# Formalizing programming variables in process algebra

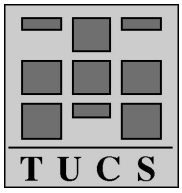
# Formalizing Programming Variables in Process Algebra


## Jos C.M. Baeten

Formal Methods Group
Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513
NL–5600 MB Eindhoven
The Netherlands
josb@win.tue.nl

## Victor Bos

Software Construction Laboratory
Turku Centre for Computer Science
Lemminkäisenkatu 14 A
FIN–20520 Turku
Finland
vbos@abo.fi



**Turku Centre for Computer Science**
**TUCS Technical Report No 493**
**December 2002**
**ISBN ISBN 952-12-1094-X**
**ISSN 1239-1891**

**Abstract**

We use an existing ACP-style process algebra as a formal framework for imperative sequential programming. The framework is realized by instantiating this process algebra with a suitable set of atomic actions and providing concrete definitions for the auxiliary functions assumed by this process algebra. In this framework, we can reason algebraically about programs with assignments and programming variables. We show the programming variables obey scoping rules known from existing programming languages. Next, we use the framework to define well known constructs of sequential programming languages, like conditional statements and loops, and show laws characterizing these constructs can be proved using the ACP axioms.

**Keywords:** Process Algebra; Program Algebra; Sequential Programming; Programming Variables

**TUCS Laboratory**
Software Construction Laboratory

# 1 Introduction

Our goal is to investigate how well ACP (*Algebra of Communicating Processes*) [BPS01, Fok00, BW90] is suited as a formal framework to reason about imperative sequential programs. We do this by specializing various parameters (most notably the set of atomic actions) of particular instances of these process algebras. More concretely, we use the state operator $\lambda_s^m(\_)$ of BPA (*Basic Process Algebra*, a sub-algebra of ACP) [BB88, BV95] to define a formal semantics of assignments and programming variable declarations as known from imperative programming languages. We show that the programming variables obey intuitive scoping rules similar to scoping rules of programming languages like Pascal, C, and Java. In addition, we show that standard process algebra constructs, like conditionals [BB90] and iteration, can be used to formalize if..then..else..fi and while..do..od statements. This results in an algebra for imperative sequential programs. We show that the program algebra enables one to reason algebraically about imperative programs. In these programs, a programming variable can be, but need not be, defined in the program. If it is defined in the program, a value is associated to it and we can use the program algebra to prove properties depending on the value of this variable. If it is not defined in the program, we can use the program algebra to prove properties not depending on the value of this variable. Thus, the program algebra provides a framework to reason about programs with defined as well as undefined variables.

Various algebraic frameworks to analyze imperative programs have been described in literature. In [HJ98, Chapter 5], an *Algebra of Programs* is described in which a program is defined as a predicate transformer written according to a certain syntax. The algebraic laws according to which programs can be manipulated can be proved using this predicate transformer semantics. In our approach, programs are just syntactic entities governed by certain algebraic laws postulated as axioms. Another difference is that the set of program variables in the Algebra of Programs is fixed, whereas in our approach, program variables can be declared explicitly (using the state operator $\lambda_s^m(\_)$). Furthermore, variables, states, and substitution are treated rather informally, whereas in our approach they are formalized in equational logic. Another well known method to define an algebra of programs is to use Kleene algebra [Koz98, CKS96]. Also here, an explicit treatment of programming variables is usually not included. Another difference between our approach and Kleene algebra is the zero-element for sequential composition. In Kleene algebra, we have $0 \cdot x = 0 = x \cdot 0$, which shows that 0 is both a left and right zero-element of sequential composition. In our setting, which is just a specialization of an ACP-style process algebra with deadlock, we only have $\delta \cdot x = \delta$, which shows $\delta$ is a left zero-element of sequential composition; there is no right zero-element for sequential composition. In [BW99], several loop constructs of programming languagea are described algebraically. The formalization is based on the refinement calculus [BW98]. An algebraic framework for imperative sequential code has been described in [BL00, BL02]. In this framework a program is essentially a (possibly infinite) sequence of instructions. In PGA, the most basic ProGram Algebra the authors discuss, program flow is controlled by sequential composition, jump instructions, and test instructions. An important difference with our work is that PGA is deterministic whereas we allow nondeterminism. Another, maybe even more important, difference is that behavioral equivalence on PGA programs is not a congruence, whereas in our setting, behavioral equivalence is ordinary strong bisimulation and is shown to be a congruence for the process algebras we use.

In this paper we will not consider concurrent programs. However, since process algebras were developed to study concurrency, we expect that our approach can be extended to analyze

features of concurrent programming languages, for instance, multi-threaded programs.

This paper is organised as follows. In Sect. 2, we provide formal specifications of programming variables and expressions. In Sect. 3, the process algebra $\mathrm{BPA}_{\delta\varepsilon\lambda}$, which is the starting point of our investigation, will be described. Sect. 4 discusses how we can instantiate $\mathrm{BPA}_{\delta\varepsilon\lambda}$ in order to formalize assignments. Next, in Sect. 5, $\mathrm{BPA}_{\delta\varepsilon\lambda}$ is extended with conditionals and iteration. This extension, which is not new, is needed to formalize if..then..else..fi and while..do..od statements. The main result of this paper is presented in Sect. 6. It consists of a process algebraic formalization of a simple programming language together with derivations of characteristic laws about the constructs of this language. Finally in Sect. 7, we draw conclusions.

## 2 Programming Variables and Expressions

Since programming variables take values from a particular data domain, we give a general equational specification of programming variables of a particular data domain. We illustrate this specification by extending it to a concrete specification of boolean expressions and natural number expressions. The equational specification formalism we use is *Membership Equational Logic* (MEL) [Mes98]. MEL features conditional axioms and subsorts. We remark that any other equational specification formalism featuring conditional axioms and subsorts would probably be equally suited for our purposes. In our MEL specifications, we use only one *kind* $\mathcal{K}$. Consequently, all sorts and terms are of this kind $\mathcal{K}$. Therefore, we need not, and will not, mention $\mathcal{K}$ explicitly in the MEL specifications. Sorts are atomic entities[1] and terms can be atomic or structured. The syntax of terms is defined by constants and operator definitions:

| | |
|---|---|
| $c : \mathit{Term}$ | Constant definition |
| $\_ \oplus \_ : \mathit{Term}$ | Binary operator definition |

Note that *Term* is not a sort (or a kind); it is just a keyword we use to define terms. The underscores in operator definitions define the location of the arguments. Arguments are supposed to be terms, too.

MEL has two kinds of axioms: *conditional membership assertions* and *conditional equations*. The syntax of these axioms is defined by the classes CMA and CEA of the following grammar. In this grammar, `term` denotes the syntactic class of terms and `sort` denotes the syntactic class of sorts. Both of these classes are user-definable and therefore left undefined for the moment.

$$
\begin{array}{lcllllcl}
\mathit{CMA} & ::= & \mathit{MA} & & & \mathit{EA} & ::= & \texttt{term} = \texttt{term} \\
 & | & \mathit{MA} & \mathit{AL} & & \mathit{MA} & ::= & \texttt{term} : \texttt{sort} \\
\mathit{CEA} & ::= & \mathit{EA} & & & \mathit{A} & ::= & \mathit{MA} \mid \mathit{EA} \\
 & | & \mathit{EA} & \mathit{AL} & & \mathit{AL} & ::= & \mathit{A} \mid \mathit{ALA}
\end{array}
$$

Membership assertions define the sort(s) of terms. Equations define equality between terms. For instance, let $\oplus$ be a binary operator, $t_1$ and $t_2$ be terms, $x_1$ and $x_2$ be logical variables, and $S$ a sort. The conditional membership assertion

$$x_1 \oplus x_2 : S \quad x_1 : S, x_2 : S$$

[1] In MEL [Mes98], parameterized specifications are allowed and these give rise to structured sorts, e.g., a sort of lists of integers is typically denoted by List[Int]. In this paper, we will not use parameterized specifications.

specifies that every term of the form $t_1 \oplus t_2$ is of sort $S$ if both $t_1$ and $t_2$ are of sort $S$. As is common in MEL, syntactic sugar in the operator definition syntax enables one to reduce the number of explicit membership assertions. For instance, the following is syntactic sugar for the binary operator definition and the conditional membership assertion given above.

$$\_ \oplus \_ : S \times S \to S$$

Membership assertions can be used to define subsort relations too. For instance, the following conditional membership assertions states that every term of sort $S_1$ is of sort $S_2$ too, thereby making $S_1$ a subsort of $S_2$.

$$x : S_2 \qquad x : S_1$$

Cyclic subsort relations are not allowed. Subsort definitions as shown here are usually abbreviated by

$$S_1 \subseteq S_2.$$

The MEL specifications we present in this paper have an initial algebra semantics. As a structuring mechanism for specifications we use syntactic replacement, or 'copy-paste'. That is, if a specification imports another specification, using syntax

**Import** *specname*,

this means specification *specname* should be 'copy-&-pasted' at that position. Only then can the initial algebra of the complete specification be determined.

The MEL specification of programming variables, expressions, and states is given in Table 1. In this theory, a sort $E$ of expressions, a sort $V$ of programming variables, and a sort $S$ of states is defined. In addition, an operator $\_ \mapsto \_$ is defined that builds a state from a programming variable and an expression. Finally, an operator $\_ \circ \_$ is defined that applies a state to an expression. The result of applying a state to an expression is again an expression. Axiom S1 states that $V$ is a subsort of $E$; every variable is an expression. Axiom S2 states that the value of a variable in a state in which that variable is defined, is the value associated to it by that state. Axiom S3 states that the value of a variable in a state in which it is undefined, is just the variable itself. Finally, axiom S4 enables one to merge two state applications into one state application. It says that if we replace the occurrences of $v$ by $e'$ in $e''$ and then replace the remaining occurrences of $v$ (which had to be present in $e'$) by $e$, then we might as well first replace the occurrences of $v$ in $e'$ by $e$, which results in $(v \mapsto e) \circ e'$, and then replace the occurrences of $v$ in $e''$ by this expression.

The *Expressions* theory is general and has to be specialized to define programming variables and expressions of a concrete data type. To illustrate this, we will define theories of boolean and natural number expressions. The first step is to define a normal equational theory of the boolean data type and the natural number data type, see Table 2.

A theory of boolean expressions and natural number expressions results from integrating the general Expression theory of Table 1 with the theories of Table 2. The integration should define how application of states on expressions interacts with the boolean operators and the natural number operators. Table 3 shows the resulting theory. For readability we included the variable definitions, although formally this is not needed, because the variables are defined in the imported specifications. We claim that the initial algebra of this specification does not contain elements that were not present in the union of the initial algebras of the specifications

**Specification** Expressions

**Sort** $V, E, S$ **variables**

$v, v' : V$

**Operators** $e, e' : E$

$\_ \mapsto \_ : V \times E \to S$

$\_ \circ \_ : S \times E \to E$

**Axioms**

S1 $V \subseteq E$

S2 $(v \mapsto e) \circ v = e$

S3 $(v \mapsto e) \circ v' = v'$ $v \not\equiv v'$

S4 $(v \mapsto e) \circ (v \mapsto e') \circ e'' = (v \mapsto (v \mapsto e) \circ e') \circ e''$

Table 1: Equational specification of expressions

of Table 2. This can be seen by first observing that there are no constants or operators to build terms of sorts $V_{\mathbb{B}}$ and $V_{\mathbb{N}}$. So, although there are sorts of programming variables, there are no programming variables (in the initial algebra). Furthermore, according to axioms BE4, BE5, and NE4, application of a state on a constant is the identity function. Finally, note that according to axioms BE6–BE9 and axioms NE5–NE8, application of a state distributes over all operators.

To use programming variables in a process description, they have to be defined as constants of the sort $V_{\mathbb{B}}$ or $V_{\mathbb{N}}$. Since there can be arbitrary many programming variables, each having its own distinct name, definition of these constants can be done only with a concrete application in mind. To illustrate how this is done, suppose we want to write a program that manipulates the boolean programming variables $vb_0$ and $vb_1$ as well as the natural number programming variables $v_0$ and $v_1$, then we should define a theory as given in Table 4. Note that the initial algebra of this specification contains elements that are not in the initial algebra of the theory of Table 3. These new elements are interpretations of terms containing some programming variables that cannot be rewritten into terms not containing programming variables. Examples of such terms are $vb_0$, $vb_0 \wedge vb_1$, $v_0$, and $v_0 - v_1$.

# 3 The Process Algebra $\mathrm{BPA}_{\delta\varepsilon\lambda}$

In this section, we present the existing process theory $\mathrm{BPA}_{\delta\varepsilon\lambda}$ [BB88]. Here, $\delta$ and $\varepsilon$ are special constants, with $\delta \neq \varepsilon$, and $\lambda$ is an operator (see below). This theory is parameterized by a set $A$ of atomic actions (with $\{\delta, \varepsilon\} \cap A = \emptyset$), a set $M$ of machines, a set $S$ of machine states, a function $act : (A \cup \{\delta\}) \times M \times S \to (A \cup \{\delta\})$, and a function $\mathit{eff} : (A \cup \{\delta\}) \times M \times S \to S$ . In addition, it has a constant, $\delta$ denoting the deadlock process, a constant $\varepsilon$ denoting the empty process, a binary alternative composition operator, $\_ + \_$, a binary sequential composition operator, $\_ \cdot \_$, and a unary state operator, $\lambda_s^m(\_)$, where $m \in M$ and $s \in S$. The sequential composition operator binds stronger than the alternative composition operator. Intuitively, process $\lambda_s^m(x)$ is a machine $m$ in state $s$, representing its memory, that tries to execute $x$, representing its program. The algebraic specification of $\mathrm{BPA}_{\delta\varepsilon\lambda}$ is given in Table 5. The operational semantics of $\mathrm{BPA}_{\delta\varepsilon\lambda}$ is given in Table 6. It defines for all actions $a \in A$ a binary transition relation $\_ \xrightarrow{a} \_$ on processes as well as a termination predicate $\_\downarrow$ on processes. If

**Specification** Booleans

**Sort** $\mathbb{B}$

**Operators**

$true : \to \mathbb{B}$
$false : \to \mathbb{B}$
$\neg\_ : \mathbb{B} \to \mathbb{B}$
$\_ \wedge \_ : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
$\_ \vee \_ : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
$\_ \Rightarrow \_ : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

**variables** $b, b', b'' : \mathbb{B}$

**Axioms**

| | |
|---|---|
| B1 | $\neg true = false$ |
| B2 | $\neg false = true$ |
| B3 | $\neg\neg b = b$ |
| B4 | $true \wedge b = b$ |
| B5 | $false \wedge b = false$ |
| B6 | $b \wedge b = b$ |
| B7 | $b \wedge b' = b' \wedge b$ |
| B8 | $(b \wedge b') \wedge b'' = b \wedge (b' \wedge b'')$ |
| B9 | $b \vee b' = \neg(\neg b \wedge \neg b')$ |
| B10 | $b \Rightarrow b' = \neg b \vee b'$ |

**Specification** Natural numbers

**Sort** $\mathbb{N}$

**Operators**

$0 : \to \mathbb{N}$
$\mathsf{S}\_ : \mathbb{N} \to \mathbb{N}$
$\mathsf{P}\_ : \mathbb{N} \to \mathbb{N}$
$\_ + \_ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$\_ - \_ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

**variables** $n, n' : \mathbb{N}$

**Axioms**

| | |
|---|---|
| N1 | $\mathsf{P}\,0 = 0$ |
| N2 | $\mathsf{P}(\mathsf{S}\,n) = n$ |
| N3 | $0 + n = n$ |
| N4 | $(\mathsf{S}\,n) + n' = \mathsf{S}(n + n')$ |
| N5 | $n + n' = n' + n$ |
| N6 | $0 - n = 0$ |
| N7 | $(\mathsf{S}\,n) - 0 = \mathsf{S}\,n$ |
| N8 | $(\mathsf{S}\,n) - (\mathsf{S}\,n') = n - n'$ |

Table 2: Equational specifications of the booleans and the natural numbers

$x \xrightarrow{a} y$ holds, $x$ can evolve into $y$ by executing $a$. If $x\!\downarrow$ holds, $x$ can terminate successfully.

Note that axioms SO1 and SO2 are redundant, because they can be derived from the definition of *act* and axioms SO3, SO4, and A8. We have included SO1 and SO2 for historical reasons.

Strong-bisimulation is defined as usual [Mil83, Par81]: A binary relation $R$ on processes is a *strong-bisimulation* if for all pairs $(p, q) \in R$ the following conditions hold:

1. $p\!\downarrow \Leftrightarrow q\!\downarrow$
2. $\forall a, p' : p \xrightarrow{a} p' \Rightarrow \exists q' : q \xrightarrow{a} q' \wedge (p', q') \in R$
3. $\forall a, q' : q \xrightarrow{a} q' \Rightarrow \exists p' : p \xrightarrow{a} p' \wedge (p', q') \in R$

It is well known that, for closed terms, the axioms of $\text{BPA}_{\delta\varepsilon\lambda}$ are sound and complete with respect to strong-bisimulation equivalence. Completeness follows from the fact that the $\lambda$ operator can be eliminated and from the fact that axioms A1–A9 form a complete axiomatization for $\text{BPA}_{\delta\varepsilon}$(i.e., $\text{BPA}_{\delta\varepsilon\lambda}$ without the $\lambda$ operator).

A standard proof technique in ACP is based on the notion of *basic terms*. Basic terms form a syntactically defined subclass of all process terms. Furthermore, it is known that for every process term there exists a bisimilar basic term. Consequently, basic terms can be used

**Specification** Boolean and natural number expressions

| **Import** | **Sorts** $V_{\mathbb{B}}, V_{\mathbb{N}}$ |
|---|---|
| Expressions | **variables** |
| Booleans | $b, b', b'' : \mathbb{B}$ |
| Natural numbers | $n, n' : \mathbb{N}$ |

**Axioms**

| | | | |
|---|---|---|---|
| BE1 | $V_{\mathbb{B}} \subseteq V$ | NE1 | $V_{\mathbb{N}} \subseteq V$ |
| BE2 | $V_{\mathbb{B}} \subseteq \mathbb{B}$ | NE2 | $V_{\mathbb{N}} \subseteq \mathbb{N}$ |
| BE3 | $\mathbb{B} \subseteq E$ | NE3 | $\mathbb{N} \subseteq E$ |
| BE4 | $s \circ \mathit{true} = \mathit{true}$ | NE4 | $s \circ 0 = 0$ |
| BE5 | $s \circ \mathit{false} = \mathit{false}$ | NE5 | $s \circ (\mathsf{S}\, n) = \mathsf{S}(s \circ n)$ |
| BE6 | $s \circ (\neg b) = \neg(s \circ b)$ | NE6 | $s \circ (\mathsf{P}\, n) = \mathsf{P}(s \circ n)$ |
| BE7 | $s \circ (b \wedge b') = (s \circ b) \wedge (s \circ b')$ | NE7 | $s \circ (n + n') = (s \circ n) + (s \circ n')$ |
| BE8 | $s \circ (b \vee b') = (s \circ b) \vee (s \circ b')$ | NE8 | $s \circ (n - n') = (s \circ n) - (s \circ n')$ |
| BE9 | $s \circ (b \Rightarrow b') = (s \circ b) \Rightarrow (s \circ b')$ | | |

Table 3: Equational specification of boolean and natural number expressions

**Specification** Programming variables
**Import**
Boolean and natural number expressions
**Operators**
$vb_0 : \rightarrow V_{\mathbb{B}}$
$vb_1 : \rightarrow V_{\mathbb{B}}$
$v_0 : \rightarrow V_{\mathbb{N}}$
$v_1 : \rightarrow V_{\mathbb{N}}$

Table 4: Programming variables

in structural induction proofs. Since we use this technique in some of our proofs, we introduce the class of basic terms here. Basic terms are defined inductively as follows.

- $\delta$ and $\varepsilon$ are basic terms,
- every $a \in A$ is a basic term,
- if $x$ is a basic term, then for all $a \in A$, $a \cdot x$ is a basic term,
- if $x$ and $y$ are basic terms such that, then $x + y$ is a basic term.

# 4 Instantiating Parameters of $\mathrm{BPA}_{\delta\varepsilon\lambda}$

In this section, we instantiate the parameters of $\mathrm{BPA}_{\delta\varepsilon\lambda}$ with the goal to get a basic process algebra with programming variables. We show that although a straightforward instantiation

**Parameters**

$A$ : set of actions such that $\{\delta, \varepsilon\} \cap A = \emptyset$
$M$ : set of machines
$S$ : set of machine states
$act : (A \cup \{\delta\}) \times M \times S \to (A \cup \{\delta\})$
$\mathit{eff} : (A \cup \{\delta\}) \times M \times S \to S$

**Signature**

| | |
|---|---|
| $\delta :\to P$ | |
| $\varepsilon :\to P$ | |
| $a :\to P$ | $a \in A$ |
| $\_ + \_ : P \times P \to P$ | |
| $\_ \cdot \_ : P \times P \to P$ | |
| $\lambda_s^m(\_) :\to P$ | $m \in M, s \in S$ |

**Axioms**

| | | | |
|---|---|---|---|
| A1 | $x + y = x + x$ | A6 | $x + \delta = x$ |
| A2 | $(x + y) + z = x + (y + z)$ | A7 | $\delta \cdot x = \delta$ |
| A3 | $x + x = x$ | A8 | $x \cdot \varepsilon = x$ |
| A4 | $(x + y) \cdot z = x \cdot z + y \cdot z$ | A9 | $\varepsilon \cdot x = x$ |
| A5 | $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | | |
| SO1 | $\lambda_s^m(a) = act(a, m, s)$ | | $a \in A \cup \{\delta\}$ |
| SO2 | $\lambda_s^m(\delta) = \delta$ | | |
| SO3 | $\lambda_s^m(\varepsilon) = \varepsilon$ | | |
| SO4 | $\lambda_s^m(a \cdot x) = act(a, m, s) \cdot \lambda_{\mathit{eff}(a,m,s)}^m(x)$ | | $a \in A \cup \{\delta\}$ |
| SO5 | $\lambda_s^m(x + y) = \lambda_s^m(x) + \lambda_s^m(y)$ | | |

Table 5: The $\text{BPA}_{\delta\varepsilon\lambda}$ theory

of the set $A$ of atomic actions with assignment actions suffices to formalize programming variables in a process algebra setting, it fails to formalize programming variables that obey conventional scoping rules. We propose a solution based on two types of actions: assignment actions and update actions. Assignment actions have not yet changed a (local) state and they are renamed into update actions once they have updated a (local) state. In this way, assignments affect at most one state and therefore behave properly if they occur in nested state operator processes. This is another solution as the one proposed in [BK02]. There, an explicit stacking mechanism was used to ensure assignments have local effect only. The current approach is simpler, because the explicit stacking mechanism is not needed.

We use the MEL specification Expressions of Table 1 to describe the data domain. This specification has a sort $E$ of expressions, a sort $V$ of programming variables, and a sort $S$ of states. Typical expressions are denoted by $e$, $e'$, ..., typical programming variables are denoted by $v$, $v'$, ..., and typical states are denoted by $s$, $s'$, .... As a first try, the set $A$ of $\text{BPA}_{\delta\varepsilon\lambda}$ is instantiated by the set of assignment actions:

$$A \mathrel{\hat{=}} \{v := e \mid v \in V, e \in E\}$$

here, $\mathrel{\hat{=}}$ denotes 'equality per definition'. The intended meaning of an assignment action is

$$1\frac{}{\varepsilon\downarrow}\quad 2\frac{}{a \xrightarrow{a} \varepsilon}\quad 3\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}\quad 4\frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}\quad 5\frac{x\downarrow}{(x + y)\downarrow}\quad 6\frac{y\downarrow}{(x + y)\downarrow}\quad 7\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$$
$$8\frac{x\downarrow, y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}\quad 9\frac{x\downarrow, y\downarrow}{(x \cdot y)\downarrow}\quad 10\frac{x \xrightarrow{a} x', act(a, m, s) \neq \delta}{\lambda_s^m(x) \xrightarrow{act(a,m,s)} \lambda_{eff(a,m,s)}^m(x')}\quad 11\frac{x\downarrow}{\lambda_s^m(x)\downarrow}$$

Table 6: SOS rules of $\text{BPA}_{\delta\varepsilon\lambda}$

that it assigns the 'value' $e$ to $v$.

Recall that the parameters of the state operator $\lambda_s^m(\_)$ are a machine $m \in M$ and a state $s \in S$. To keep it simple, we will use only one machine. Therefore, we instantiate $M$ with a singleton set. Since there is only one machine, we will drop the $m$ in $\lambda_s^m(x)$ and just write $\lambda_s(x)$.

Finally, we instantiate the *act* and *eff* functions. In the definition of these functions, we ignore the machine argument $m \in M$ for reasons explained above.

$$\begin{array}{lcl} act(\delta, s) & \hat{=} & \delta \\ act(v := e, v \mapsto e') & \hat{=} & v := (v \mapsto e') \circ e \\ act(v := e, v' \mapsto e') & \hat{=} & v := (v' \mapsto e') \circ e \quad \textbf{if } v \not\equiv v' \end{array}$$

$$\begin{array}{lcl} eff(\delta, s) & \hat{=} & s \\ eff(v := e, v \mapsto e') & \hat{=} & v \mapsto (v \mapsto e') \circ e \\ eff(v := e, v' \mapsto e') & \hat{=} & v' \mapsto e' \quad \textbf{if } v \not\equiv v' \end{array}$$

$\text{BPA}_{\delta\varepsilon\lambda}$ instantiated as described above results in a process algebra of processes built up from assignment actions. We will now discuss several examples. In these examples, we use the specification of Table 4 to write boolean expressions and natural number expressions. In addition, we use the following abbreviations for natural number expressions: $1 \hat{=} \mathsf{S}\,0$, $2 \hat{=} \mathsf{S}\,1$, and $3 \hat{=} \mathsf{S}\,2$

Consider process $\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)$. The transition $\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0 := 2} \lambda_{v_0 \mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3)$ can be derived as follows.

$true$

$\Rightarrow$ {Rule 2}

$v_0 := v_0 + 1 \xrightarrow{v_0 := v_0 + 1} \varepsilon$

$\Rightarrow$ {Rule 7}

$v_0 := v_0 + 1 \cdot v_0 := v_0 + 3 \xrightarrow{v_0 := v_0 + 1} \varepsilon \cdot v_0 := v_0 + 3$

$\Rightarrow$ {Rule 10 and definition of *act* gives: $act(v_0 := v_0 + 1, v_0 \mapsto 1) = v_0 := 2$}

$\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0 := 2} \lambda_{eff(v_0 := v_0 + 1, v_0 \mapsto 1)}(\varepsilon \cdot v_0 := v_0 + 3)$

$\equiv$ {Definitions of *eff*}

$\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0 := 2} \lambda_{v_0 \mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3)$

As can be seen, variables are updates as expected. However, there is a problem with nested state operators. Consider process $\lambda_{v_0 \mapsto 0}(\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3))$. The transition $\lambda_{v_0 \mapsto 0}(\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0 := 2} \lambda_{v_0 \mapsto 2}(\lambda_{v_0 \mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3))$, in which one assignment updates two variables, can be derived as follows.

$true$

$\Rightarrow$ {previous derivation}

$\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0:=2} \lambda_{v_0\mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3)$

$\Rightarrow$ {Rule 10 and definition of *act* gives: $act(v_0 := 2, v_0 \mapsto 0) = v_0 := 2$}

$\lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0:=2} \lambda_{\mathit{eff}(v_0:=2,v_0\mapsto 0)}(\lambda_{v_0\mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3))$

$\equiv$ {Definitions of *eff*}

$\lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0:=2} \lambda_{v_0\mapsto 2}(\lambda_{v_0\mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3))$

Obviously, this is not the behaviour we want. The outermost $v_0$ should be hidden by the innermost $v_0$ and, consequently, there should be no way the assignment $v_0 := v_0 + 1$ can affect the outermost occurrence of $v_0$. To solve the problem, we add a new kind of atomic action, called *update* actions, and we change the definitions of *act* and *eff*. Intuitively, update actions are assignment actions that have been effected already. That is, after an assignment action has updated a variable, it is renamed into an update action. Update actions are denoted by $v \leftarrow e$. The new definition of $A$ reads

$$A \mathrel{\hat{=}} \{v := e \mid v \in V, e \in E\} \cup \{v \leftarrow e \mid v \in V, e \in E\}.$$

The *act* and *eff* functions are redefined as follows.

$$\begin{array}{lcll}
act(\delta, s) & \hat{=} & \delta & \\
act(v := e, v \mapsto e') & \hat{=} & v \leftarrow (v \mapsto e') \circ e & \\
act(v := e, v' \mapsto e') & \hat{=} & v := (v' \mapsto e') \circ e & \textbf{if } v \not\equiv v' \\
act(v \leftarrow e, s) & \hat{=} & v \leftarrow s(e) &
\end{array}$$

$$\begin{array}{lcll}
\mathit{eff}(\delta, s) & \hat{=} & s & \\
\mathit{eff}(v := e, v \mapsto e') & \hat{=} & v \mapsto (v \mapsto e') \circ e & \\
\mathit{eff}(v := e, v' \mapsto e') & \hat{=} & v' \mapsto e' & \textbf{if } v \not\equiv v' \\
\mathit{eff}(v \leftarrow e, s) & \hat{=} & s &
\end{array}$$

Using these definitions of *act* and *eff*, we can prove

$$\text{AC1} \quad act(act(a, v \mapsto e), v \mapsto e') = act(a, v \mapsto (v \mapsto e') \circ e).$$

Later, we use this equality to prove some properties about processes.

**Proof AC1** We have to prove $act(act(a, v \mapsto e), v \mapsto e') = act(a, v \mapsto (v \mapsto e') \circ e)$, where $a \in (A \cup \{\delta\})$. We distinguish the following cases: $a \equiv \delta$, $a \equiv v' := e''$, and $a \equiv v' \leftarrow e''$.

$a \equiv \delta$. The proof is as follows.

$act(act(\delta, v \mapsto e), v \mapsto e')$

$=$ {Definition of *act*}

$act(\delta, v \mapsto e')$

$=$ {Definition of *act*}

$\delta$

$=$ {Definition of *act*}

$act(\delta, v \mapsto (v \mapsto e') \circ e)$

$a \equiv v' := e''$. There are two sub-cases: $v \equiv v'$ and $v \not\equiv v'$.

$v \equiv v'$. The proof is as follows.

$$
\begin{aligned}
& act(act(v := e'', v \mapsto e), v \mapsto e') \\
= \ & \{\text{Definition of } act\} \\
& act(v \leftarrow (v \mapsto e) \circ e'', v \mapsto e') \\
= \ & \{\text{Definition of } act\} \\
& v \leftarrow (v \mapsto e') \circ (v \mapsto e) \circ e'' \\
= \ & \{\text{Axiom S4}\} \\
& v \leftarrow (v \mapsto (v \mapsto e') \circ e) \circ e'' \\
= \ & \{\text{Definition of } act\} \\
& act(v := e'', v \mapsto (v \mapsto e') \circ e)
\end{aligned}
$$

$v \not\equiv v'$. The proof is as follows.

$$
\begin{aligned}
& act(act(v' := e'', v \mapsto e), v \mapsto e') \\
= \ & \{\text{Definition of } act\} \\
& act(v' := (v \mapsto e) \circ e'', v \mapsto e') \\
= \ & \{\text{Definition of } act\} \\
& v' := (v \mapsto e') \circ (v \mapsto e) \circ e'' \\
= \ & \{\text{Axiom S4}\} \\
& v' := (v \mapsto (v \mapsto e') \circ e) \circ e'' \\
= \ & \{\text{Definition of } act\} \\
& act(v' := e'', v \mapsto (v \mapsto e') \circ e)
\end{aligned}
$$

$a \equiv v' \leftarrow e''$. The proof is as follows.

$$
\begin{aligned}
& act(act(v' \leftarrow e'', v \mapsto e), v \mapsto e') \\
= \ & \{\text{Definition of } act\} \\
& act(v' \leftarrow (v \mapsto e) \circ e'', v \mapsto e') \\
= \ & \{\text{Definition of } act\} \\
& v' \leftarrow (v \mapsto e') \circ (v \mapsto e) \circ e'' \\
= \ & \{\text{Axiom S4}\} \\
& v' \leftarrow (v \mapsto (v \mapsto e') \circ e) \circ e'') \\
= \ & \{\text{Definition of } act\} \\
& act(v' \leftarrow e'', v \mapsto (v \mapsto e') \circ e)
\end{aligned}
$$

Consider again process $\lambda_{v_0 \mapsto 0}(\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3))$. Now, the previous transition, $\lambda_{v_0 \mapsto 0}(\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0 := 2} \lambda_{v_0 \mapsto 2}(\lambda_{v_0 \mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3))$, cannot be derived anymore. Instead, the transition $\lambda_{v_0 \mapsto 0}(\lambda_{v_0 \mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0 \leftarrow 2} \lambda_{v_0 \mapsto 0}(\lambda_{v_0 \mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3))$, in which the outermost occurrence of $v_0$ is properly hidden by the innermost occurrence of $v_0$, can be derived as follows. The third step in the derivation below shows why the previous derivation fails; the assignment action $v_0 := v_0 + 1$ is renamed into an update action $v_0 \leftarrow 2$ and, according to the new definitions of *act* and *eff*, update actions have no effect on the state $s$ in $\lambda_s(x)$ processes.

$$
\begin{aligned}
& true \\
\Rightarrow \ & \{\text{Rule 2}\}
\end{aligned}
$$

$v_0 := v_0 + 1 \xrightarrow{v_0:=v_0+1} \varepsilon$

$\Rightarrow$ {Rule 7}

$v_0 := v_0 + 1 \cdot v_0 := v_0 + 3 \xrightarrow{v_0:=v_0+1} \varepsilon \cdot v_0 := v_0 + 3$

$\Rightarrow$ {Rule 10 and new definition of *act* gives: $act(v_0 := v_0 + 1, v_0 \mapsto 1) = v_0 \leftarrow 2$}

$\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0\leftarrow 2} \lambda_{\mathit{eff}(v_0:=v_0+1, v_0\mapsto 1)}(\varepsilon \cdot v_0 := v_0 + 3)$

$\equiv$ {New definitions of *eff*}

$\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0\leftarrow 2} \lambda_{v_0\mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3)$

$\Rightarrow$ {Rule 10 and new definition of *act* gives: $act(v_0 \leftarrow 2, v_0 \mapsto 0) = v_0 \leftarrow 2$}

$\lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0\leftarrow 2} \lambda_{\mathit{eff}(v_0\leftarrow 2, v_0\mapsto 0)}(\lambda_{v_0\mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3))$

$\equiv$ {New definitions of *eff*}

$\lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0\leftarrow 2} \lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 2}(\varepsilon \cdot v_0 := v_0 + 3))$

This result is confirmed by the result we get by using the axioms of Table 5. For instance, using the axioms, the equality $\lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) = v_0 \leftarrow 2 \cdot \lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 2}(v_0 := v_0 + 3))$ can be derived as follows.

$\lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3))$

$=$ {Axiom SO4}

$\lambda_{v_0\mapsto 0}(act(v_0 := v_0 + 1, v_0 \mapsto 1) \cdot \lambda_{\mathit{eff}(v_0:=v_0+1, v_0\mapsto 1)}(v_0 := v_0 + 3))$

$=$ {Definitions of *act* and *eff*}

$\lambda_{v_0\mapsto 0}(v_0 \leftarrow 2 \cdot \lambda_{v_0\mapsto 2}(v_0 := v_0 + 3))$

$=$ {Axiom SO4}

$act(v_0 \leftarrow 2, v_0 \mapsto 0) \cdot \lambda_{\mathit{eff}(v_0\leftarrow 2, v_0\mapsto 0)}(\lambda_{v_0\mapsto 2}(v_0 := v_0 + 3))$

$=$ {Definitions of *act* and *eff*}

$v_0 \leftarrow 2 \cdot \lambda_{v_0\mapsto 0}(\lambda_{v_0\mapsto 2}(v_0 := v_0 + 3))$

Using the new definitions of *act* and *eff*, identities SO4a–SO4d, see below, can be derived. SO4a and SO4b are specializations of SO4. Identity SO4c shows that update actions cannot change the state, but a state can change an update action. The intuition of identity SO4d is that the value of a variable is determined by the most deeply nested occurrence of a state operator with that variable in its state. So, one might expect $\lambda_{v_0\mapsto e}(\lambda_{v_0\mapsto e'}(x)) = \lambda_{v_0\mapsto e'}(x)$. However, since the value $e'$ is an expression, it might contain variables and $v_0$ could be one of them. If $v_0$ occurs in $e'$, its value it determined by the state of the enclosing state operator, i.e., $\lambda_{v_0\mapsto e}(\_)$. This explains the application $(v_0 \mapsto e)e'$ in the right hand side of SO4d. Identities SO4a–SO4c can be proved straightforwardly using the axioms of $\mathrm{BPA}_{\delta\varepsilon\lambda}$ and the definitions of *act* and *eff*. Identity SO4d is proved by structural induction, which means it is proved only for closed terms.

| | | |
|---|---|---|
| SO4a | $\lambda_{v\mapsto e}(v' := e' \cdot x) = v' := ((v \mapsto e) \circ e') \cdot \lambda_{v\mapsto e}(x)$ | if $v \not\equiv v'$ |
| SO4b | $\lambda_{v\mapsto e}(v := e' \cdot x) = v \leftarrow ((v \mapsto e) \circ e') \cdot \lambda_{v\mapsto((v\mapsto e)\circ e')}(x)$ | |
| SO4c | $\lambda_{v\mapsto e}(v' \leftarrow e' \cdot x) = v' \leftarrow (v \mapsto e)e' \cdot \lambda_{v\mapsto e}(x)$ | |
| SO4d | $\lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(x)) = \lambda_{v\mapsto(v\mapsto e)\circ e'}(x)$ | |

**Proof SO4a**

$\lambda_{v\mapsto e}(v' := e' \cdot x)$

$=$ {Axiom SO4}

$act(v' := e', v \mapsto e) \cdot \lambda_{\mathit{eff}(v':=e', v\mapsto e)}(x)$

$=$ {Definition of *act* and *eff* and $v \not\equiv v'$}

$v' := ((v \mapsto e) \circ e') \cdot \lambda_{v\mapsto e}(x)$

**Proof SO4b**

$\lambda_{v\mapsto e}(v := e' \cdot x)$

$=$ {Axiom SO4}

$act(v := e', v \mapsto e) \cdot \lambda_{\mathit{eff}(v:=e', v\mapsto e)}(x)$

$=$ {Definition of *act* and *eff*}

$v \leftarrow ((v \mapsto e) \circ e') \cdot \lambda_{v\mapsto((v\mapsto e)\circ e')}(x)$

**Proof SO4c**

$\lambda_{v\mapsto e}(v' \leftarrow e' \cdot x)$

$=$ {Axiom SO4}

$act(v' \leftarrow e', v \mapsto e) \cdot \lambda_{\mathit{eff}(v'\leftarrow e', v\mapsto e)}(x)$

$=$ {Definition of *act* and *eff*}

$v' \leftarrow ((v \mapsto e) \circ e') \cdot \lambda_{v\mapsto e}(x)$

**Proof SO4d** We have to prove $\lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(x)) = \lambda_{v\mapsto(v\mapsto e)\circ e'}(x)$. We do this by structural induction on $x$, using the fact that every $\mathrm{BPA}_{\delta\varepsilon\lambda}$ term can is equal to a $\mathrm{BPA}_{\delta\varepsilon\lambda}$-basic term. Basic terms are defined at the end of Sect. 3. This leads to five cases.

$x \equiv \varepsilon$. The proof is as follows.

$\lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(\varepsilon))$

$=$ {Axiom SO3}

$\lambda_{v\mapsto e}(\varepsilon)$

$=$ {Axiom SO3}

$\varepsilon$

$=$ {Axiom SO3}

$\lambda_{v\mapsto(v\mapsto e)\circ e'}(\varepsilon)$

$x \equiv \delta$. The proof is as follows.

$\lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(\delta))$

$=$ {Axiom SO2}

$\lambda_{v\mapsto e}(\delta)$

$=$ {Axiom SO2}

$\delta$

$=$ {Axiom SO2}

$\lambda_{v\mapsto(v\mapsto e)\circ e'}(\delta)$

$x \equiv a$, for some action $a$. The proof is as follows.

$$
\begin{aligned}
& \lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(a)) \\
= \ & \{\text{Axiom SO1}\} \\
& \lambda_{v\mapsto e}(act(a, v \mapsto e')) \\
= \ & \{\text{Axiom SO1}\} \\
& act(act(a, v \mapsto e'), v \mapsto e) \\
= \ & \{\text{Equation AC1}\} \\
& act(a, v \mapsto (v \mapsto e) \circ e') \\
= \ & \{\text{Axiom SO1}\} \\
& \lambda_{v\mapsto (v\mapsto e)\circ e'}(a)
\end{aligned}
$$

$x \equiv a \cdot s$, for some action $a$ and $\text{BPA}_{\delta\varepsilon\lambda}$-basic term $s$. The proof is by case distinction on $a$. Consequently, there are three sub-cases: $a \equiv \delta$; $a \equiv v' := e''$; for some $v'$ and $e''$; and $a \equiv v' \leftarrow e''$ for some $v'$ and $e''$.

$a \equiv \delta$. The proof is as follows.

$$
\begin{aligned}
& \lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(\delta \cdot s)) \\
= \ & \{\text{Axiom A7}\} \\
& \lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(\delta)) \\
= \ & \{\text{Axiom SO2 two times}\} \\
& \delta \\
= \ & \{\text{Axiom SO2}\} \\
& \lambda_{v\mapsto (v\mapsto e)\circ e'}(\delta) \\
= \ & \{\text{Axiom A7}\} \\
& \lambda_{v\mapsto (v\mapsto e)\circ e'}(\delta \cdot s)
\end{aligned}
$$

$a \equiv v' := e''$. We distinguish two cases: $v = v'$ and $v \not\equiv v'$. The proof of the first case is as follows.

$$
\begin{aligned}
& \lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(v := e'' \cdot s)) \\
= \ & \{\text{Axiom SO4}\} \\
& \lambda_{v\mapsto e}(act(v := e'', v \mapsto e') \cdot \lambda_{\mathit{eff}(v:=e'', v\mapsto e')}(s)) \\
= \ & \{\text{Axiom SO4}\} \\
& act(act(v := e'', v \mapsto e'), v \mapsto e) \cdot \lambda_{\mathit{eff}(act(v:=e'', v\mapsto e'), v\mapsto e)}(\lambda_{\mathit{eff}(v:=e'', v\mapsto e')}(s)) \\
= \ & \{\text{Equality AC1}\} \\
& act(v := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(act(v:=e'', v\mapsto e'), v\mapsto e)}(\lambda_{\mathit{eff}(v:=e'', v\mapsto e')}(s)) \\
= \ & \{\text{Definitions of } act \text{ and } \mathit{eff}\} \\
& act(v := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(v\leftarrow (v\mapsto e')\circ e'', v\mapsto e)}(\lambda_{v\mapsto (v\mapsto e')\circ e''}(s)) \\
= \ & \{\text{Definition of } \mathit{eff}\} \\
& act(v := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v\mapsto e}(\lambda_{v\mapsto (v\mapsto e')\circ e''}(s)) \\
= \ & \{\text{Induction hypothesis}\} \\
& act(v := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v\mapsto (v\mapsto e)\circ((v\mapsto e')\circ e'')}(s) \\
= \ & \{\text{Axiom S4}\}
\end{aligned}
$$

$$
\begin{aligned}
& act(v := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v \mapsto (v \mapsto (v \mapsto e) \circ e') \circ e''}(s) \\
= \; & \{\text{Definition of } \mathit{eff}\} \\
& act(v := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(v := e'', v \mapsto (v \mapsto e) \circ e')}(s) \\
= \; & \{\text{Axiom SO4}\} \\
& \lambda_{v \mapsto (v \mapsto e) \circ e'}(v := e'' \cdot s)
\end{aligned}
$$

The proof of the second case, $v \not\equiv v'$ is as follows.

$$
\begin{aligned}
& \lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(v' := e'' \cdot s)) \\
= \; & \{\text{Axiom SO4}\} \\
& \lambda_{v \mapsto e}(act(v' := e'', v \mapsto e') \cdot \lambda_{\mathit{eff}(v' := e'', v \mapsto e')}(s)) \\
= \; & \{\text{Axiom SO4}\} \\
& act(act(v' := e'', v \mapsto e'), v \mapsto e) \cdot \lambda_{\mathit{eff}(act(v' := e'', v \mapsto e'), v \mapsto e)}(\lambda_{\mathit{eff}(v' := e'', v \mapsto e')}(s)) \\
= \; & \{\text{Equality AC1}\} \\
& act(v' := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(act(v' := e'', v \mapsto e'), v \mapsto e)}(\lambda_{\mathit{eff}(v' := e'', v \mapsto e')}(s)) \\
= \; & \{\text{Definitions of } act \text{ and } \mathit{eff} \text{ and } v \not\equiv v'\} \\
& act(v' := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(v' := (v \mapsto e') \circ e'', v \mapsto e)}(\lambda_{v \mapsto e'}(s)) \\
= \; & \{\text{Definition of } \mathit{eff} \text{ and } v \not\equiv v'\} \\
& act(v' := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(s)) \\
= \; & \{\text{Induction hypothesis}\} \\
& act(v' := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v \mapsto (v \mapsto e) \circ e'}(s) \\
= \; & \{\text{Definition of } \mathit{eff} \text{ and } v \not\equiv v'\} \\
& act(v' := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(v' := e'', v \mapsto (v \mapsto e) \circ e')}(s) \\
= \; & \{\text{Axiom SO4}\} \\
& \lambda_{v \mapsto (v \mapsto e) \circ e'}(v' := e'' \cdot s)
\end{aligned}
$$

$a \equiv v' \leftarrow e''$. The proof is as follows.

$$
\begin{aligned}
& \lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(v' \leftarrow e'' \cdot s)) \\
= \; & \{\text{Axiom SO4}\} \\
& \lambda_{v \mapsto e}(act(v' \leftarrow e'', v \mapsto e') \cdot \lambda_{\mathit{eff}(v' \leftarrow e'', v \mapsto e')}(s)) \\
= \; & \{\text{Axiom SO4}\} \\
& act(act(v' \leftarrow e'', v \mapsto e'), v \mapsto e) \cdot \lambda_{\mathit{eff}(act(v' \leftarrow e'', v \mapsto e'), v \mapsto e)}(\lambda_{\mathit{eff}(v' \leftarrow e'', v \mapsto e')}(s)) \\
= \; & \{\text{Equality AC1}\} \\
& act(v' \leftarrow e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(act(v' \leftarrow e'', v \mapsto e'), v \mapsto e)}(\lambda_{\mathit{eff}(v' \leftarrow e'', v \mapsto e')}(s)) \\
= \; & \{\text{Definitions of } act \text{ and } \mathit{eff}\} \\
& act(v' \leftarrow e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(v' \leftarrow (v \mapsto e') \circ e'', v \mapsto e)}(\lambda_{v \mapsto e'}(s)) \\
= \; & \{\text{Definition of } \mathit{eff}\} \\
& act(v' \leftarrow e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(s)) \\
= \; & \{\text{Induction hypothesis}\} \\
& act(v' \leftarrow e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v \mapsto (v \mapsto e) \circ e'}(s) \\
= \; & \{\text{Definition of } \mathit{eff}\} \\
& act(v' \leftarrow e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\mathit{eff}(v' \leftarrow e'', v \mapsto (v \mapsto e) \circ e')}(s) \\
= \; & \{\text{Axiom SO4}\} \\
& \lambda_{v \mapsto (v \mapsto e) \circ e'}(v' \leftarrow e'' \cdot s)
\end{aligned}
$$

This completes the proof for the case $x \equiv a \cdot s$, where $a$ some action and $t$ a basic term.

$x \equiv s + t$, for some $\text{BPA}_{\delta\varepsilon\lambda}$-basic terms $s$ and $t$. The proof is as follows.

$$\begin{aligned}
&\lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(s+t))\\
=\ &\{\text{Axiom SO5 two times}\}\\
&\lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(s)) + \lambda_{v\mapsto e}(\lambda_{v\mapsto e'}(t))\\
=\ &\{\text{Induction hypothesis two times}\}\\
&\lambda_{v\mapsto (v\mapsto e)\circ e'}(s) + \lambda_{v\mapsto (v\mapsto e)\circ e'}(t)\\
=\ &\{\text{Axiom SO5}\}\\
&\lambda_{v\mapsto (v\mapsto e)\circ e'}(s+t)
\end{aligned}$$

# 5 Conditionals and Iteration

In this section we add *conditionals* and *iteration* to $\text{BPA}_{\delta\varepsilon\lambda}$. This extension enables us to formalize programming language constructs like $\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else} \ldots \mathsf{fi}$ and $\mathsf{while} \ldots \mathsf{do} \ldots \mathsf{od}$, as we show in Sect. 6.

The treatment of conditions follows [BB90] closely. Consequently, in the SOS, our transitions will also be labeled by pairs $(a, e_b) \in A \times E_b$, with $E_b$ the set of boolean expressions, see Table 2. In addition, the termination predicate (which is not present in [BB90]) will be labeled with $e_b \in E_b$. Traditionally, conditionals are denoted by $x \lhd e_b \rhd y$, where $x$ and $y$ are processes and $e_b$ a boolean expression. If $e_b = \mathit{true}$, this conditional process behaves like $x$ and if $e_b = \mathit{false}$, it behaves like $y$.

The iteration operator we add to $\text{BPA}_{\delta\varepsilon\lambda}$ is the unary Kleene star [Kle56, CE58] and is denoted by $x^*$. Intuitively, a process $x^*$ executes $x$ zero or more times. In a setting with an empty process, sequential composition, and the unary Kleene star, a binary Kleene star can be defined as follows: $x^*y \mathrel{\hat{=}} x^* \cdot y$. Sewell [Sew94] showed that a finite axiomatization of strong bisimulation for process algebras featuring choice, sequential composition, binary Kleene star and the constant $\delta$ or $\varepsilon$, does not exist. As a result, considerable interest has been shown in complete axiomatizations of either restricted versions of process algebras with the binary Kleene star or restricted forms of the Kleene-star operator [BBP94, Fok94, FZ94, Fok96, AI96, AG95, AFGI96, Gla97].

Finite complete axiomatizations of the Kleene star with conditional axioms *do* exist. The oldest such systems are probably Salomaa's axiomatisations $F_1$ and $F_2$ [Sal66]. If we translate the conditional rule of $F_1$ into our setting, we would get a conditional axiom similar to the one presented in [FZ94]:

$$\text{if } x = y \cdot x + z \text{ and } \neg\, y{\downarrow} \quad \Rightarrow \quad x = y^* \cdot z.$$

Kozen [Koz94] showed that this axiom is not algebraic in the sense that it is not preserved under substitution of terms for actions. He presents another axiom system, again with conditional axioms, which does not have this drawback. Since a complete axiomatization is not our primary goal, we will refrain from using process algebras with conditional axioms.

Table 7 shows the axioms for $\text{BPA}_{\delta\varepsilon\lambda}$ with conditionals and iteration. Table 8 defines its operational semantics. As before, we assume $A \cap \{\delta, \varepsilon\} = \emptyset$ and $\delta \neq \varepsilon$. In addition, we assume $A$, $S$, $\mathit{act}$, and $\mathit{eff}$ to be defined as above. Regarding the axioms defining the

conditionals, we remark that in [BB90] only axioms C1 and C2 are present and it is assumed the other axioms can be derived using induction on the booleans. However, in our setting with programming variables, this is not possible, because boolean expressions may contain programming variables. Therefore, we extended the list of axioms with C3–C12. With respect to axioms R1 and R2, defining the unary Kleene star, we remark the following. Axiom R1 is known from Kleene algebras [Koz98, CKS96, Koz94] and shows the main characteristic of process $x^*$: *zero or more times* $x$. As pointed out in [BBP94], in a setting with a binary Kleene star and an empty process, the unary Kleene star can be defined by $x^*\varepsilon$. Axiom R1 results then from substituting $\varepsilon$ for $y$ in the binary Kleene star axiom SEI1: $x^*y = y + x \cdot (x^*y)$. Axiom R2 is due to [Koz94], where it is called Proposition 7. Axioms SEI2 and SEI3 of [BBP94], translated into our setting with a unary Kleene star, can be proved from the axioms given in Table 7. In fact, SEI2 would be translated into

$$(x^* \cdot y) \cdot z = x^* \cdot (y \cdot z),$$

which follows immediately from the associativity of '$\cdot$' (Axiom A5). SEI3 would be translated into

$$x^* \cdot (y \cdot (x + y)^* \cdot z + z) = (x + y)^* \cdot z,$$

which we prove below as equation EQ5.

The definition of strong-bisimulation becomes more complicated. This is due to the fact that for two bisimilar processes, a transition of one process might have to be simulated by many transitions in the other process. For example, according to Axiom C11 we have $x \lhd v_0 \rhd x = x \lhd v_1 \rhd x$ for all processes $x$ and (distinct) boolean programming variables $v_0$ and $v_1$. Consequently, the definition of strong-bisimulation should be adapted in such a way that these processes are bisimilar. To motivate the definition given below, which is based on strong-bisimulation for process algebras with propositional signals [BB97], we will focus more on the example. Suppose $x \xrightarrow{a,e_b} x'$. Then, using Rules 12 and 13, $x \lhd v_0 \rhd x$ has the following transitions: $x \lhd v_0 \rhd x \xrightarrow{a,e_b \wedge v_0} x'$ and $x \lhd v_0 \rhd x \xrightarrow{a,e_b \wedge \neg v_0} x'$. Similarly, $x \lhd v_1 \rhd x$ has the following transitions: $x \lhd v_1 \rhd x \xrightarrow{a,e_b \wedge v_1} x'$ and $x \lhd v_1 \rhd x \xrightarrow{a,e_b \wedge \neg v_1} x'$. The programming variables $v_0$ and $v_1$ are of type boolean, therefore it makes sense to assume their value is either *true* or *false*. Replacing the programming variables by their possible values, results in the following transitions.

$$\begin{array}{ll} x \lhd v_0 \rhd x \xrightarrow{a,e_b \wedge true} x' & x \lhd v_1 \rhd x \xrightarrow{a,e_b \wedge true} x' \\ x \lhd v_0 \rhd x \xrightarrow{a,e_b \wedge false} x' & x \lhd v_1 \rhd x \xrightarrow{a,e_b \wedge false} x' \end{array}$$

So, after replacing variables by their possible values, the processes are bisimilar. Formally, replacing programming variables by values is done by *valuations*. Valuations are functions from programming variables to terms without programming variables and extend straightforwardly to functions from boolean terms to boolean terms without programming variables. We denote valuations by $\sigma$, $\sigma'$, ... and assume they are defined for every programming variable.

Finally, the definition of strong-bisimulation is adapted as follows. A binary relation $R$ on processes is a *strong-bisimulation* if for all pairs $(p, q) \in R$ the following conditions hold:

1. $\forall e_b : p{\downarrow^{e_b}} \Rightarrow \forall \sigma : \exists e'_b : q{\downarrow^{e'_b}} \ \wedge \ \sigma(e_b) = \sigma(e'_b)$
2. $\forall e'_b : q{\downarrow^{e'_b}} \Rightarrow \forall \sigma : \exists e_b : p{\downarrow^{e_b}} \ \wedge \ \sigma(e_b) = \sigma(e'_b)$

**Signature**

| | |
|---|---|
| $\delta : \rightarrow P$ | |
| $\varepsilon : \rightarrow P$ | |
| $a : \rightarrow P$ | $a \in A$ |
| $\_ + \_ : \ P \times P \rightarrow P$ | |
| $\_ \cdot \_ : \ P \times P \rightarrow P$ | |
| $\lambda_s(\_) :\rightarrow P$ | $s \in S$ |
| $\_ \lhd e_b \rhd \_ : P \times P \rightarrow P$ | $e_b \in E_b$ |
| $\_^* : P \rightarrow P$ | |

**Axioms**

| | | | |
|---|---|---|---|
| A1 | $x + y = x + x$ | A6 | $x + \delta = x$ |
| A2 | $(x + y) + z = x + (y + z)$ | A7 | $\delta \cdot x = \delta$ |
| A3 | $x + x = x$ | A8 | $x \cdot \varepsilon = x$ |
| A4 | $(x + y) \cdot z = x \cdot z + y \cdot z$ | A9 | $\varepsilon \cdot x = x$ |
| A5 | $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | | |
| SO1 | $\lambda_s(a) = act(a, s)$ | | $a \in A \cup \{\delta\}$ |
| SO2 | $\lambda_s(\delta) = \delta$ | | |
| SO3 | $\lambda_s(\varepsilon) = \varepsilon$ | | |
| SO4 | $\lambda_s(a \cdot x) = act(a, s) \cdot \lambda_{\mathit{eff}(s,a)}(x)$ | | $a \in A \cup \{\delta\}$ |
| SO5 | $\lambda_s(x + y) = \lambda_s(x) + \lambda_s(y)$ | | |
| C1 | $x \lhd \mathit{true} \rhd y = x$ | | |
| C2 | $x \lhd \mathit{false} \rhd y = y$ | | |
| C3 | $x \lhd e_b \rhd y = (x \lhd e_b \rhd \delta) + (y \lhd \neg e_b \rhd \delta)$ | | |
| C4 | $x \lhd e_b \rhd y = y \lhd \neg e_b \rhd x$ | | |
| C5 | $(x \lhd e_b \rhd y) \lhd e_b \rhd z = x \lhd e_b \rhd z$ | | |
| C6 | $x \lhd e_b \rhd (y \lhd e_b \rhd z) = x \lhd e_b \rhd z$ | | |
| C7 | $(x \lhd e_b \rhd y) \cdot z = (x \cdot z) \lhd e_b \rhd (y \cdot z)$ | | |
| C8 | $(x + y) \lhd e_b \rhd z = (x \lhd e_b \rhd z) + (y \lhd e_b \rhd z)$ | | |
| C9 | $x \lhd e_b \ \wedge \ e'_b \rhd y = (x \lhd e_b \rhd y) \lhd e'_b \rhd y$ | | |
| C10 | $x \lhd e_b \ \vee \ e'_b \rhd \delta = (x \lhd e_b \rhd \delta) + (x \lhd e'_b \rhd \delta)$ | | |
| C11 | $x \lhd e_b \rhd x = x$ | | |
| C12 | $\lambda_s(x \lhd e_b \rhd y) = \lambda_s(x) \lhd s \circ e_b \rhd \lambda_s(y)$ | | |
| R1 | $x^* = \varepsilon + x \cdot (x^*)$ | | |
| R2 | $(x + y)^* = x^* \cdot (y \cdot x^*)^*$ | | |

Table 7: Axioms of $\text{BPA}_{\delta\varepsilon\lambda}$ with conditionals and iteration

3. $\forall a, e_b, p' : p \xrightarrow{a,e_b} p' \Rightarrow \forall \sigma : \exists e'_b, q' : q \xrightarrow{a,e'_b} q' \ \wedge \ (p', q') \in R \ \wedge \ \sigma(e_b) = \sigma(e'_b)$

4. $\forall a, e'_b, q' : q \xrightarrow{a,e'_b} q' \Rightarrow \forall \sigma : \exists e_b, p' : p \xrightarrow{a,e_b} p' \ \wedge \ (p', q') \in R \ \wedge \ \sigma(e_b) = \sigma(e'_b)$

The axioms of $\text{BPA}_{\delta\varepsilon\lambda}$ with conditionals and iteration are sound with respect to strong-bisimulation equivalence. Due to the result of Sewell mentioned above, we know a complete axiomatisation does not exist. Using the axioms of Table 8, we can prove many process

$$1\frac{}{\varepsilon\downarrow^{true}}\quad 2\frac{}{a \xrightarrow{a,true} \varepsilon}\quad 3\frac{x \xrightarrow{a,e_b} x'}{x+y \xrightarrow{a,e_b} x'}\quad 4\frac{y \xrightarrow{a,e_b} y'}{x+y \xrightarrow{a,e_b} y'}\quad 5\frac{x\downarrow^{e_b}}{(x+y)\downarrow^{e_b}}\quad 6\frac{y\downarrow^{e_b}}{(x+y)\downarrow^{e_b}}$$

$$7\frac{x \xrightarrow{a,e_b} x'}{x\cdot y \xrightarrow{a,e_b} x'\cdot y}\quad 8\frac{x\downarrow^{e_b}, y \xrightarrow{a,e'_b} y'}{x\cdot y \xrightarrow{a,(e_b\wedge e'b)} y'}\quad 9\frac{x\downarrow^{e_b}, y\downarrow^{e'_b}}{(x\cdot y)\downarrow^{e_b\wedge e'_b}}$$

$$10\frac{x \xrightarrow{a,e_b} x',\ act(a,s)\neq\delta,\ s\circ e_b \neq false}{\lambda_s(x) \xrightarrow{act(a,s),s\circ e_b} \lambda_{eff(a,s)}(x')}\quad 11\frac{x\downarrow^{s\circ e_b},\ s\circ e_b\neq false}{\lambda_s(x)\downarrow^{s\circ e_b}}$$

$$12\frac{x \xrightarrow{a,e_b} x'}{x\lhd e'_b\rhd y \xrightarrow{a,(e_b\wedge e'_b)} x'}\quad 13\frac{y \xrightarrow{a,e_b} y'}{x\lhd e'_b\rhd y \xrightarrow{a,(e_b\wedge \neg e'_b)} y'}\quad 14\frac{x\downarrow^{e_b}}{(x\lhd e'_b\rhd y)\downarrow^{(e_b\wedge e'_b)}}$$

$$15\frac{y\downarrow^{e_b}}{(x\lhd e'_b\rhd y)\downarrow^{(e_b\wedge \neg e'_b)}}\quad 16\frac{}{(x^*)\downarrow^{e_b}}\quad 17\frac{x \xrightarrow{a,e_b} x'}{x^* \xrightarrow{a,e_b} x'\cdot(x^*)}$$

Table 8: SOS rules of $\text{BPA}_{\delta\varepsilon\lambda}$ with conditionals and iteration

identities. We included process identities EQ1–EQ6 as an illustration.

| | | | |
|---|---|---|---|
| EQ1 | $x\lhd e_b \vee e'_b\rhd y = x\lhd e_b\rhd(x\lhd e'_b\rhd y)$ | EQ4 | $(x\lhd e_b\rhd\delta)^*\cdot z\lhd\neg e_b\rhd y = z\lhd\neg e_b\rhd y$ |
| EQ2 | $(x\lhd e_b\rhd y)\lhd\neg e_b\rhd z = y\lhd\neg e_b\rhd z$ | EQ5 | $x^*\cdot(y\cdot(x+y)^*\cdot z+z) = (x+y)^*\cdot z$ |
| EQ3 | $(x\lhd e_b\rhd\delta)\lhd\neg e_b\rhd\delta = \delta$ | EQ6 | $x^* = x^*\cdot x^*$ |

**Proof EQ1**

$x\lhd e_b \vee e'_b\rhd y$

$=$ {Boolean logic}

$x\lhd\neg(\neg e_b \wedge \neg e'_b)\rhd y$

$=$ {Axiom C4 (from right to left)}

$y\lhd\neg e_b \wedge \neg e'_b\rhd x$

$=$ {Boolean logic}

$y\lhd\neg e'_b \wedge \neg e_b\rhd x$

$=$ {Axiom C9}

$(y\lhd\neg e'_b\rhd x)\lhd\neg e_b\rhd x$

$=$ {Axiom C4 two times (from right to left)}

$x\lhd e_b\rhd(x\lhd e'_b\rhd y)$

**Proof EQ2**

$(x\lhd e_b\rhd y)\lhd\neg e_b\rhd z$

$=$ {Axiom C4}

$z\lhd e_b\rhd(x\lhd e_b\rhd y)$

$=$ {Axiom C6}

$z\lhd e_b\rhd y$

$=$ {Axiom C4}

$y\lhd\neg e_b\rhd z$

**Proof EQ3**

$(x \lhd e_b \rhd \delta) \lhd \neg e_b \rhd \delta$

$=$ {Equality EQ2}

$\delta \lhd \neg e_b \rhd \delta$

$=$ {Axiom C11}

$\delta$

**Proof EQ4**

$(x \lhd e_b \rhd \delta)^* \cdot z \lhd \neg e_b \rhd y$

$=$ {Axiom R1}

$(\varepsilon + (x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot z \lhd \neg e_b \rhd y$

$=$ {Axiom A4}

$(\varepsilon \cdot z + ((x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot z) \lhd \neg e_b \rhd y$

$=$ {Axiom C8}

$(\varepsilon \cdot z \lhd \neg e_b \rhd y) + (((x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot z \lhd \neg e_b \rhd y)$

$=$ {Axioms A5, C7, and A7}

$(\varepsilon \cdot z \lhd \neg e_b \rhd y) + ((x \cdot (x \lhd e_b \rhd \delta)^* \cdot z) \lhd e_b \rhd \delta) \lhd \neg e_b \rhd y$

$=$ {Equation EQ2}

$(\varepsilon \cdot z \lhd \neg e_b \rhd y) + (\delta \lhd \neg e_b \rhd y)$

$=$ {Axioms A9 and C8}

$(z + \delta) \lhd \neg e_b \rhd y)$

$=$ {Axioms A1 and A6}

$z \lhd \neg e_b \rhd y$

**Proof EQ5**

$x^* \cdot (y \cdot (x + y)^* \cdot z + z)$

$=$ {Axiom R2}

$x^* \cdot (y \cdot (x^* \cdot (y \cdot x^*)^*) \cdot z + z)$

$=$ {Axioms A5, A1, and A9}

$x^* \cdot (\varepsilon \cdot z + (y \cdot x^*) \cdot (y \cdot x^*)^* \cdot z)$

$=$ {Axiom A4}

$x^* \cdot ((\varepsilon + (y \cdot x^*) \cdot (y \cdot x^*)^*) \cdot z)$

$=$ {Axiom R1}

$x^* \cdot ((y \cdot x^*)^* \cdot z)$

$=$ {Axioms A5 and R2}

$(x + y)^* \cdot z$

**Proof EQ6**

$x^*$

$= \{\text{Axiom A3}\}$

$(x + x)^*$

$= \{\text{Axiom A8}\}$

$(x + x)^* \cdot \varepsilon$

$= \{\text{Equation EQ5}\}$

$x^* \cdot (x \cdot (x + x)^* \cdot \varepsilon + \varepsilon)$

$= \{\text{Axiom A8}\}$

$x^* \cdot (x \cdot (x + x)^* + \varepsilon)$

$= \{\text{Axiom A3}\}$

$x^* \cdot (x \cdot x^* + \varepsilon)$

$= \{\text{Axiom A1}\}$

$x^* \cdot (\varepsilon + x \cdot x^*)$

$= \{\text{Axiom R1}\}$

$x^* \cdot x^*$

# 6 Sequential Imperative Programs

In previous sections we indicated how certain programming language constructs can be formalised using $\text{BPA}_{\delta\varepsilon\lambda}$ with conditionals and iteration. In this section, we focus on this issue in more detail by formalizing a small programming language $L$. In the first column of Table 9, the statements of $L$ are given. Here, $e$ is an expression, $e_b$ is a boolean expression, $v$ is a program variable, and $S_1$ and $S_2$ are $L$-statements. In this paper, we do not care whether programming variables are defined; the properties of $L$ statements we derive below are independent of the value of programming variables. The semantics of $L$-statements in terms of $\text{BPA}_{\delta\varepsilon\lambda}$ is given by the function $\mathcal{M} : L \to \text{BPA}_{\delta\varepsilon\lambda}$ which is defined in the second column of Table 9.

| $L$-statement $S$ | $\text{BPA}_{\delta\varepsilon\lambda}$-process $\mathcal{M}(S)$ |
|---|---|
| $v := e$ | $v := e$ |
| if $e_b$ then $S_1$ fi | $\mathcal{M}(S_1) \lhd e_b \rhd \varepsilon$ |
| if $e_b$ then $S_1$ else $S_2$ fi | $\mathcal{M}(S_1) \lhd e_b \rhd \mathcal{M}(S_2)$ |
| $S_1; S_2$ | $\mathcal{M}(S_1) \cdot \mathcal{M}(S_2)$ |
| while $e_b$ do $S$ od | $(\mathcal{M}(S) \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$ |

Table 9: Syntax and semantics of $L$-statements

Using the translation of Table 9 and the axioms of $\text{BPA}_{\delta\varepsilon\lambda}$ with conditionals and iteration, it is possible to perform computations on programs. Table 10 shows some program equalities we can derive. We will now prove these equalities.

| | | | |
|---|---|---|---|
| EQ7 | $\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd \neg e_b \rhd y$ | $=$ | $\varepsilon \lhd \neg e_b \rhd y$ |
| EQ8 | $\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}$ | $=$ | $\mathsf{if}\ e_b\ \mathsf{then}\ x\ \mathsf{fi}\ ;\ \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}$ |
| EQ9 | $\lambda_s(\mathsf{if}\ e_b\ \mathsf{then}\ x\ \mathsf{fi})$ | $=$ | $\mathsf{if}\ s \circ e_b\ \mathsf{then}\ \lambda_s(x)\ \mathsf{fi}$ |
| EQ10 | $\lambda_s(\mathsf{if}\ e_b\ \mathsf{then}\ x\ \mathsf{else}\ y\ \mathsf{fi})$ | $=$ | $\mathsf{if}\ s \circ e_b\ \mathsf{then}\ \lambda_s(x)\ \mathsf{else}\ \lambda_s(y)\ \mathsf{fi}$ |
| EQ11 | $\lambda_s(\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od})$ | $=$ | $\lambda_s(x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}) \lhd s \circ e_b \rhd \varepsilon$ |
| EQ12 | $\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}$ | $=$ | $\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}\ ;\ \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}$ |
| EQ13 | $\mathsf{if}\ e_b\ \mathsf{then}\ x; z\ \mathsf{else}\ y; z\ \mathsf{fi}$ | $=$ | $\mathsf{if}\ e_b\ \mathsf{then}\ x\ \mathsf{else}\ y\ \mathsf{fi}\ ;\ z$ |

Table 10: Examples of program equalities

**Proof EQ7**

$\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd \neg e_b \rhd y$

$=$ {Translation of statements according to Table 9}

$(x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta) \lhd \neg e_b \rhd y$

$=$ {Axiom R1}

$((\varepsilon + (x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta) \lhd \neg e_b) \rhd y$

$=$ {Axiom A4}

$(\varepsilon \cdot (\varepsilon \lhd \neg e_b \rhd \delta) + ((x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)) \lhd \neg e_b \rhd y$

$=$ {Axiom C8}

$(\varepsilon \cdot (\varepsilon \lhd \neg e_b \rhd \delta) \lhd \neg e_b \rhd y) + (((x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)) \lhd \neg e_b \rhd y$

$=$ {Axioms A9 and C5}

$(\varepsilon \lhd \neg e_b \rhd y) + (((x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)) \lhd \neg e_b \rhd y$

$=$ {Axioms A5 and C3}

$(\varepsilon \lhd \neg e_b \rhd y) + ((((x \lhd e_b \rhd \delta) \cdot ((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta))) \lhd \neg e_b \rhd \delta) + (y \lhd \neg\neg e_b \rhd \delta))$

$=$ {Axiom A7}

$(\varepsilon \lhd \neg e_b \rhd y) +$
$((((x \lhd e_b \rhd \delta) \cdot ((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta))) \lhd \neg e_b \rhd \delta \cdot ((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta))) + (y \lhd \neg\neg e_b \rhd \delta))$

$=$ {Axiom C7}

$(\varepsilon \lhd \neg e_b \rhd y) + (((x \lhd e_b \rhd \delta) \lhd \neg e_b \rhd \delta) \cdot ((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta)) + (y \lhd \neg\neg e_b \rhd \delta))$

$=$ {Equation EQ3}

$(\varepsilon \lhd \neg e_b \rhd y) + (\delta \cdot ((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta)) + (y \lhd \neg\neg e_b \rhd \delta))$

$=$ {Axioms A7, A1, and A6}

$(\varepsilon \lhd \neg e_b \rhd y) + (y \lhd \neg\neg e_b \rhd \delta)$

$=$ {Axiom C3}

$(\varepsilon \lhd \neg e_b \rhd \delta) + (y \lhd \neg\neg e_b \rhd \delta) + (y \lhd \neg\neg e_b \rhd \delta)$

$=$ {Axioms A2 and A3}

$(\varepsilon \lhd \neg e_b \rhd \delta) + (y \lhd \neg\neg e_b \rhd \delta)$

$=$ {Axiom C3}

$\varepsilon \lhd \neg e_b \rhd y$

**Proof EQ8**

$\mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Translation of statements according to Table 9}

$(x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$

$=$ {Axiom R1}

$(\varepsilon + (x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$

$=$ {Axiom A4}

$\varepsilon \cdot (\varepsilon \lhd \neg e_b \rhd \delta) + ((x \lhd e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$

$=$ {Axioms A9 and A5}

$(\varepsilon \lhd \neg e_b \rhd \delta) + (x \lhd e_b \rhd \delta) \cdot ((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta))$

$=$ {Translation of while statements according to Table 9}

$(\varepsilon \lhd \neg e_b \rhd \delta) + (x \lhd e_b \rhd \delta) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Equality EQ7}

$(\mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od} \lhd \neg e_b \rhd \delta) + (x \lhd e_b \rhd \delta) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Axiom A7}

$(\mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od} \lhd \neg e_b \rhd \delta \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}) + (x \lhd e_b \rhd \delta) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Axiom A9}

$(\varepsilon \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od} \lhd \neg e_b \rhd \delta \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}) + (x \lhd e_b \rhd \delta) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Axiom C7}

$(\varepsilon \lhd \neg e_b \rhd \delta) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od} + (x \lhd e_b \rhd \delta) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Axioms A4 and A1}

$((x \lhd e_b \rhd \delta) + (\varepsilon \lhd \neg e_b \rhd \delta)) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Axiom C3}

$(x \lhd e_b \rhd \varepsilon) \cdot \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

$=$ {Translation of if-then and ';' statements according to Table 9}

$\mathsf{if}\, e_b\, \mathsf{then}\, x\, \mathsf{fi}\, ;\, \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

**Proof EQ9**

$\lambda_s(\mathsf{if}\, e_b\, \mathsf{then}\, x\, \mathsf{fi})$

$=$ {Translation of if-then statements according to Table 9}

$\lambda_s(x \lhd e_b \rhd \varepsilon)$

$=$ {Axiom C12}

$\lambda_s(x) \lhd s \circ e_b \rhd \lambda_s(\varepsilon)$

$=$ {Axiom SO3}

$\lambda_s(x) \lhd s \circ e_b \rhd \varepsilon$

$=$ {Translation of if-then statements according to Table 9}

$\mathsf{if}\, s \circ e_b\, \mathsf{then}\, \lambda_s(x)\, \mathsf{fi}$

**Proof EQ10**

$\lambda_s(\mathsf{if}\ e_b\ \mathsf{then}\ x\ \mathsf{else}\ y\ \mathsf{fi})$

$=$ {Translation of if-then-else statements according to Table 9}

$\lambda_s(x \lhd e_b \rhd y)$

$=$ {Axiom C12}

$\lambda_s(x) \lhd s \circ e_b \rhd \lambda_s(y)$

$=$ {Translation of if-then-else statements according to Table 9}

$\mathsf{if}\ s \circ e_b\ \mathsf{then}\ \lambda_s(x)\ \mathsf{else}\ \lambda_s(y)\ \mathsf{fi}$

**Proof EQ11**

$\lambda_s(\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od})$

$=$ {Equation EQ8}

$\lambda_s(\mathsf{if}\ e_b\ \mathsf{then}\ x\ \mathsf{fi}\ ;\ \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od})$

$=$ {Translation of statements according to Table 9}

$\lambda_s((x \lhd e_b \rhd \varepsilon) \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od})$

$=$ {Axiom C7}

$\lambda_s(x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd e_b \rhd \varepsilon \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od})$

$=$ {Axiom C3}

$\lambda_s((x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd e_b \rhd \delta) + (\varepsilon \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd \neg e_b \rhd \delta))$

$=$ {Axiom A9}

$\lambda_s((x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd e_b \rhd \delta) + (\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd \neg e_b \rhd \delta))$

$=$ {Equality EQ7}

$\lambda_s((x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd e_b \rhd \delta) + (\varepsilon \lhd \neg e_b \rhd \delta))$

$=$ {Axiom C3}

$\lambda_s(x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od} \lhd e_b \rhd \varepsilon)$

$=$ {Axiom C12}

$\lambda_s(x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}) \lhd s \circ e_b \rhd \lambda_s(\varepsilon)$

$=$ {Axiom SO3}

$\lambda_s(x \cdot \mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}) \lhd s \circ e_b \rhd \varepsilon$

**Proof EQ12**

$\mathsf{while}\ e_b\ \mathsf{do}\ x\ \mathsf{od}$

$=$ {Translation of statements according to Table 9}

$((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta))$

$=$ {Axiom C5}

$(x \lhd e_b \rhd \delta)^* \cdot ((\varepsilon \lhd \neg e_b \rhd \delta) \lhd \neg e_b \rhd \delta)$

$=$ {Axioms A9 and A7}

$(x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \cdot (\varepsilon \lhd \neg e_b \rhd \delta) \lhd \neg e_b \rhd \delta \cdot (\varepsilon \lhd \neg e_b \rhd \delta))$

$=$ {Axiom C7}

$(x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$

$=$ {Equation EQ4}

$(x \lhd e_b \rhd \delta)^* \cdot ((x \lhd e_b \rhd \delta)^* \cdot \varepsilon \lhd \neg e_b \rhd \delta) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$

$=$ {Axioms A8, A9, and A7}

$(x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \cdot (x \lhd e_b \rhd \delta)^* \lhd \neg e_b \rhd \delta \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$

$= \{\text{Axiom C7}\}$

$(x \lhd e_b \rhd \delta)^* \cdot ((\varepsilon \lhd \neg e_b \rhd \delta) \cdot (x \lhd e_b \rhd \delta)^*) \cdot (\varepsilon \lhd \neg e_b \rhd \delta)$

$= \{\text{Axiom A5}\}$

$((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta)) \cdot ((x \lhd e_b \rhd \delta)^* \cdot (\varepsilon \lhd \neg e_b \rhd \delta))$

$= \{\text{Translation of statements according to Table 9}\}$

$\mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}\, ;\, \mathsf{while}\, e_b\, \mathsf{do}\, x\, \mathsf{od}$

**Proof EQ13**

$\mathsf{if}\, e_b\, \mathsf{then}\, x; z\, \mathsf{else}\, y; z\, \mathsf{fi}$

$= \{\text{Translation of statements according to Table 9}\}$

$x \cdot z \lhd e_b \rhd y \cdot z$

$= \{\text{Axiom C7}\}$

$(x \lhd e_b \rhd y) \cdot z$

$= \{\text{Translation of statements according to Table 9}\}$

$\mathsf{if}\, e_b\, \mathsf{then}\, x\, \mathsf{else}\, y\, \mathsf{fi}\, ;\, z$

# 7 Conclusions

We have shown that ACP-style process algebras [BPS01, Fok00, BW90] can be used as a formal framework for algebraic reasoning about imperative sequential programs. In particular, we provided two instantiations of the parameters of the existing process algebra $\text{BPA}_{\delta\varepsilon\lambda}$ [BB88, BV95]. Both instantiations result in a program algebra for imperative sequential programs with assignment actions and programming variables. However, in the first (more straightforward) instantiation, program variables do not adhere to scoping rules known from existing programming languages like Pascal, C, and Java. In the second instantiation, this problem is solved by distinguishing normal assignment actions from update actions. An assignment action is renamed (by the state operator) into an update action if it has updated a programming variable. We use this program algebra to define the formal semantics of a simple programming language containing typical sequential programming constructs like if.. then.. else.. fi and while.. do.. od. Furthermore, we show that these constructs obey laws that can be proved using the axioms of ACP. The fact that we could use the process algebras off the shelf shows the power of these formalisms.

Future work can address several topics. First of all, it would be interesting to implement the program algebra in verification tools. These tools should help in formal reasoning about sequential programs. Another promising research area is concurrency: can we use the existing concurrency theory of ACP-style process algebras to reason about multi-threaded programs? We expect this to result in a powerful formal framework. However, it probably needs tool support even more than our program algebra for sequential programs, because proofs about concurrency generally contain huge amounts of detail. Yet another research direction could be to investigate if the existing ACP theory of abstraction and branching bisimulation can be used to check correctness of imperative sequential (or concurrent) programs. We have achieved some results in this area already, but more work is needed to draw conclusions. The final topic we mention is to investigate how well our framework can be adapted to capture object oriented programming. A possibility is to use a state operator for each object. For instance, process $\lambda_s^m(x)$ is an object $m$ with state $s$.

## References


[AFGI96] L. Aceto, W. Fokkink, R. van Glabbeek, and A. Ingólfsdóttir. Axiomatizing prefix iteration with silent step. *Information and Computation*, 127(1):26–40, 1996.

[AG95] L. Aceto and J.F. Groote. A complete equational axiomatization for mpa with string iteration. Technical Report RS-95-28, Department of Mathematics and Computer Science, Aalborg University, 1995.

[AI96] L. Aceto and A. Ingólfsdóttir. An equational axiomatization of obeservation congruence for prefix iteration. In M. Wirsing and M. Nivat, editors, *Proceedings AMAST'96*, number 1101 in Lecture Notes in Computer Science, pages 195–209, Munich, Germany, 1996. Springer-Verlag.

[BB88] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.

[BB90] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and mathematical method, Summer School*, number F 88 in NATO ASI Series, pages 273–323, Marktoberdorf, 1990.

[BB97] J.C.M Baeten and J.A. Bergstra. Process algebra with propositional signals. *Theoretical Computer Science*, 177(2):381–405, May 1997.

[BBP94] J. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *The Computer Journal*, 37:243–258, 1994. Originally appeared as report P9314, Programming Research Group, University of Amsterdam, 1993.

[BK02] V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. PhD thesis, Technische Universiteit Eindhoven, March 2002.

[BL00] J.A. Bergstra and M.E. Loots. Program algebra for component code. *Formal Aspects of Computing*, 12:1–17, 2000.

[BL02] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *The Journal of Logic and Algebraic Programming*, 51(2):125–156, June 2002.

[BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science B.V., Amsterdam, The Netherlands, first edition, 2001.

[BV95] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, Semantic Modelling, pages 149–268. Oxford University Press, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[BW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus, A Systematic Introduction*. Graduate texts in computer science. Springer, 1998.

[BW99] R.J.R. Back and J. von Wright. Reasoning algebraically about loops. *Acta Informatica*, 36:295–334, 1999.

[CE58] I.M. Copi and J.B. Elgot, C.C.and Wright. Realization of events by logical nets. *Journal of the ACM*, 5:181–196, 1958.

[CKS96] Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of kleene algebra with tests. Technical Report TR96-1598, Cornell University, Ithaca, N.Y., 19, 1996.

[Fok94] W. Fokkink. A complete equational axiomatization for prefix iteration. *Information Processing Letters*, 52(6):333–337, 1994.

[Fok96] Wan Fokkink. A complete axiomatization for prefix iteration in branching bisimulation. *Fundamenta Informaticae*, 26(2):103–113, 1996.

[Fok00] W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer-Verlag, Berlin, 2000.

[FZ94] Wan Fokkink and Hans Zantema. Basic process algebra with iteration: completeness of its equational axioms. *The Computer Journal*, 37(4):259–267, 1994.

[Gla97] Rob J. van Glabbeek. Axiomatizing flat iteration. In *International Conference on Concurrency Theory*, pages 228–242, 1997.

[HJ98] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[Kle56] S.C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[Koz94] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, May 1994.

[Koz98] Dexter Kozen. On hoare logic and kleene algebra with tests. In *Proceedings of the 14th Symposium on Logic in Computer Science*. Institute of Electrical and Electronics Engineers, 1998.

[Mes98] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, July 1983.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings 5th GI (Gesellschaft für Informatik) Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Karlsruhe, Germany, 1981.

[Sal66] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, 1966.

[Sew94] P. Sewell. Bisimulation is not finitely (first order) equationally axiomatisable. In *Proceedings 9 Annual Symposium on Logic in Computer Science*, pages 62–70, Paris, France, 1994. IEEE Computer Society Press.

**Turku Centre for Computer Science**
**Lemminkäisenkatu 14**
**FIN-20520 Turku**
**Finland**

http://www.tucs.fi

**University of Turku**
- **Department of Information Technology**
- **Department of Mathematics**

**Åbo Akademi University**
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

**Turku School of Economics and Business Administration**
- **Institute of Information Systems Science**