Wales Swansea, 1998. To appear in *Theoretical Computer Science*.

[84] ——, *Rooted branching bisimulation as a congruence*, Report CSR 16-98, University of Wales Swansea, 1998. To appear in *Journal of Computer and System Sciences*.

[85] W. Fokkink and R. v. Glabbeek, *Ntyft/ntyxt rules reduce to ntree rules*, Information and Computation, 126 (1996), pp. 1–10.

[86] W. Fokkink and S. Klusener, *An effective axiomatization for real time ACP*, Information and Computation, 122 (1995), pp. 286–299.

[87] W. Fokkink and C. Verhoef, *A conservative look at operational semantics with variable binding*, Information and Computation, 146 (1998), pp. 24–54.

[88] ——, *Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories*, in Proceedings 2nd Conference on Fundamental Approaches to Software Engineering, Amsterdam, The Netherlands, J.-P. Finance, ed., vol. 1577 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 98–113.

[89] W. Fokkink and H. Zantema, *Basic process algebra with iteration: Completeness of its equational axioms*, Computer Journal, 37 (1994), pp. 259–267.

[90] A. v. Gelder, K. Ross, and J. Schlipf, *Unfounded sets and well-founded semantics for general logic programs*, in Proceedings 7th ACM Symposium on Principles of Database Systems, Austin, Texas, ACM, 1988, pp. 221–230. Preliminary version of [91].

[91] ——, *The well-founded semantics for general logic programs*, J. Assoc. Comput. Mach., 38 (1991), pp. 620–650.

[92] M. Gelfond and V. Lifschitz, *The stable model semantics for logic programming*, in Proceedings 5th Conference on Logic Programming, Seattle, Washington, R. Kowalski and K. Bowen, eds., MIT Press, 1988, pp. 1070–1080.

[93] G. Gentzen, *Investigations into logical deduction*, in The Collected Papers of Gerhard Gentzen, M. Szabo, ed., North-Holland, 1969, pp. 68–128.

[94] R. v. Glabbeek, *Bounded nondeterminism and the approximation induction principle in process algebra*, in Proceedings 4th Symposium on Theoretical Aspects of Computer Science, Passau, Germany, F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds., vol. 247 of Lecture Notes in Computer Science, Springer-Verlag, 1987, pp. 336–347.

[95] ——, *Comparative Concurrency Semantics and Refinement of Actions*, PhD thesis, Free University, Amsterdam, 1990.

[96] ——, *The linear time – branching time spectrum*, in Baeten and Klop [26], pp. 278–297.

[97] ——, *A complete axiomatization for branching bisimulation congruence of finite-state behaviours*, in Proceedings 18th Symposium on Mathematical Foundations of Computer Science 1993, Gdansk, Poland, A. Borzyszkowski and S. Sokołowski, eds., vol. 711 of Lecture Notes in Computer Science, Springer-Verlag, 1993, pp. 473–484.

[98] ——, *Full abstraction in structural operational semantics (extended abstract)*, in Proceedings 3rd Conference on Algebraic Methodology and Software Technology, Enschede, The Netherlands, M. Nivat, C. Rattray, T. Rus, and G. Scollo, eds., Workshops in Computing, Springer-Verlag, 1993, pp. 77–84.

[99] ——, *The linear time – branching time spectrum II: the semantics of sequential processes with silent moves*, in Best [44], pp. 66–81.

[100] ——, *The meaning of negative premises in transition system specifications II*, Report STAN-CS-TN-95-16, Department of Computer Science, Stanford University, 1995.

[101] ——, *On the expressiveness of ACP*, in Ponse et al. [181], pp. 188–217.

[102] ——, *The meaning of negative premises in transition system specifications II (extended abstract)*, in Automata, Languages and Programming, 23rd Colloquium, F. Meyer auf der Heide and B. Monien, eds., vol. 1099 of Lecture Notes in Computer Science, Paderborn, Germany, 1996, Springer-Verlag, pp. 502–513.

[103] ——, *Personal communication*, March 1999.

[104] R. v. Glabbeek and P. Weijland, *Branching time and abstraction in bisimulation semantics (extended abstract)*, in Information Processing 89, G. Ritter, ed., North-Holland, 1989, pp. 613–618. Preliminary version of [105].

[105] ——, *Branching time and abstraction in bisimulation semantics*, J. Assoc. Comput. Mach., 43 (1996), pp. 555–600.

[106] J. Goguen, J. Thatcher, E. Wagner, and J. Wright, *Initial algebra semantics and continuous algebras*, J. Assoc. Comput. Mach., 24 (1977), pp. 68–95.

[107] J. F. Groote, *Transition system specifications with negative premises (extended abstract)*, in Baeten and Klop [26], pp. 332–341. Preliminary version of [108].

[108] ——, *Transition system specifications with negative premises*, Theoretical Comput. Sci., 118 (1993), pp. 263–299.

[109] J. F. Groote and A. Ponse, *The syntax and semantics of $\mu$CRL*, in Ponse et al. [181], pp. 26–62.

[110] J. F. Groote and F. Vaandrager, *Structured operational semantics and bisimulation as a congruence (extended abstract)*, in Proceedings 16th Colloquium on Automata, Languages and Programming, Stresa, Italy, G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, eds., vol. 372 of Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 423–438. Preliminary version of [111].

[111] ——, *Structured operational semantics and bisimulation as a congruence*, Information and Computation, 100 (1992), pp. 202–260.

[112] I. Guessarian, *Algebraic Semantics*, vol. 99 of Lecture Notes in Computer Science, Springer-Verlag, 1981.

[113] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, Foundations of Computing, MIT Press, 1992.

[114] M. Hagiya and J. Mitchell, eds., *Proceedings 2nd Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, vol. 789 of Lecture Notes in Computer Science, Springer-Verlag, 1994.

[115] P. Hartel, *LATOS – a Lightweight Animation Tool for Operational Semantics*, Report DSSE-TR-97-1, University of Southampton, 1997.

[116] M. Hennessy, *A term model for synchronous processes*, Information and Control, 51 (1981), pp. 58–75.

[117] ——, *Acceptance trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 896–928.

[118] ——, *Algebraic Theory of Processes*, MIT Press, Cambridge, Massachusetts, 1988.

[119] ——, *The Semantics of Programming Languages — An Elementary Introduction Using Structural Operational Semantics*, John Wiley & Sons, Chichester, England, 1990.

[120] ——, *A fully abstract denotational model for higher-order processes*, Information and Computation, 112 (1994), pp. 55–95.

[121] M. Hennessy and A. Ingólfsdóttir, *Communicating processes with value-passing and assignment*, Formal Aspects of Computing, 5 (1993), pp. 432–466.

[122] ——, *A theory of communicating processes with value-passing*, Information and Computation, 107 (1993), pp. 202–236.

[123] M. Hennessy and R. Milner, *On observing nondeterminism and concurrency*, in Proceedings 7th Colloquium on Automata, Languages and Programming, Noorwijkerhout, The Netherlands, J. d. Bakker and J. van Leeuwen, eds., vol. 85 of Lecture Notes in Computer Science, Springer-Verlag, 1980, pp. 299–309. Preliminary version of [124].

[124] ——, *Algebraic laws for nondeterminism and concurrency*, J. Assoc. Comput. Mach., 32 (1985), pp. 137–161.

[125] M. Hennessy and G. Plotkin, *Full abstraction for a simple programming language*, in Proceedings 8th Symposium on Mathematical Foundations of Computer Science, Olomouc, Czechoslovakia, J. Bečvář, ed., vol. 74 of Lecture Notes in Computer Science, Springer-Verlag, 1979, pp. 108–120.

[126] ——, *A term model for CCS*, in Proceedings 9th Symposium on Mathematical Foundations of Computer Science, Rydzyna, Poland, P. Dembiński, ed., vol. 88 of Lecture Notes in Computer Science, Springer-Verlag, 1980, pp. 261–274.

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

[128] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, 1985.

[129] D. Howe, *Proving congruence of bisimulation in functional programming languages*, Information and Computation, 124 (1996), pp. 103–112.

[130] A. Ingólfsdóttir, *Semantic Models for Communicating Process with Value-Passing*, PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, 1994.

[131] A. Jeffrey, *CSP is completely expressive*, Computer Science Technical Report 92:02, School of Cognitive and Computing Sciences, University of Sussex, 1992.

[132] H. Jifeng and C. Hoare, *From algebra to operational semantics*, Inf. Process. Lett., 45 (1993), pp. 75–80.

[133] B. Jonsson and J. Parrow, eds., *Proceedings 5th Conference on Concurrency Theory,* Uppsala, Sweden, vol. 836 of Lecture Notes in Computer Science, Springer-Verlag, 1994.

[134] R. Keller, *Formal verification of parallel programs*, Comm. ACM, 19 (1976), pp. 371–384.

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

[136] S. Klusener, *Completeness in real time process algebra*, in Proceedings 2nd Conference on Concurrency Theory, Amsterdam, The Netherlands, J. Baeten and J. F. Groote, eds., vol. 527 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 376–392.

[137] K. G. Larsen, *Modal specifications*, Tech. Rep. R 89-09, Institute for Electronic Systems, University of Aalborg, 1989.

[138] K. G. Larsen and A. Skou, *Bisimulation through probabilistic testing*, Information and Computation, 94 (1991), pp. 1–28.

[139] ——, *Compositional verification of probabilistic processes*, in Cleaveland [64], pp. 456–471.

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

[141] *Logic in Computer Science, Proceedings 7th Symposium,* Santa Cruz, California, IEEE Computer Society Press, 1992.

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

[143] ——, *On program correctness and the stepwise development of implementations*, in Proceedings of the Convegno di Informatica Teorica, University of Pisa, March 1973, 1973, pp. 219–251.

[144] E. Madelaine and D. Vergamini, *Finiteness conditions and structural construction of automata for all process algebras*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 3 (1991), pp. 275–292.

[145] I. Mason, S. Smith, and C. Talcott, *From operational semantics to domain theory*, Information and Computation, 128 (1996), pp. 26–47.

[146] S. Mauw and G. Veltink, *An introduction to $PSF_d$*, in Proceedings 3rd Conference on Theory and Practice of Software Development, Vol. 2, Barcelona, Spain, J. Díaz and F. Orejas, eds., vol. 352 of Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 272–285.

[147] J. McCarthy, *Towards a mathematical science of computation*, in Information Processing 1962, C. Popplewell, ed., 1963, pp. 21–28.

[148] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[149] A. Meyer, *Semantical paradigms: notes for an invited lecture*, in Proceedings 3rd Symposium on Logic in Computer Science, Edinburgh, IEEE Computer Society Press, 1988, pp. 236–242.

[150] G. Milne and R. Milner, *Concurrent processes and their syntax*, J. Assoc. Comput. Mach., 26 (1979), pp. 302–321.

[151] R. Milner, *Processes: A mathematical model of computing agents*, in Proceedings Logic Colloquium 1973, Bristol, UK, H. Rose and J. Shepherdson, eds., North-Holland, 1973, pp. 158–173.

[152] ——, *Fully abstract models of typed $\lambda$-calculi*, Theoretical Comput. Sci., 4 (1977), pp. 1–22.

[153] ——, *A Calculus of Communicating Systems*, vol. 92 of Lecture Notes in Computer Science, Springer-Verlag, 1980.

[154] ——, *A modal characterisation of observable machine behaviour*, in 6th Colloquium on Trees in Algebra and Programming, Genoa, Italy, E. Astesiano and C. Böhm, eds., vol. 112 of Lecture Notes in Computer Science, Springer-Verlag, 1981, pp. 25–34.

[155] ——, *On relating synchrony and asynchrony*, Tech. Rep. CSR–75–80, Department of Computer Science, University of Edinburgh, 1981.

[156] ——, *Calculi for synchrony and asynchrony*, Theoretical Comput. Sci., 25 (1983), pp. 267–310.

[157] ——, *A complete inference system for a class of regular behaviours*, J. Comput. System Sci., 28 (1984), pp. 439–466.

[158] ——, *Communication and Concurrency*, Prentice-Hall International, Englewood Cliffs, 1989.

[159] ——, *A complete axiomatisation for observational congruence of finite-state behaviors*, Information and Computation, 81 (1989), pp. 227–247.

[160] ——, *Elements of interaction (Turing Award Lecture)*, Comm. ACM, 36 (1993), pp. 78–89.

[161] R. Milner, J. Parrow, and D. Walker, *A calculus of mobile processes, part I + II*, Information and Computation, 100 (1992), pp. 1–77.

[162] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[163] F. Moller, *The importance of the left merge operator in process algebras*, in Proceedings 17th Colloquium on Automata, Languages and Programming, Warwick, UK, M. Paterson, ed., vol. 443 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 752–764.

[164] F. Moller and G. Birtwistle, eds., *Logics for Concurrency: Structure versus Automata*, vol. 1043 of Lecture Notes in Computer Science, Springer-Verlag, 1996.

[165] F. Moller and C. Tofts, *A temporal calculus of communicating systems*, in Baeten and Klop [26], pp. 401–415.

[166] P. Mosses, M. Nielsen, and M. Schwartzbach, eds., *Proceedings 6th Conference on Theory and Practice of Software Development (TAPSOFT'95)*, Århus, Denmark, vol. 915 of Lecture Notes in Computer Science, Springer-Verlag, 1995.

[167] X. Nicollin and J. Sifakis, *The algebra of timed processes,* **ATP***: theory and application*, Information and Computation, 114 (1994), pp. 131–178.

[168] H. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*, Wiley Professional Computing, John Wiley & Sons, Chichester, England, 1992.

[169] V. v. Oostrom and E. de Vink, *Transition system specifications in stalk format with bisimulation as a congruence*, in Proceedings 11th Symposium on Theoretical Aspects of Computer Science, Caen, France, P. Enjalbert, E. Mayr, and K. Wagner, eds., vol. 775 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 569–580.

[170] S. Owicki and D. Gries, *An axiomatic proof technique for parallel programs*, Acta Inf., 6 (1976), pp. 319–340.

[171] D. Park, *Concurrency and automata on infinite sequences*, in 5th GI Conference, Karlsruhe, Germany, P. Deussen, ed., vol. 104 of Lecture Notes in Computer Science, Springer-Verlag, 1981, pp. 167–183.

[172] J. Parrow, *The expressive power of parallelism*, Future Generation Computer Systems, 6 (1990), pp. 271–285.

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

[174] G. Plotkin, *A powerdomain construction*, SIAM J. Comput., 5 (1976), pp. 452–487.

[175] ——, *LCF considered as a programming language*, Theoretical Comput. Sci., 5 (1977), pp. 223–256.

[176] ——, *Lecture notes in domain theory*, 1981. University of Edinburgh.

[177] ——, *A structural approach to operational semantics*, Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[178] ——, *An operational semantics for CSP*, in Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II, Garmisch-Partenkirchen, Germany, D. Bjørner, ed., North-Holland, 1983, pp. 199–225.

[179] A. Pnueli, *The temporal logic of programs*, in Proceedings 18th Symposium on Foundations of Computer Science, Providence, Rhode Island, IEEE, 1977, pp. 46–57.

[180] A. Ponse, *Computable processes and bisimulation equivalence*, Formal Aspects of Computing, 8 (1996), pp. 648–678.

[181] A. Ponse, C. Verhoef, and B. v. Vlijmen, eds., *Proceedings 1st Workshop on the Algebra of Communicating Processes,* Utrecht, The Netherlands, Workshops in Computing, Springer-Verlag, 1995.

[182] T. Przymusinski, *On the declarative semantics of deductive databases and logic programs*, in Foundations of Deductive Databases and Logic Programming, J. Minker, ed., Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988, pp. 193–216.

[183] ——, *The well-founded semantics coincides with the three-valued stable semantics*, Fundamenta Informaticae, 13 (1990), pp. 445–463.

[184] W. Reisig, *Petri nets – an introduction*, EATCS Monographs on Theoretical Computer Science, Volume 4, Springer-Verlag, 1985.

[185] J. Reppy, *CML: A higher-order concurrent language*, in Programming Language Design and Implementation, SIGPLAN, ACM, 1991, pp. 293–259.

[186] H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill Book Co., 1967.

[187] A. Roscoe, *The Theory and Practice of Concurrency*, Prentice-Hall International, 1998.

[188] K. Rosenthal, *Quantales and their Applications*, Research Notes in Mathematics, Pitman, London, 1990.

[189] J. Rutten, *Deriving denotational models for bisimulation from structured operational semantics*, in Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group, J. d. Bakker and J. Rutten, eds., World Scientific, 1992, pp. 425–441.

[190] ——, *Nonwellfounded sets and programming language semantics*, in Brookes et al. [61], pp. 193–206.

[191] ——, *Processes as terms: non-well-founded models for bisimulation*, Mathematical Structures in Computer Science, 2 (1992), pp. 257–275.

[192] J. Rutten and D. Turi, *Initial algebra and final coalgebra semantics for concurrency*, in de Bakker et al. [73], pp. 530–581.

[193] A. Salomaa, *Theory of Automata*, vol. 100 of International Series of Monographs in Pure and Applied Mathematics, Pergamon Press, Oxford, 1969.

[194] D. Sands, *From SOS rules to proof principles: An operational metatheory for functional languages*, in Conference Record 24th ACM Symposium on Principles of Programming Languages, Paris, France, 1997, pp. 428–441.

[195] D. Sangiorgi, *The lazy lambda calculus in a concurrency scenario*, Information and Computation, 111 (1994), pp. 120–153.

[196] ——, *$\pi I$: A symmetric calculus based on internal mobility*, in Mosses et al. [166], pp. 172–186.

[197] S. Schneider, *An operational semantics for timed CSP*, Information and Computation, 116 (1995), pp. 193–213.

[198] D. Scott and C. Strachey, *Towards a mathematical semantics for computer languages*, in Proceedings Symposium on Computers and Automata, vol. 21 of Microwave Research Institute Symposia Series, 1971.

[199] R. d. Simone, *Calculabilité et Expressivité dans l'Algèbre de Processus Parallèles* Meije, thèse de 3$^e$ cycle, Univ. Paris 7, 1984.

[200] ——, *On* Meije *and SCCS: infinite sum operators vs. non-guarded definitions*, Theoretical Comput. Sci., 30 (1984), pp. 133–138.

[201] ——, *Higher-level synchronising devices in* Meije–*SCCS*, Theoretical Comput. Sci., 37 (1985), pp. 245–267.

[202] R. d. Simone and D. Vergamini, *Aboard AUTO*, Tech. Rep. 111, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1989.

[203] A. Simpson, *Compositionality via cut-elimination: Hennessy-Milner logic for an arbitrary GSOS*, in Proceedings 10th Symposium on Logic in Computer Science, San Diego, California, 1995, IEEE Computer Society Press, pp. 420–430.

[204] S. Smith, *From operational to denotational semantics*, in Brookes et al. [61], pp. 54–76.

[205] M. Smyth, *Powerdomains*, J. Comput. System Sci., 16 (1978), pp. 23–36.

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

[207] C. Stirling, *Modal logics for communicating systems*, Theoretical Comput. Sci., 49 (1987), pp. 311–347.

[208] ——, *A generalization of Owicki-Gries's Hoare logic for a concurrent while-language*, Theoretical Comput. Sci., 58 (1988), pp. 347–359.

[209] ——, *Modal and temporal logics*, in Handbook of Logic in Computer Science, S. Abramsky, D. Gabbay, and T. Maibaum, eds., vol. 2, Oxford University Press, 1992, pp. 477–563.

[210] ——, *Modal and temporal logics for processes*, in Moller and Birtwistle [164], pp. 149–237.

[211] V. Stoltenberg-Hansen, I. Lindström, and E. Griffor, *Mathematical Theory of Domains*, Cambridge Tracts in Theoretical Computer Science 22, Cambridge University Press, 1994.

[212] A. Stoughton, *Fully abstract models of programming languages*, Research Notes in Theoretical Computer Science, Pitman, London, 1988.

[213] ——, *Substitution revisited*, Theoretical Comput. Sci., 59 (1988), pp. 317–325.

[214] D. Turi and G. Plotkin, *Towards a mathematical operational semantics*, in Proceedings 12th Symposium on Logic in Computer Science, Warsaw, Poland, 1997, IEEE Computer Society Press, pp. 280–291.

[215] I. Ulidowski, *Equivalences on observable processes*, in LICS92 [141], pp. 148–159.

[216] ——, *Axiomatisations of weak equivalences for De Simone languages*, in Proceedings 6th Conference on Concurrency Theory, Philadelphia, PA, I. Lee and S. Smolka, eds., vol. 962 of Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 219–233.

[217] ——, *Finite axiom systems for testing preorder and De Simone process languages*, in Wirsing and Nivat [235], pp. 210–224.

[218] I. Ulidowski and I. Phillips, *Formats of ordered SOS rules with silent actions*, in Proceedings 7th Conference on Theory and Practice of Software Development, Lille, France, M. Bidoit and M. Dauchet, eds., vol. 1214 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 297–308.

[219] F. Vaandrager, *Process algebra semantics of POOL*, in Baeten [21], pp. 173–236.

[220] ——, *On the relationship between process algebra and input/output automata (extended abstract)*, in Proceedings 6th Symposium on Logic in Computer Science, Amsterdam, The Netherlands, IEEE Computer Society Press, 1991, pp. 387–398.

[221] ——, *Expressiveness results for process algebras*, in de Bakker et al. [72], pp. 609–638.

[222] M. van den Brand, P. Klint, and C. Verhoef, *Reverse engineering and system renovation – annotated bibliography*, Software Engineering Notes, 22 (1997), pp. 56–67.

[223] J. J. Vereijken, *Discrete-Time Process Algebra*, PhD thesis, Eindhoven University of Technology, 1997.

[224] C. Verhoef, *An operator definition principle (for process algebras)*, Report P9105, Programming Research Group, University of Amsterdam, 1991.

[225] ——, *Linear Unary Operators in Process Algebra*, PhD thesis, University of Amsterdam, 1992.

[226] ——, *A congruence theorem for structured operational semantics with predicates and negative premises*, in Jonsson and Parrow [133], pp. 433–448. Preliminary version of [228].

[227] ——, *A general conservative extension theorem in process algebra*, in Proceedings IFIP Working Conference on Programming Concepts, Methods and Calculi, San Miniato, Italy, E.-R. Olderog, ed., IFIP Transactions A-56, Elsevier, 1994, pp. 149–168.

[228] ——, *A congruence theorem for structured operational semantics with predicates and negative premises*, Nordic Journal of Computing, 2 (1995), pp. 274–302.

[229] J. Vrancken, *The algebra of communicating processes with empty process*, Theoretical Comput. Sci., 177 (1997), pp. 287–328.

[230] D. Walker, *Automated analysis of mutual exclusion algorithms using CCS*, Formal Aspects of Computing, 1 (1989), pp. 273–292.

[231] ——, *Bisimulation and divergence*, Information and Computation, 85 (1990), pp. 202–241.

[232] D. Watt, *Programming Concepts and Paradigms*, Prentice Hall, 1990.

[233] S. Weber, B. Bloom, and G. Brown, *Compiling Joy into silicon: an exercise in applied structural operational semantics*, in de Bakker et al. [72], pp. 639–659.

[234] G. Winskel, *A complete proof system for SCCS with modal assertions*, Fundamenta Informaticae, IX (1986), pp. 401–420.

[235] M. Wirsing and M. Nivat, eds., *Proceedings 5th Conference on Algebraic Methodology and Software Technology,* Munich, Germany, vol. 1101 of Lecture Notes in Computer Science, Springer-Verlag, 1996.

[236] A. Wright and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation, 115 (1994), pp. 38–94.