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
Lemminkäieniau 14 A
FIN-20520 Turku
Finland
vbos@abo.fi
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
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 \( \lambda^m_s(\cdot) \) 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 \texttt{if}..\texttt{then}..\texttt{else}..\texttt{fi} and \texttt{while}..\texttt{do}..\texttt{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 \( \lambda^m_s(\cdot) \)). 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 languages are described algebraically. The formalization is based on the refinement calculus [BW98]. An algebraic framework for imperative sequential code has been described in [BL02, 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 $BPA_{\delta;\lambda}$, which is the starting point of our investigation, will be described. Sect. 4 discusses how we can instantiate $BPA_{\delta;\lambda}$ in order to formalize assignments. Next, in Sect. 5, $BPA_{\delta;\lambda}$ 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 $K$. Consequently, all sorts and terms are of this kind $K$. Therefore, we need not, and will not, mention $K$ explicitly in the MEL specifications. Sorts are atomic entities and terms can be atomic or structured. The syntax of terms is defined by constants and operator definitions:

$$
c: Term \quad \text{Constant definition}
$$

$$
\oplus: Term \quad \text{Binary operator definition}
$$

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.

$$
\begin{align*}
\text{CMA} &::= MA \\
&| MA \ AL \\
\text{CEA} &::= EA \\
&| EA \ AL \\
\text{EA} &::= \text{term} = \text{term} \\
\text{MA} &::= \text{term} : \text{sort} \\
\text{A} &::= \text{MA} | \text{EA} \\
\text{AL} &::= \text{A} | \text{ALA}
\end{align*}
$$

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

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

1In 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.

$$\cdot \oplus \cdot : S \times S \rightarrow S$$

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 $S_2$.

$$x : S_2 \quad x : S_1$$

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 $\cdot \mapsto \cdot$ is defined that builds a state from a programming variable and an expression. Finally, an operator $\cdot \circ \cdot$ 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

<table>
<thead>
<tr>
<th>Sort</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V, E, S$</td>
<td>$v, v': V$</td>
</tr>
<tr>
<td>$E$</td>
<td>$e, e': E$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash: V \times E \rightarrow S$</td>
<td></td>
</tr>
<tr>
<td>$\circ: S \times E \rightarrow E$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Axioms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$V \subseteq E$</td>
</tr>
<tr>
<td>S2</td>
<td>$(v \leftrightarrow e) \circ v = e$</td>
</tr>
<tr>
<td>S3</td>
<td>$(v \leftrightarrow e) \circ v' = e'$</td>
</tr>
<tr>
<td>S4</td>
<td>$(v \leftrightarrow e) \circ (v \leftrightarrow e') \circ e'' = (v \leftrightarrow (v \leftrightarrow e) \circ e') \circ e''$</td>
</tr>
</tbody>
</table>

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 $V_B$ and $V_N$. 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 $V_B$ or $V_N$. 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 $vb_0$, $vb_0 \land vb_1$, $v_0$, and $v_0 - v_1$.

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

In this section, we present the existing process theory $\mathbf{BPA}_{\delta, \varepsilon, \lambda}$ [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 $act: (A \cup \{\delta\}) \times M \times S \rightarrow (A \cup \{\delta\})$, and a function $eff: (A \cup \{\delta\}) \times M \times S \rightarrow S$. In addition, it has a constant, $\delta$ denoting the deadlock process, a constant $\varepsilon$ denoting the empty process, a binary alternative composition operator, $+\cdot$, a binary sequential composition operator, $\cdot\cdot\cdot$, and a unary state operator, $\lambda^m_n(\cdot)$, where $m \in M$ and $s \in S$. The sequential composition operator binds stronger than the alternative composition operator. Intuitively, process $\lambda^m_s(x)$ is a machine $m$ in state $s$, representing its memory, that tries to execute $x$, representing its program. The algebraic specification of $\mathbf{BPA}_{\delta, \varepsilon, \lambda}$ is given in Table 5. The operational semantics of $\mathbf{BPA}_{\delta, \varepsilon, \lambda}$ is given in Table 6. It defines for all actions $a \in A$ a binary transition relation $\vdash \xrightarrow{a} \vdash$ on processes as well as a termination predicate $\vdash$ on processes. If
**Specification** Booleans

<table>
<thead>
<tr>
<th>Sort</th>
<th>Operators</th>
<th>Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathbb{B})</td>
<td>(\mathit{true} : \to \mathbb{B})</td>
<td>(\mathbf{B1}) (\neg \mathit{true} = \mathit{false})</td>
</tr>
<tr>
<td></td>
<td>(\mathit{false} : \to \mathbb{B})</td>
<td>(\mathbf{B2}) (\neg \mathit{false} = \mathit{true})</td>
</tr>
<tr>
<td></td>
<td>(\neg : \mathbb{B} \to \mathbb{B})</td>
<td>(\mathbf{B3}) (\neg b = b)</td>
</tr>
<tr>
<td></td>
<td>(\land : \mathbb{B} \times \mathbb{B} \to \mathbb{B})</td>
<td>(\mathbf{B4}) (\mathit{true} \land b = b)</td>
</tr>
<tr>
<td></td>
<td>(\lor : \mathbb{B} \times \mathbb{B} \to \mathbb{B})</td>
<td>(\mathbf{B5}) (\mathit{false} \land b = \mathit{false})</td>
</tr>
<tr>
<td></td>
<td>(\Rightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B})</td>
<td>(\mathbf{B6}) (b \land b' = b' \land b)</td>
</tr>
<tr>
<td>Variables</td>
<td>(b, b', b'' : \mathbb{B})</td>
<td>(\mathbf{B7}) (b \lor b' = \neg (b' \land \neg b'))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\mathbf{B8}) ((b \land b') \land b'' = b' \land (b' \land b''))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\mathbf{B9}) (b \Rightarrow b' = \neg b' \lor b)</td>
</tr>
</tbody>
</table>

**Specification** Natural numbers

<table>
<thead>
<tr>
<th>Sort</th>
<th>Operators</th>
<th>Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathbb{N})</td>
<td>(0 : \to \mathbb{N})</td>
<td>(\mathbf{N1}) (P \cdot 0 = 0)</td>
</tr>
<tr>
<td></td>
<td>(S : \mathbb{N} \to \mathbb{N})</td>
<td>(\mathbf{N2}) (P(Sn) = n)</td>
</tr>
<tr>
<td></td>
<td>(\mathit{P} : \mathbb{N} \to \mathbb{N})</td>
<td>(\mathbf{N3}) (0 + n = n)</td>
</tr>
<tr>
<td></td>
<td>(\mathit{P} : \mathbb{N} \to \mathbb{N})</td>
<td>(\mathbf{N4}) ((Sn) + n' = S(n + n'))</td>
</tr>
<tr>
<td></td>
<td>(\mathit{P} : \mathbb{N} \to \mathbb{N})</td>
<td>(\mathbf{N5}) (n + n' = n' + n)</td>
</tr>
<tr>
<td>Variables</td>
<td>(n, n' : \mathbb{N})</td>
<td>(\mathbf{N6}) (0 - n = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\mathbf{N7}) ((Sn) - 0 = Sn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\mathbf{N8}) ((Sn) - (Sn') = n - n')</td>
</tr>
</tbody>
</table>

**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 \(\mathit{act}\) 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. \(\forall a, p' : p \xrightarrow{a} p' \Rightarrow \exists q' : q \xrightarrow{a} q' \land (p', q') \in R\)
3. \(\forall a, q' : q \xrightarrow{a} q' \Rightarrow \exists p' : p \xrightarrow{a} p' \land (p', q') \in R\)

It is well known that, for closed terms, the axioms of \(\mathit{BPA}_{\mathit{de}\lambda}\) 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 \(\mathit{BPA}_{\mathit{de}}\) (i.e., \(\mathit{BPA}_{\mathit{de}\lambda}\) 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
Expressions Booleans Natural numbers

Sorts variables

\[ V, V \]

\[ B, b, b', b'' : B \]

\[ N, n, n' : N \]

Axioms

<table>
<thead>
<tr>
<th>BE1</th>
<th>[ V \subseteq V ]</th>
<th>NE1</th>
<th>[ V_N \subseteq V ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE2</td>
<td>[ V_B \subseteq B ]</td>
<td>NE2</td>
<td>[ V_N \subseteq N ]</td>
</tr>
<tr>
<td>BE3</td>
<td>[ B \subseteq E ]</td>
<td>NE3</td>
<td>[ N \subseteq E ]</td>
</tr>
<tr>
<td>BE4</td>
<td>[ s \circ \text{true} = \text{true} ]</td>
<td>NE4</td>
<td>[ s \circ 0 = 0 ]</td>
</tr>
<tr>
<td>BE5</td>
<td>[ s \circ \text{false} = \text{false} ]</td>
<td>NE5</td>
<td>[ s \circ (S \cdot n) = S(s \circ n) ]</td>
</tr>
<tr>
<td>BE6</td>
<td>[ s \circ (-b) = -(s \circ b) ]</td>
<td>NE6</td>
<td>[ s \circ (P \cdot n) = P(s \circ n) ]</td>
</tr>
<tr>
<td>BE7</td>
<td>[ s \circ (b \land b') = (s \circ b) \land (s \circ b') ]</td>
<td>NE7</td>
<td>[ s \circ (n + n') = (s \circ n) + (s \circ n') ]</td>
</tr>
<tr>
<td>BE8</td>
<td>[ s \circ (b \lor b') = (s \circ b) \lor (s \circ b') ]</td>
<td>NE8</td>
<td>[ s \circ (n - n') = (s \circ n) - (s \circ n') ]</td>
</tr>
<tr>
<td>BE9</td>
<td>[ s \circ (b \rightarrow b') = (s \circ b) \Rightarrow (s \circ b') ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Equational specification of boolean and natural number expressions

<table>
<thead>
<tr>
<th>Specification</th>
<th>Programming variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import</td>
<td>Boolean and natural number expressions</td>
</tr>
<tr>
<td>Operators</td>
<td>[ v_{b_0} : \rightarrow V_B ]</td>
</tr>
<tr>
<td></td>
<td>[ v_{b_1} : \rightarrow V_B ]</td>
</tr>
<tr>
<td></td>
<td>[ v_0 : \rightarrow V_N ]</td>
</tr>
<tr>
<td></td>
<td>[ v_1 : \rightarrow V_N ]</td>
</tr>
</tbody>
</table>

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 \( \text{BPA}_{\delta \in \lambda} \)

In this section, we instantiate the parameters of \( \text{BPA}_{\delta \in \lambda} \) 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 : \((A \cup \{\delta\}) \times M \times S \rightarrow (A \cup \{\delta\})\)

eff : \((A \cup \{\delta\}) \times M \times S \rightarrow S\)

Signature

\(\delta : \rightarrow P\)

\(\varepsilon : \rightarrow P\)

\(a : \rightarrow P\) \quad \(a \in A\)

\(- + : P \times P \rightarrow P\)

\(- \cdot - : P \times P \rightarrow P\)

\(\lambda^m_s(\_): \rightarrow P\) \quad \(m \in M, s \in S\)

Axioms

A1 \(x + y = x + x\)

A2 \((x + y) + z = x + (y + z)\)

A3 \(x + x = x\)

A4 \((x + y) \cdot z = x \cdot z + y \cdot z\)

A5 \((x \cdot y) \cdot z = x \cdot (y \cdot z)\)

SO1 \(\lambda^m_a(a) = \text{act}(a, m, s)\) \quad \(a \in A \cup \{\delta\}\)

SO2 \(\lambda^m_\delta(\delta) = \delta\)

SO3 \(\lambda^m_\varepsilon(\varepsilon) = \varepsilon\)

SO4 \(\lambda^m_s(a \cdot x) = \text{act}(a, m, s) \cdot \lambda^m_{\text{eff}(a, m, s)}(x)\) \quad \(a \in A \cup \{\delta\}\)

SO5 \(\lambda^m_s(x + y) = \lambda^m_s(x) + \lambda^m_s(y)\)

Table 5: The \(\text{BPA}_{\delta, \lambda}\) 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 \(e, e', \ldots\), typical programming variables are denoted by \(v, v', \ldots\), and typical states are denoted by \(s, s', \ldots\). As a first try, the set \(A\) of \(\text{BPA}_{\delta, \lambda}\) is instantiated by the set of assignment actions:

\[A \doteq \{v := e \mid v \in V, e \in E\}\]

here, \(\doteq\) denotes ‘equality per definition’. The intended meaning of an assignment action is
ignore the machine argument singleton set. Since there is only one machine, we will drop the nested state operators. Consider process transition and 3 \wedge \not= v 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 that it assigns the `value' which one assignment updates two variables, can be derived as follows.

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.

\[
\begin{align*}
act(\delta, s) &= \delta \\
act(v := e, v' \mapsto e') &= v := (v' \mapsto e') \circ e \\
eff(\delta, s) &= s \\
eff(v := e, v' \mapsto e') &= v \mapsto (v' \mapsto e') \circ e \\
eff(v := e, v' \mapsto e') &= v' \mapsto e' \text{ if } v \not=} v'
\end{align*}
\]

\( \mathbf{BPA}_{\delta, \lambda} \) 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 \( \equiv \) S 0, 2 \( \equiv \) S 1, and 3 \( \equiv \) S 2

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

\[
\text{true} \Rightarrow \{ \text{Rule 2} \}
\]

\[
v_0 := v_0 + 1 \xrightarrow{v_0:=v_0+1} \varepsilon
\]

\[
\text{true} \Rightarrow \{ \text{Rule 7} \}
\]

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

\[
\text{true} \Rightarrow \{ \text{Rule 10 and definition of act gives: } act(v_0 := v_0 + 1, v_0 \mapsto 1) = v_0 := 2 \}
\]

\[
\lambda_{v_0-1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0:=2} \lambda_{v_0-2}(\varepsilon \cdot v_0 := v_0 + 3)
\]

\[
\text{true} \Rightarrow \{ \text{Definitions of eff} \}
\]

\[
\lambda_{v_0-1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0:=2} \lambda_{v_0-2}(\varepsilon \cdot v_0 := v_0 + 3)
\]

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

\[
\text{true} \Rightarrow \{ \text{previous derivation} \}
\]

Table 6: SOS rules of BPA_{\delta, \lambda}
Obviously, this is not the behaviour we want. The outermost
innermost
has updated a variable, it is renamed into an update action. Update actions are denoted by
assignment actions that have been effected already. That is, after an assignment action
the outermost occurrence of \( v \) called \( \text{update} \)
The act
Using these definitions of \( \text{act} \)
Later, we use this equality to prove some properties about processes.
Proof AC1
\[ \text{act}(\text{act}(a, v \mapsto e), v \mapsto e') = \text{act}(a, v \mapsto (v \mapsto e') \circ e). \]
Later, we use this equality to prove some properties about processes.

**Proof AC1** We have to prove \( \text{act}(\text{act}(a, v \mapsto e), v \mapsto e') = \text{act}(a, v \mapsto (v \mapsto e') \circ e) \), 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.
\[ \text{act}(\text{act}(\delta, v \mapsto e), v \mapsto e') \]
\[ = \{ \text{Definition of \text{act}} \} \]
\[ \text{act}(\delta, v \mapsto e') \]
\[ = \{ \text{Definition of \text{act}} \} \]
\[ \delta \]
\[ = \{ \text{Definition of \text{act}} \} \]
\[ \text{act}(\delta, v \mapsto (v \mapsto e') \circ e) \]
\( a \equiv v' := e'' \). There are two sub-cases: \( v \equiv v' \) and \( v \not\equiv v' \).
\( v \equiv v' \). The proof is as follows.

\[
\text{act}(\text{act}(v := e'', v \mapsto e) , v \mapsto e')
\]

\[
= \{ \text{Definition of act} \}
\]

\[
\text{act}(v \mapsto (v \mapsto e) \circ e'' , v \mapsto e')
\]

\[
= \{ \text{Definition of act} \}
\]

\[
v \mapsto (v \mapsto e') \circ (v \mapsto e) \circ e''
\]

\[
= \{ \text{Axiom S4} \}
\]

\[
v \mapsto (v \mapsto (v \mapsto e') \circ e) \circ e''
\]

\[
= \{ \text{Definition of act} \}
\]

\[
\text{act}(v := e'', v \mapsto (v \mapsto e') \circ e)
\]

\( v \not\equiv v' \). The proof is as follows.

\[
\text{act}(\text{act}(v' := e'', v \mapsto e) , v \mapsto e')
\]

\[
= \{ \text{Definition of act} \}
\]

\[
\text{act}(v' := (v \mapsto e) \circ e'' , v \mapsto e')
\]

\[
= \{ \text{Definition of act} \}
\]

\[
v' := (v \mapsto e') \circ (v \mapsto e) \circ e''
\]

\[
= \{ \text{Axiom S4} \}
\]

\[
v' := (v \mapsto (v \mapsto e') \circ e) \circ e''
\]

\[
= \{ \text{Definition of act} \}
\]

\[
\text{act}(v' := e'', v \mapsto (v \mapsto e') \circ e)
\]

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

\[
\text{act}(\text{act}(v' := e'' , v \mapsto e) , v \mapsto e')
\]

\[
= \{ \text{Definition of act} \}
\]

\[
\text{act}(v' \leftarrow (v \mapsto e) \circ e'' , v \mapsto e')
\]

\[
= \{ \text{Definition of act} \}
\]

\[
v' \leftarrow (v \mapsto e') \circ (v \mapsto e) \circ e''
\]

\[
= \{ \text{Axiom S4} \}
\]

\[
v' \leftarrow (v \mapsto (v \mapsto e') \circ e) \circ e''
\]

\[
= \{ \text{Definition of act} \}
\]

\[
\text{act}(v' \leftarrow e'' , v \mapsto (v \mapsto e') \circ e)
\]

Consider again process \( \lambda_{v_0 := 0}(\lambda_{v_0 := 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \). Now, the previous transition, \( \lambda_{v_0 := 0}(\lambda_{v_0 := 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \overset{v_0 := 2}{\longrightarrow} \lambda_{v_0 := 2}(\lambda_{v_0 := 2}(v_0 := v_0 + 3)) \), cannot be derived anymore. Instead, the transition \( \lambda_{v_0 := 0}(\lambda_{v_0 := 1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \overset{v_0 := 2}{\longrightarrow} \lambda_{v_0 := 0}(\lambda_{v_0 := 2}(v_0 := v_0 + 3)) \), 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 \( \text{act} \) and \( \text{eff} \), update actions have no effect on the state \( s \) in \( \lambda_s(x) \) processes.

true

\[
\Rightarrow \{ \text{Rule 2} \}
\]
\[ v_0 := v_0 + 1 \xrightarrow{v_0 \cdot v_1 + 1} \varepsilon \]

\[ \Rightarrow \{ \text{Rule 7} \} \]
\[ v_0 := v_0 + 1 \cdot v_0 := v_0 + 3 \xrightarrow{v_0 \cdot v_1 + 1} \varepsilon \cdot v_0 := v_0 + 3 \]

\[ \Rightarrow \{ \text{Rule 10 and new definition of } act \text{ gives: } act(v_0 := v_0 + 1, v_0 \mapsto 1) = v_0 \mapsto 2 \} \]
\[ \lambda_{v_0-1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0 \cdot v_1 + 2} \lambda_{eff(v_0 := v_0 + 1, v_0 \mapsto 1)}(\varepsilon \cdot v_0 := v_0 + 3) \]

\[ = \{ \text{New definitions of } eff \} \]
\[ \lambda_{v_0-1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3) \xrightarrow{v_0 \cdot v_1 - 2} \lambda_{v_0-2}(\varepsilon \cdot v_0 := v_0 + 3) \]

\[ \Rightarrow \{ \text{Rule 10 and new definition of } act \text{ gives: } act(v_0 \mapsto 2, v_0 \mapsto 0) = v_0 \mapsto 2 \} \]
\[ \lambda_{v_0-0}(\lambda_{v_0-1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0 \cdot v_1 - 2} \lambda_{eff(v_0 := v_0 + 2, v_0 \mapsto 0)}(\lambda_{v_0-2}(\varepsilon \cdot v_0 := v_0 + 3)) \]

\[ = \{ \text{New definitions of } eff \} \]
\[ \lambda_{v_0-0}(\lambda_{v_0-1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \xrightarrow{v_0 \cdot v_1 - 2} \lambda_{v_0-0}(\lambda_{v_0-2}(\varepsilon \cdot v_0 := v_0 + 3)) \]

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

\[ \lambda_{v_0-0}(\lambda_{v_0-1}(v_0 := v_0 + 1 \cdot v_0 := v_0 + 3)) \]

\[ = \{ \text{Axiom SO4} \} \]
\[ \lambda_{v_0-0}(act(v_0 := v_0 + 1, v_0 \mapsto 1) \cdot \lambda_{eff(v_0 := v_0 + 1, v_0 \mapsto 1)}(v_0 := v_0 + 3)) \]

\[ = \{ \text{Definitions of } act \text{ and } eff \} \]
\[ \lambda_{v_0-0}(v_0 \mapsto 2 \cdot \lambda_{v_0-2}(v_0 := v_0 + 3)) \]

\[ = \{ \text{Axiom SO4} \} \]
\[ act(v_0 \mapsto 2, v_0 \mapsto 0) \cdot \lambda_{eff(v_0 := v_0 + 2, v_0 \mapsto 0)}(\lambda_{v_0-2}(v_0 := v_0 + 3)) \]

\[ = \{ \text{Definitions of } act \text{ and } eff \} \]
\[ \lambda_{v_0-0}(\lambda_{v_0-2}(v_0 := v_0 + 3)) \]

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 \( \lambda_{v_0 \mapsto v}(\lambda_{v_0 \mapsto v'}(x)) = \lambda_{v_0 \mapsto v'}(x) \). However, since the value \( v' \) is an expression, it might contain variables and \( v_0 \) could be one of them. If \( v_0 \) occurs in \( v' \), its value is determined by the state of the enclosing state operator, i.e., \( \lambda_{v_0 \mapsto v}(\ldots) \). This explains the application \( (v_0 \mapsto v) \) in the right hand side of SO4d. Identities SO4a–SO4c can be proved straightforwardly using the axioms of BPA\( _{d,e} \) and the definitions of \( act \) and \( eff \). Identity SO4d is proved by structural induction, which means it is proved only for closed terms.

\[ \text{SO4a } \lambda_{v_0 \mapsto v}(v' := e' \cdot x) = v' := ((v \mapsto e) \circ e') \cdot \lambda_{v_0 \mapsto e}(x) \quad \text{if } v \neq v' \]

\[ \text{SO4b } \lambda_{v_0 \mapsto v}(v' := e' \cdot x) = v \leftarrow ((v \mapsto e) \circ e') \cdot \lambda_{v_0 \mapsto (v \mapsto e \circ e')}(x) \]

\[ \text{SO4c } \lambda_{v_0 \mapsto v}(v' \mapsto e' \cdot x) = v' \leftarrow (v \mapsto e) e' \cdot \lambda_{v_0 \mapsto e}(x) \]

\[ \text{SO4d } \lambda_{v_0 \mapsto v}(\lambda_{v_0 \mapsto v'}(x)) = \lambda_{v_0 \mapsto (v \mapsto e \circ e')}(x) \]
Proof SO4a
\[ \lambda_{v \rightarrow e'}(v' := e' \cdot x) \]
= \{Axiom SO4\}
\[ \text{act}(v' := e', v \mapsto e) \cdot \lambda_{\text{eff}(v' := e', v \rightarrow e)}(x) \]
= \{Definition of act and eff and } \]
\[ v' := ((v \mapsto e) \circ e') \cdot \lambda_{v \rightarrow e}(x) \]

Proof SO4b
\[ \lambda_{v \rightarrow e}(v := e' \cdot x) \]
= \{Axiom SO4\}
\[ \text{act}(v := e', v \mapsto e) \cdot \lambda_{\text{eff}(v := e', v \rightarrow e)}(x) \]
= \{Definition of act and eff\}
\[ v := ((v \mapsto e) \circ e') \cdot \lambda_{v \rightarrow (e \circ e')}(x) \]

Proof SO4c
\[ \lambda_{v \rightarrow e'}(v' := e' \cdot x) \]
= \{Axiom SO4\}
\[ \text{act}(v' := e', v \mapsto e) \cdot \lambda_{\text{eff}(v' := e', v \rightarrow e)}(x) \]
= \{Definition of act and eff\}
\[ v' := ((v \mapsto e) \circ e') \cdot \lambda_{v \rightarrow e}(x) \]

Proof SO4d We have to prove \[ \lambda_{v \rightarrow e}(\lambda_{v \rightarrow e'}(x)) = \lambda_{v \rightarrow (e \circ e')}(x) \]. We do this by structural induction on \( x \), using the fact that every BPA\( \delta; \lambda \) term can is equal to a BPA\( \delta; \lambda \)-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.
\[ \lambda_{v \rightarrow e}(\lambda_{v \rightarrow e'}(\varepsilon)) \]
= \{Axiom SO3\}
\[ \lambda_{v \rightarrow e}(\varepsilon) \]
= \{Axiom SO3\}
\[ \varepsilon \]
= \{Axiom SO3\}
\[ \lambda_{v \rightarrow (e \circ e')}(\varepsilon) \]

\( x \equiv \delta \). The proof is as follows.
\[ \lambda_{v \rightarrow e}(\lambda_{v \rightarrow e'}(\delta)) \]
= \{Axiom SO2\}
\[ \lambda_{v \rightarrow e}(\delta) \]
= \{Axiom SO2\}
\[ \delta \]
= \{Axiom SO2\}
\[ \lambda_{v \rightarrow (e \circ e')}(\delta) \]
$x \equiv a$, for some action $a$. The proof is as follows.

\[
\lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(a)) = \{\text{Axiom SO1}\} \\
\lambda_{v \mapsto e}(act(a, v \mapsto e')) = \{\text{Axiom SO1}\} \\
act(\lambda_{v \mapsto e}(act(a, v \mapsto e'), v \mapsto e)) = \{\text{Equation AC1}\} \\
act(a, v \mapsto (v \mapsto e) \circ e') = \{\text{Axiom SO1}\} \\
\lambda_{v \mapsto (v \mapsto e) \circ e'}(a)
\]

$x \equiv a \cdot s$, for some action $a$ and $\text{BPA}_{\delta, \lambda}$-basic term $s$. The proof is by case distinction on $a$.

Consequently, there are three sub-cases: $a \equiv \delta$; $a \equiv v' \equiv e''$; for some $v'$ and $e''$; and $a \equiv v' \equiv e''$ for some $v'$ and $e''$.

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

\[
\lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(\delta \cdot s)) = \{\text{Axiom A7}\} \\
\lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(\delta)) = \{\text{Axiom SO2 two times}\} \\
\delta = \{\text{Axiom SO2}\} \\
\lambda_{v \mapsto (v \mapsto e) \circ e'}(\delta) = \{\text{Axiom A7}\} \\
\lambda_{v \mapsto (v \mapsto e) \circ e'}(\delta \cdot s)
\]

$a \equiv v' \equiv e''$. We distinguish two cases: $v = v'$ and $v \neq v'$. The proof of the first case is as follows.

\[
\lambda_{v \mapsto e}(\lambda_{v \mapsto e'}(v := e'' \cdot s)) = \{\text{Axiom SO4}\} \\
\lambda_{v \mapsto e}(act(v := e'', v \mapsto e') \cdot \lambda_{\text{eff}}(v := e'', v := e'))(s)) = \{\text{Axiom SO4}\} \\
act(\lambda_{v \mapsto e}(act(v := e'', v \mapsto e'), v \mapsto e)) = \{\text{Equation AC1}\} \\
act(v := e'', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{\text{eff}}(act(v := e'', v := e'), v := e)(\lambda_{\text{eff}}(v := e'', v := e'))(s)) = \{\text{Definitions of act and eff}\} \\
act(v := e'', v \mapsto (v := e) \circ e') \cdot \lambda_{\text{eff}}(act(v := e'', v := e'), v := e)(\lambda_{\text{eff}}(v := e'', v := e'))(s)) = \{\text{Definition of eff}\} \\
act(v := e'', v \mapsto (v := e) \circ e') \cdot \lambda_{\text{eff}}(v := e'', v := e')(\lambda_{v \mapsto e}(v := e'))(s)) = \{\text{Induction hypothesis}\} \\
act(v := e'', v \mapsto (v := e) \circ e') \cdot \lambda_{\text{eff}}(v := e'', v := e')(\lambda_{v \mapsto (v \mapsto e) \circ e'})(s) = \{\text{Axiom S4}\}
\]
\[ \text{act}(v := e''', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{v \mapsto (v \mapsto (v \mapsto e) \circ e')} (s) \]

= \{\text{Definition of eff}\}

\[ \text{act}(v := e''', v \mapsto (v \mapsto e) \circ e') \cdot \lambda_{e' \mapsto (v := e''', v \mapsto (v \mapsto e) \circ e')} (s) \]

= \{ \text{Axiom SO4}\}

\[ \lambda_{v \mapsto (v \mapsto (v \mapsto e) \circ e')} (v := e'' \cdot s) \]

The proof of the second case, \( v \neq v' \) is as follows.

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) \]

= \{ \text{Axiom SO4}\}

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) \cdot \lambda_{e' \mapsto (v := e'' \cdot s)} (s) \]

= \{ \text{Axiom SO4}\}

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ \{ \text{Equality AC1}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ \{ \text{Definition of eff and v \neq v'}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ \{ \text{Definition of eff and v \neq v'}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ \{ \text{Axiom SO4}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ a \equiv v' \leftarrow e'' \cdot s \]

The proof is as follows.

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) \]

= \{ \text{Axiom SO4}\}

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) \cdot \lambda_{e' \mapsto (v := e'' \cdot s)} (s) \]

= \{ \text{Axiom SO4}\}

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) ; \lambda_{e' \mapsto (v := e'' \cdot s)} (s) \]

\[ \{ \text{Equality AC1}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) ; \lambda_{e' \mapsto (v := e'' \cdot s)} (s) \]

\[ \{ \text{Definition of eff and v \neq v'}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ \{ \text{Definition of eff}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ \{ \text{Definition of eff}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]

\[ \{ \text{Axiom SO4}\} \]

\[ \lambda_{v \mapsto e} (\lambda_{v \mapsto e'} (v' := e'' \cdot s)) (s) \]
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 $\text{BPA}_{\delta, \lambda}$-basic terms $s$ and $t$. The proof is as follows.

$$
\lambda_{v \rightarrow e}(\lambda_{v \rightarrow e}(s + t))
= \{ \text{Axiom SO5 two times} \}
\lambda_{v \rightarrow e}(\lambda_{v \rightarrow e}(s)) + \lambda_{v \rightarrow e}(\lambda_{v \rightarrow e}(t))
= \{ \text{Induction hypothesis two times} \}
\lambda_{v \rightarrow (v \rightarrow e) \circ e}(s) + \lambda_{v \rightarrow (v \rightarrow e) \circ e}(t)
= \{ \text{Axiom SO5} \}
\lambda_{v \rightarrow (v \rightarrow e) \circ e}(s + t)
$$

5 Conditionals and Iteration

In this section we add conditionals and iteration to $\text{BPA}_{\delta, \lambda}$. This extension enables us to formalize programming language constructs like if...then...else...fi and while...do...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 $x \{ e_b \rightarrow B \} y$, where $x$ and $y$ are processes and $e_b$ a boolean expression. If $e_b = \text{true}$, this conditional process behaves like $x$ and if $e_b = \text{false}$, it behaves like $y$.

The iteration operator we add to $\text{BPA}_{\delta, \lambda}$ 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: $x^*y = x^* \cdot y$. 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]:

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

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 $\text{BPA}_{\delta, \lambda}$ 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 $A$, $S$, $act$, and $\text{eff}$ 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 
processes is a strong-bisimulation for process algebras [BB97], we will focus more 
forward to functions from boolean terms to boolean terms without programming variables. We 
replace the programming variables by their possible values, results in 

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

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 
\( x < v_0 \triangleright x = x < v_1 \triangleright x \) 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, \( x < v_0 \triangleright x \) has the following 
transitions: \( x < v_0 \triangleright x \xrightarrow{a,e_b \lor v_0} x' \) and \( x < v_0 \triangleright x \xrightarrow{a,e_b \land \neg v_0} x' \). Similarly, \( x < v_1 \triangleright x \) has the 
following transitions: \( x < v_1 \triangleright x \xrightarrow{a,e_b \lor v_1} x' \) and \( x < v_1 \triangleright x \xrightarrow{a,e_b \land \neg v_1} x' \). 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.

\[
\begin{align*}
  x < v_0 \triangleright x & \xrightarrow{a,e_b \land true} x' \\
  x < v_0 \triangleright x & \xrightarrow{a,e_b \land false} x' \\
  x < v_1 \triangleright x & \xrightarrow{a,e_b \land true} x' \\
  x < v_1 \triangleright x & \xrightarrow{a,e_b \land false} x'
\end{align*}
\]

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. \( \forall e_b : p \downarrow e_b \Rightarrow \forall \sigma : \exists e_b' : q \downarrow e_b' \land \sigma(e_b) = \sigma(e_b') \)

2. \( \forall e_b' : q \downarrow e_b' \Rightarrow \forall \sigma : \exists e_b : p \downarrow e_b \land \sigma(e_b) = \sigma(e_b') \)
axiomatisation does not exist. Using the axioms of Table 8, we can prove many process
bisimulation equivalence. Due to the result of Sewell mentioned above, we know a complete

3.

bpa

The axioms of bpa with conditionals and iteration are sound with respect to strong-

The axioms of Table 8, we can prove many process
identities. We included process identities EQ1–EQ6 as an illustration.

\[ \begin{align*}
1 & \quad \varepsilon_1^{\text{true}} \quad \frac{\operatorname{a} \cdot \varepsilon_1}{\varepsilon_1} \\
2 & \quad \frac{\operatorname{a} \cdot \varepsilon_2}{z} \\
3 & \quad \frac{x \cdot \operatorname{a} \cdot \varepsilon_3}{x'}, \quad \frac{y \cdot \operatorname{a} \cdot \varepsilon_6}{y'} \\
4 & \quad \frac{x \cdot y \cdot \operatorname{a} \cdot \varepsilon_4}{x+y} \quad \frac{x \cdot y \cdot \operatorname{a} \cdot \varepsilon_5}{x+y} \\
5 & \quad \frac{x \cdot \varepsilon_7}{(x+y)} \quad \frac{y \cdot \varepsilon_8}{(x+y)} \\
6 & \quad \frac{x \cdot y \cdot \varepsilon_9}{x+y} \\
7 & \quad \frac{x \cdot \operatorname{a} \cdot \varepsilon_{10}}{x'} \\
8 & \quad \frac{x \cdot \varepsilon_{11}}{x'} \\
9 & \quad \frac{x \cdot \varepsilon_{12}}{x'} \\
10 & \quad \frac{x \cdot \operatorname{a} \cdot \varepsilon_{13}}{x'} \\
11 & \quad \frac{x \cdot \varepsilon_{14}}{x'} \\
12 & \quad \frac{x \cdot \varepsilon_{15}}{x'} \\
13 & \quad \frac{x \cdot \varepsilon_{16}}{x'} \\
14 & \quad \frac{x \cdot \varepsilon_{17}}{x'} \\
15 & \quad \frac{x \cdot \varepsilon_{18}}{x'} \\
16 & \quad \frac{x \cdot \varepsilon_{19}}{x'} \\
17 & \quad \frac{x \cdot \varepsilon_{20}}{x'} \\
\end{align*} \]

Table 8: SOS rules of BPA_{\delta\lambda} with conditionals and iteration

Proof EQ1
\[ x \not< e_b \lor e'_b \triangleright y \quad \frac{\{\text{Boolean logic}\}}{x \not< \neg(e_b \land e'_b) \triangleright y} \]
\[ \frac{\{\text{Axiom C4 (from right to left)}\}}{y \not< e_b \land \neg e'_b \triangleright x} \]
\[ \frac{\{\text{Boolean logic}\}}{y \not< e'_b \land \neg e_b \triangleright x} \]
\[ \frac{\{\text{Axiom C9}\}}{(y \not< e'_b \triangleright x) \not< \neg e_b \triangleright x} \]
\[ \frac{\{\text{Axiom C4 two times (from right to left)}\}}{x \not< e_b \triangleright (x \not< e'_b \triangleright y)} \]

Proof EQ2
\[ (x \not< e_b \triangleright y) \not< \neg e_b \triangleright z \]
\[ \frac{\{\text{Axiom C4}\}}{z \not< e_b \triangleright (x \not< e_b \triangleright y)} \]
\[ \frac{\{\text{Axiom C6}\}}{z \not< e_b \triangleright y} \]
\[ \frac{\{\text{Axiom C4}\}}{y \not< \neg e_b \triangleright z} \]

EQ1 \quad x \not< e_b \lor e'_b \triangleright y = x \not< e_b \triangleright (x \not< e'_b \triangleright y)

EQ2 \quad (x \not< e_b \triangleright y) \not< \neg e_b \triangleright z = y \not< \neg e_b \triangleright z

EQ3 \quad (x \not< e_b \triangleright \delta) \not< \neg e_b \triangleright \delta = \delta

EQ4 \quad (x \not< e_b \triangleright \delta)^* \cdot z \not< \neg e_b \triangleright y = z \not< \neg e_b \triangleright y

EQ5 \quad x^* \cdot (y \cdot (x + y))^* \cdot z + z = (x + y)^* \cdot z

EQ6 \quad x^* = x^* \cdot x^*
Proof EQ3
\[(x \triangleleft e_b \triangleright \delta) \triangleleft \neg e_b \triangleright \delta\]
= \{\text{Equality EQ2}\}
\[\delta \triangleleft \neg e_b \triangleright \delta\]
= \{\text{Axiom C11}\}
\[\delta\]

Proof EQ4
\[(x \triangleleft e_b \triangleright \delta)^* \cdot z \triangleleft \neg e_b \triangleright y\]
= \{\text{Axiom R1}\}
\[(\varepsilon + (x \triangleleft e_b \triangleright \delta) \cdot (x \triangleleft e_b \triangleright \delta)^*) \cdot z \triangleleft \neg e_b \triangleright y\]
= \{\text{Axiom A4}\}
\[(\varepsilon \cdot z + ((x \triangleleft e_b \triangleright \delta) \cdot (x \triangleleft e_b \triangleright \delta)^*) \cdot z) \triangleleft \neg e_b \triangleright y\]
= \{\text{Axiom C8}\}
\[(\varepsilon \cdot z \triangleleft \neg e_b \triangleright y) + ((x \cdot (x \triangleleft e_b \triangleright \delta)^* \cdot z) \triangleleft e_b \triangleright \delta) \triangleleft \neg e_b \triangleright y\]
= \{\text{Equation EQ2}\}
\[(\varepsilon \cdot z \triangleleft \neg e_b \triangleright y) + (\delta \triangleleft \neg e_b \triangleright y)\]
= \{\text{Axioms A9 and C8}\}
\[(z + \delta) \triangleleft \neg e_b \triangleright y\]
= \{\text{Axioms A1 and A6}\}
\[z \triangleleft \neg e_b \triangleright y\]

Proof EQ5
\[x^* \cdot (y \cdot (x + y)^*)^* \cdot z + z\]
= \{\text{Axiom R2}\}
\[x^* \cdot (y \cdot (x^* \cdot (y \cdot x^*)^*)^* \cdot z + z\]
= \{\text{Axioms A5, A1, and A9}\}
\[x^* \cdot (\varepsilon \cdot z + (y \cdot x^*) \cdot (y \cdot x^*)^* \cdot z)\]
= \{\text{Axiom A4}\}
\[x^* \cdot ((\varepsilon + (y \cdot x^*) \cdot (y \cdot x^*)^*) \cdot z)\]
= \{\text{Axiom R1}\}
\[x^* \cdot ((y \cdot x^*)^* \cdot z)\]
= \{\text{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 BPA_{δλ} 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 BPA_{δλ} is given by the function \( \mathcal{M} : L \rightarrow \text{BPA}_{δλ} \) which is defined in the second column of Table 9.

<table>
<thead>
<tr>
<th>( L )-statement ( S )</th>
<th>BPA_{δλ}-process ( \mathcal{M}(S) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v := e )</td>
<td>( v := e )</td>
</tr>
<tr>
<td>if ( e_b ) then ( S_1 ) fi</td>
<td>( \mathcal{M}(S_1) \downarrow e_b \uparrow \varepsilon )</td>
</tr>
<tr>
<td>if ( e_b ) then ( S_1 ) else ( S_2 ) fi</td>
<td>( \mathcal{M}(S_1) \downarrow e_b \uparrow \mathcal{M}(S_2) )</td>
</tr>
<tr>
<td>( S_1 ); ( S_2 )</td>
<td>( \mathcal{M}(S_1) \cdot \mathcal{M}(S_2) )</td>
</tr>
<tr>
<td>while ( e_b ) do S od</td>
<td>( (\mathcal{M}(S) \downarrow e_b \uparrow \delta)^* \cdot (\varepsilon \downarrow \neg e_b \uparrow \delta) )</td>
</tr>
</tbody>
</table>

Table 9: Syntax and semantics of \( L \)-statements

Using the translation of Table 9 and the axioms of BPA_{δλ} 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.
Proof EQ7

while $e_b \text{ do } x \text{ od } < \neg e_b \triangleright y = \varepsilon < \neg e_b \triangleright y$

EQ8 while $e_b \text{ do } x \text{ od } = \text{if } e_b \text{ then } x \text{ fi } ; \text{ while } e_b \text{ do } x \text{ od}$

EQ9 $\lambda_x (\text{if } e_b \text{ then } x \text{ fi }) = \text{if } s \circ e_b \text{ then } \lambda_x (x) \text{ fi}$

EQ10 $\lambda_y (\text{if } e_b \text{ then } x \text{ else } y \text{ fi }) = \text{if } s \circ e_b \text{ then } \lambda_y (x) \text{ else } \lambda_y (y) \text{ fi}$

EQ11 $\lambda_y (\text{while } e_b \text{ do } x \text{ od }) = \lambda_y (x \cdot \text{while } e_b \text{ do } x \text{ od }) < \neg e_b \triangleright \varepsilon$

EQ12 while $e_b \text{ do } x \text{ od } = \text{while } e_b \text{ do } x \text{ od } ; \text{ while } e_b \text{ do } x \text{ od}$

EQ13 if $e_b \text{ then } x ; z \text{ else } y ; z \text{ fi } = \text{if } e_b \text{ then } x \text{ else } y \text{ fi } ; z$

Table 10: Examples of program equalities
Proof EQ8

\[
\text{while } e_b \text{ do } x \text{ od} = \{ \text{Translation of statements according to Table 9} \} \\
(x \prec e_b \succ \delta)^* \cdot (x \prec \neg e_b \succ \delta)
\]

= \{ Axiom R1 \}

\[
(\varepsilon + (x \prec e_b \succ \delta) \cdot (x \prec e_b \succ \delta)^*) \cdot (x \prec \neg e_b \succ \delta)
\]

= \{ Axiom A4 \}

\[
\varepsilon \cdot (x \prec \neg e_b \succ \delta) + ((x \prec e_b \succ \delta) \cdot (x \prec e_b \succ \delta)^*) \cdot (x \prec \neg e_b \succ \delta)
\]

= \{ Axioms A9 and A5 \}

\[
(x \prec \neg e_b \succ \delta) + (x \prec e_b \succ \delta) \cdot (x \prec e_b \succ \delta)^* \cdot (x \prec \neg e_b \succ \delta)
\]

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

\[
(x \prec \neg e_b \succ \delta) + (x \prec e_b \succ \delta) \cdot \text{while } e_b \text{ do } x \text{ od}
\]

= \{ Equality EQ7 \}

\[
(\text{while } e_b \text{ do } x \text{ od} \prec \neg e_b \succ \delta) + (x \prec e_b \succ \delta) \cdot \text{while } e_b \text{ do } x \text{ od}
\]

= \{ Axiom A7 \}

\[
(\text{while } e_b \text{ do } x \text{ od} \prec \neg e_b \succ \delta \cdot \text{while } e_b \text{ do } x \text{ od}) + (x \prec e_b \succ \delta) \cdot \text{while } e_b \text{ do } x \text{ od}
\]

= \{ Axiom A9 \}

\[
(\varepsilon \cdot \text{while } e_b \text{ do } x \text{ od} \prec \neg e_b \succ \delta \cdot \text{while } e_b \text{ do } x \text{ od}) + (x \prec e_b \succ \delta) \cdot \text{while } e_b \text{ do } x \text{ od}
\]

= \{ Axiom C7 \}

\[
(x \prec \neg e_b \succ \delta) \cdot \text{while } e_b \text{ do } x \text{ od} + (x \prec e_b \succ \delta) \cdot \text{while } e_b \text{ do } x \text{ od}
\]

= \{ Axioms A4 and A1 \}

\[
((x \prec e_b \succ \delta) + (x \prec \neg e_b \succ \delta)) \cdot \text{while } e_b \text{ do } x \text{ od}
\]

= \{ Axiom C3 \}

\[
(x \prec e_b \succ \delta) \cdot \text{while } e_b \text{ do } x \text{ od}
\]

= \{ Translation of if-then and ‘;’ statements according to Table 9 \}

\[
\text{if } e_b \text{ then } x \text{ fi ; while } e_b \text{ do } x \text{ od}
\]

Proof EQ9

\[
\lambda_s(\text{if } e_b \text{ then } x \text{ fi}) = \{ \text{Translation of if-then statements according to Table 9} \}
\]

\[
\lambda_s(x \prec e_b \succ \varepsilon)
\]

= \{ Axiom C12 \}

\[
\lambda_s(x) \prec s \circ e_b \succ \lambda_s(\varepsilon)
\]

= \{ Axiom SO3 \}

\[
\lambda_s(x) \prec s \circ e_b \succ \varepsilon
\]

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

\[
\text{if } s \circ e_b \text{ then } \lambda_s(x) \text{ fi}
\]
Proof EQ10
\[ \lambda_s(\text{if } e_b \text{ then } x \text{ else } y \text{ fi}) \]
= \{ Translation of if-then-else statements according to Table 9 \}
\[ \lambda_s(x < e_b \triangleright y) \]
= \{ Axiom C12 \}
\[ \lambda_s(x) \triangleleft s \circ e_b \triangleright \lambda_s(y) \]
= \{ Translation of if-then-else statements according to Table 9 \}
if \( s \circ e_b \) then \( \lambda_s(x) \) else \( \lambda_s(y) \) fi

Proof EQ11
\[ \lambda_s(\text{while } e_b \text{ do } x \text{ od}) \]
= \{ Equation EQ8 \}
\[ \lambda_s(\text{if } e_b \text{ then } x \triangleright \text{ while } e_b \text{ do } x \text{ od}) \]
= \{ Translation of statements according to Table 9 \}
\[ \lambda_s((x < e_b \triangleright \varepsilon) \cdot \text{while } e_b \text{ do } x \text{ od}) \]
= \{ Axiom C7 \}
\[ \lambda_s((x \cdot \text{while } e_b \text{ do } x \text{ od} < e_b \triangleright \varepsilon) \cdot \text{while } e_b \text{ do } x \text{ od}) \]
= \{ Axiom C3 \}
\[ \lambda_s((x \cdot \text{while } e_b \text{ do } x \text{ od} < e_b \triangleright \delta) + (\varepsilon \cdot \text{while } e_b \text{ do } x \text{ od} < \neg e_b \triangleright \delta)) \]
= \{ Axiom A9 \}
\[ \lambda_s((x \cdot \text{while } e_b \text{ do } x \text{ od} < e_b \triangleright \delta) + (\text{while } e_b \text{ do } x \text{ od} < \neg e_b \triangleright \delta)) \]
= \{ Equality EQ7 \}
\[ \lambda_s((x \cdot \text{while } e_b \text{ do } x \text{ od} < e_b \triangleright \delta) + (\varepsilon \cdot \neg e_b \triangleright \delta)) \]
= \{ Axiom C3 \}
\[ \lambda_s(x \cdot \text{while } e_b \text{ do } x \text{ od} < e_b \triangleright \varepsilon) \]
= \{ Axiom C12 \}
\[ \lambda_s(x \cdot \text{while } e_b \text{ do } x \text{ od}) < s \circ e_b \triangleright \lambda_s(\varepsilon) \]
= \{ Axiom SO3 \}
\[ \lambda_s(x \cdot \text{while } e_b \text{ do } x \text{ od}) < s \circ e_b \triangleright \varepsilon \]

Proof EQ12
while \( e_b \text{ do } x \text{ od} \)
= \{ Translation of statements according to Table 9 \}
\[ ((x < e_b \triangleright \delta)^* \cdot (\varepsilon < \neg e_b \triangleright \delta)) \]
= \{ Axiom C5 \}
\[ (x < e_b \triangleright \delta)^* \cdot ((\varepsilon < \neg e_b \triangleright \delta) < \neg e_b \triangleright \delta) \]
= \{ Axioms A9 and A7 \}
\[ (x < e_b \triangleright \delta)^* \cdot (\varepsilon \cdot (\varepsilon < \neg e_b \triangleright \delta) < \neg e_b \triangleright \delta \cdot (\varepsilon < \neg e_b \triangleright \delta)) \]
= \{ Axiom C7 \}
\[ (x < e_b \triangleright \delta)^* \cdot (\varepsilon \cdot (x < e_b \triangleright \delta)^* < \neg e_b \triangleright \delta \cdot (x < e_b \triangleright \delta)^*) \cdot (\varepsilon < \neg e_b \triangleright \delta) \]
\[ f \]

\section{Conclusions}
We have shown that ACP-style process algebras \cite{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 \( \text{bpa} \) \cite{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 \( \text{if.. then.. else.. fi} \) and \( \text{while.. do.. od} \). 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 on 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


